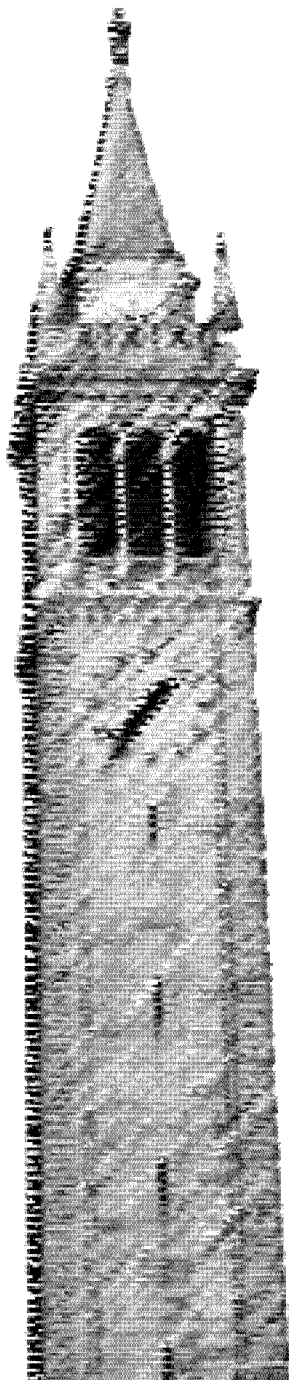


Decentralized Ledgers: Design and Applications

Yuncong Hu



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-104

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-104.html>

May 13, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Decentralized Ledgers: Design and Applications

by

Yuncong Hu

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Raluca Popa, Co-chair
Associate Professor Alessandro Chiesa, Co-chair
Professor Xiaodong Song
Associate Professor Elaine Shi

Spring 2022

Decentralized Ledgers: Design and Applications

Copyright 2022
by
Yuncong Hu

Abstract

Decentralized Ledgers: Design and Applications

by

Yuncong Hu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Associate Professor Raluca Popa, Co-chair

Associate Professor Alessandro Chiesa, Co-chair

A vast majority of online services nowadays are built atop centralized systems, in which data is stored and managed by a trusted party. However, centralization brings significant drawbacks, such as a central point of attack, poor transparency, and poor auditability. Recent success in the field of blockchain has led to great interest in secure decentralized ledgers because they enable a set of heterogeneous parties to reach a consensus on the validity of data on ledgers without relying on centralized service providers. In addition, decentralized ledgers often promise a foundation of decentralized trust and auditability for applications. Unfortunately, the benefits of decentralized ledgers often come at the expense of privacy and efficiency.

In this dissertation, I will present my work on secure and efficient decentralized ledgers. I will show how to improve the efficiency and privacy of decentralized ledgers, which involves the design of new cryptographic tools such as zero-knowledge proofs and authenticated data structures. I will also show how to leverage decentralized ledgers to build practical and secure systems for real-world applications such as the Internet of Things and file sharing services.

To my family, friends, mentors, and colleagues.

Contents

Contents	ii
List of Figures	v
List of Tables	viii
1 Introduction	1
1.1 Merkle ² : An efficient transparency log system	2
1.2 Marlin: A more efficient zero-knowledge proof system	2
1.3 Gemini: An efficient zero-knowledge proof system for large instances	3
1.4 Ghostor: A secure and practical decentralized data-sharing system	3
1.5 JEDI: A secure and efficient decentralized messaging system for IoT	4
1.6 NIDAR: An efficient anonymous routing based on decentralized trust	4
2 Merkle²: A Low-Latency Transparency Log System	6
2.1 Introduction	7
2.2 System overview	11
2.3 Threat model and security guarantee	13
2.4 Merkle ² 's data structure	15
2.5 Monitoring protocol in Merkle ²	19
2.6 Lookup protocol in Merkle ²	24
2.7 Applications of Merkle ²	26
2.8 Implementation and evaluation	28
2.9 Related works	35
2.10 Security of Merkle ²	37
2.11 Optimizations	42
3 MARLIN: Preprocessing zkSNARKs with Universal and Updatable SRS	45
3.1 Introduction	46
3.2 Techniques	55
3.3 Preliminaries	65
3.4 Algebraic holographic proofs	66

3.5	AHP for constraint systems	69
3.6	Polynomial commitment schemes with extractability	83
3.7	Preprocessing arguments with universal SRS	90
3.8	From AHPs to preprocessing arguments with universal SRS	93
3.9	MARLIN: an efficient preprocessing zkSNARK with universal SRS	102
4	Gemini: Elastic SNARKs for Diverse Environments	106
4.1	Introduction	107
4.2	Techniques	110
4.3	Preliminaries	132
4.4	Streaming model	135
4.5	Tensor-product protocol	138
4.6	Elastic protocols for scalar products	143
4.7	A non-holographic protocol for R1CS	150
4.8	Achieving holography	154
4.9	Polynomial commitment schemes	163
4.10	Elastic argument systems	166
5	Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust	170
5.1	Introduction	171
5.2	System Overview	176
5.3	Threat Model and Security Guarantees	178
5.4	Hiding User Identities	180
5.5	Achieving Verifiable Consistency	183
5.6	Mitigating Resource Abuse	187
5.7	Applying Ghostor to Applications	189
5.8	Implementation	193
5.9	Evaluation	194
5.10	Related Work	201
6	JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT	203
6.1	Introduction	204
6.2	JEDI's Model and Threat Model	210
6.3	End-to-End Encryption in JEDI	213
6.4	Integrity in JEDI	219
6.5	Revocation in JEDI	222
6.6	Implementation	226
6.7	Evaluation	228
6.8	Related Work	237
7	Non-Interactive Differentially Anonymous Router	239
7.1	Introduction	240

7.2	Definitions and Preliminaries	245
7.3	Two-Layer NIDAR Construction	248
7.4	Multi-Layer NIDAR	266
	Bibliography	280

List of Figures

2.1	System overview of Merkle ² . Shaded areas indicate components introduced by Merkle ² .	11
2.2	The center tree is the chronological tree in which internal nodes store root hashes of prefix trees. We denote H the hash value of the node, and $Root$ the root hash of the Merkle prefix tree. Each leaf in the chronological tree has a position number (from 0 to 3). We denote by $[X:Y]$ the node that covers leaves with position number from X to Y . The other two trees are prefix trees with the Merkle root $Root_{[0,1]}$ and $Root_{[0,3]}$ respectively. We denote by $Index$ the index of leaves in prefix trees.	15
2.3	A forest transition starting from one leaf to four leaves. The red nodes indicate the Merkle roots in the digest. Leaves in bold indicate ID-value pairs added in each epoch.	16
2.4	Alice's values are Val_0 and Val_1 . The attacker may add values for the ID "Alice" in the chronological trees [24:27]. The signature chain prevents such attacks.	22
2.5	Alice appends the first value Val for her ID at position 21. Someone (either another honest user or the attacker) has appended Val' for ID "Alice" at position 17 already. Alice will verify the first-value proof, which contains non-membership proofs for ID "Alice" in the prefix trees at the green nodes.	23
2.6	Alice appends the master verifying key as the first value. Following values are signed by the master signing key.	25
2.7	Append time.	29
2.8	Lookup proof size and verification time.	29
2.9	Monitoring cost for auditors in AAD and Merkle ² .	29
2.10	Monitoring cost for ID owners in CONIKS.	29
2.11	Monitoring cost for ID owners in Merkle ² .	29
2.12	Memory cost.	32
2.13	Server throughput.	32
2.14	Users supported by one server.	32
2.15	The progression of Merkle ² through 4 appends. The maximum height is set to $H = 2$. The green nodes represent the most recent appends to Merkle ² . Red indicates the internal nodes whose prefix trees the ID-value pair was appended to.	42
2.16	Blue nodes in the left tree will be compressed into one node in the right tree because they are all nodes (internal or leaf) with at most one child.	43
2.17	Node X will be split into three nodes (A , B , C). The new ID-value pair will be stored in node C .	44

3.1	Comparison of two preprocessing zkSNARKs with universal (and updatable) SRS: the prior state of the art and our construction. We include the current state of the art for circuit-specific SRS (in gray), for reference. Here $\mathbb{G}_1/\mathbb{G}_2/\mathbb{F}_q$ denote the number of elements or operations over the respective group/field; also, $f\text{-MSM}(m)$ and $v\text{-MSM}(m)$ denote fixed-base and variable-base multi-scalar multiplications (MSM) each of size m , respectively. The number of pairings that we report for Sonic’s verifier is lower than that reported in [MBKM19] because we account for standard batching techniques for pairing equations.	47
3.2	Measured performance of MARLIN and [Gro16] over the BLS12-381 curve. We could not include measurements for [MBKM19, Sonic] because at the time of writing there is no working implementation of its unhelped variant.	48
3.3	Diagram of our methodology to construct preprocessing SNARGs with universal SRS.	49
3.4	AHP for the lincheck problem.	81
3.5	AHP for R1CS.	82
3.6	Our approach to construct polynomial commitment schemes.	86
4.1	A streaming algorithm for computing the coefficients of $\mathbf{f}^{(j)}$ from $\mathbf{f}^{(0)} := \mathbf{f}$. Nodes in blue denote the coefficients that are stored in memory at any moment.	118
4.2	Running time (above) and memory usage (below) for the elastic prover in the <i>preprocessing</i> protocol (blue) and the <i>non-preprocessing</i> protocol (red), for different R1CS sizes with $N = M$. The black squares indicate the size for which the time-efficient prover triggers an out-of-memory crash (it uses too much memory).	131
4.3	On the left-hand side, the stream for generating the coefficients of all <i>all</i> folded polynomials $\mathbf{f}^{(0)}, \dots, \mathbf{f}^{(n)}$, as a pair composed of the current round number, and the next coefficient. On the right, the stream for generating coefficients of the vector $\mathbf{f}^{(j)}$, given $\mathcal{S}(\mathbf{f})$ and $\boldsymbol{\rho} = (\rho_0, \dots, \rho_{n-1})$ with $n \geq j$	140
4.4	Streams $\mathcal{S}(\mathbf{a}^*), \mathcal{S}(\mathbf{b}^*), \mathcal{S}(\mathbf{c}^*)$, and $\mathcal{S}(\mathbf{y})$	152
4.5	Stream of the vectors $\mathbf{z}^*, \mathbf{r}_A^*, \mathbf{r}_B^*, \mathbf{r}_C^*$. $\mathcal{S}_{\text{cmaj}}(U)$ produces the sparse representation triplets (row, col, value) in column-major; we use pattern-matching with “_” to assign the value we are interested in.	156
4.6	Stream of the vectors for the lookup protocol.	159
4.7	Linear-time algorithm for generating the timestamp vectors in Construction 9 [BEGKN91; Set20].	160
5.1	An example of what a server attacker sees in a typical end-to-end encrypted (E2EE) system versus Ghostor’s Anonymous E2EE	172
5.2	Information leakage in a data-sharing system and associated privacy properties	172
5.3	Ghostor’s contributions. Ghostor’s techniques can be applied to both oblivious and non-oblivious systems.	173
5.4	System overview of Ghostor. Shaded areas indicate components introduced by Ghostor.	176
5.5	Object layout in Ghostor	181
5.6	Blind signature	194

5.7	YCSB workloads (R: read, W: write)	194
5.8	Operations for verification	195
5.9	Latency measurements	197
5.10	Benchmarks comparing throughput of the six setups described in Section 5.9.2	197
5.11	Microbenchmarks of PoW mechanism and Tor	199
5.12	Ghostor-MH	199
5.13	End-to-end latencies of client-side operations	200
6.1	IoT comprises a diverse set of devices, which span more than four orders of magnitude of computing power (estimated in Dhrystone MIPS). ¹	205
6.2	JEDI keys can be qualified and delegated, supporting decentralized, cryptographically-enforced access control via key delegation. Each person has a decryption key for the indicated resource subtree that is valid until the indicated expiry time. Black arrows denote delegation.	206
6.3	Applying JEDI to a smart buildings IoT system. Components introduced by JEDI are shaded. The subscriber's key is obtained via JEDI's decentralized delegation (Figure 6.2).	211
6.4	Pattern S used to encrypt message sent to a/b on June 08, 2017 at 6 AM. The figure uses 8 slots for space reasons; JEDI is meant to be used with more slots (e.g., 20).	215
6.5	Key management of the CS method. Red nodes indicate nodes associated with revoked leaves. The green node is the root of the subtree covering unrevoked leaves.	223
6.6	Performance of JEDI's cryptography	229
6.7	Critical-path operations in bw2, with/without JEDI	230
6.8	Occasional bw2 operations, with and without JEDI	231
7.1	An R -way butterfly network when the recursion is fully expanded. In this example, $n_{\text{Layer}} = 3$, $2n/Z = 27$, and $R = (2n/Z)^{1/n_{\text{Layer}}} = 3$. The elements are being routed from level 1 to level L , and the onion layers of encryption are performed in the reverse order where level 1 is the outer-most layer. Each small box \square denotes a bucket, and each dashed big box is either a RowPerm or a ColPerm instance. The last level is special and employs ColPerm instances which route the real elements to the front in a random order.	270

List of Tables

2.1	Asymptotic costs of the server in Merkle ² against other systems. n refers to the number of entries in the log, λ is the security parameter for AAD, E is the number of epochs, and P is the number of appends between epochs. Red indicates the worst performance in the category. For the storage cost, we measure the number of nodes in data structures throughout the system life. For the monitoring cost, the auditor column refers to the size of proof provided to each auditor per epoch; the owner column refers to the size of proof provided to each data owner for each log entry throughout the system life. We do not consider “collective verification” for ECT since it relies on a different threat model. The original CONIKS design copies and reconstructs the whole data structure in each epoch to enable the data owners, who go offline, to verify their data in epochs they missed. With CONIKS*, we optimize CONIKS by leveraging persistent data structures [DSST86], which we discuss in Section 2.8.	9
2.2	Latency (in ms).	31
2.3	Message size of server’s responses.	31
3.1	Comparison of the <i>non</i> -holographic protocol for RICS in [BCRSVW19], and the AHP for RICS that we construct. Here n denotes the number of variables and m the number of non-zero coefficients in the matrices.	50
3.2	Efficiency of our polynomial commitment schemes. Here f-MSM(m) and v-MSM(m) denote fixed-base and variable-base multi-scalar multiplications (MSM) each of size m , respectively. All MSMs are carried out over \mathbb{G}_1 . For simplicity we assume above that the query set evaluates each polynomial at the same point. If there are multiple points in the set, then proof size and time for checking proofs scales linearly with the number of points. Furthermore, we assume above that the n committed polynomials all have degree d	86
5.1	Our goals and how Ghostor achieves each one	175
5.2	Per-object keys in Ghostor. The server uses the global signing keypair (SVK, SSK) to sign digests for objects.	180
5.3	A digest for an operation in Ghostor	183
6.1	Latency of JEDI’s implementation of BLS12-381	228
6.2	CPU and power costs on the Hamilton platform	232

6.3	Average current and expected battery life (for 1400 mAh battery) for sense-and-send, with varying sample interval	233
6.4	Comparison of JEDI with other crypto-based IoT/cloud systems	235

Acknowledgments

I cannot accomplish my Ph.D. and dissertation without the support of many people. First and foremost I am extremely grateful to my supervisors, Prof. Raluca Ada Popa and Prof. Alessandro Chiesa for their invaluable advice, continuous support, and patience during my Ph.D. study. Their immense knowledge and plentiful experience have greatly improved every aspect of my research skills and philosophy, ranging from picking good research problems, to thinking about how to solve them, to writing clear and rigorous papers, and finally to presenting our research. I hope that I could pass on to my students even a fraction of the things that I have learnt from the two of you.

I also was very fortunate to have worked with other faculty at UC Berkeley, CMU, and NTT Research, including Prof. David E. Culler, Prof. Elaine Shi, and Prof. Shin'ichiro Matsuo. These faculty gave me valuable feedback on my work, and inspired my research to be more interdisciplinary. I particularly appreciate Prof. Elaine Shi for her help since my undergraduate.

None of my research would be possible without my amazing co-authors, listed here in alphabetical order: Michael P Andersen, Jonathan Bootle, Benedikt Bünz, Alessandro Chiesa, David E. Culler, Kian Hooshmand, Harika Kalidhindi, Sam Kumar, Mary Maller, Shin'ichiro Matsuo, Pratyush Mishra, Michele Orrù, Raluca Ada Popa, Elaine Shi, Noah Vesely, Nicholas P. Ward, Seung Jin Yang and Mingxun Zhou.

I also had the privilege of interacting with many cool people in UC Berkeley and the RISE Lab, including Weikeng Chen, Xinyun Chen, Zihao Chen, Emma Dauterman, Ankur Dave, Ioannis Demertzis, Vivian Fang, Hai Huang, Sukrit Kalra, Jingcheng Liu, Jianan Lu, Peihan Miao, Dev Ojha, Rishabh Poddar, Deevashwer Rathee, Mayank Rathee, Jeongseok Son, Katerina Sotiraki, Chia-Che Tsai, Sameer Wagh, Lun Wang, Xin Wang, Jean-Luc Watson, Tiancheng Xie, Jiaheng Zhang, Hong Zhang, Shuang Zhang, Lianmin Zheng, Wenting Zheng, Li Zhu, and Yuan Zhu. Besides, I would also thank my other friends, who helped make bay area my home away from home: Haojia Guo, Yuli Han, Dinglong Li, Xueting Liu, Haojun Ma, Haobin Ni, Shay Shao, Yin Sheng, Ke Wu, Siqu Yao, Xihu Zhang, and Kaixin Zheng. I particularly thank Weikeng Chen, who has been my roommate since the first year. I also thank our landlords Diane and David for their lovely apartment.

Last but not least, I want to thank my parents and family for their continuous support and encouragement throughout grad school! I particularly thank my aunt Lixin and uncle Derrick, who have always welcomed me into their home in Oregon with open arms.

Chapter 1

Introduction

A vast majority of online services nowadays are built atop centralized systems, in which data is stored and managed by a trusted party. However, centralization brings significant drawbacks, such as a central point of attack, poor transparency, and poor auditability. In particular, systems with a central point of attack allow the attacker to compromise a central point to break the whole system; thus, there are massive data breaches on centralized systems each year.

Recent success in the field of blockchain has led to great interest in secure decentralized ledgers because they enable a set of heterogeneous parties to reach a consensus on the validity of data on ledgers without relying on centralized service providers. In addition, decentralized ledgers often promise a foundation of decentralized trust and auditability for applications. For example, certificate transparency [LLK13b] provides a trustworthy web certificates infrastructure through auditing a centralized service provider in a decentralized way; decentralized finance [WPGKHK21] is a blockchain-based form of finance that does not rely on central financial intermediaries. Unfortunately, the benefits of decentralized ledgers often come at the expense of privacy and efficiency. For example, Bitcoin supports only five transactions per second [Nak08] (orders of magnitude slower than centralized ledgers) and suffers from privacy problems due to public transactions [Ben+14]; prior transparency logs support only a few users due to the expensive auditing.

My research focuses on secure and efficient decentralized ledgers. In particular, in my graduate work, I have both designed new cryptographic tools to improve the privacy and efficiency of decentralized ledgers and built practical systems with decentralized trust from decentralized ledgers.

Within the category of improving the privacy and efficiency of decentralized ledgers, **Merkle**² [HHKYP21] solves the efficiency problem by constructing a new authenticated data structure and building a more efficient decentralized ledger. As a result, Merkle² is able to support faster updates and 100x more users than prior state-of-the-art systems. To protect privacy, I design and build new zero-knowledge proof systems that support various applications in decentralized ledgers. In particular, **Marlin** [CHMMVW20] provides better usability and efficiency than prior proof systems; thus, it has been widely used in blockchain companies such as Aleo [Ale] to build decentralized applications. In addition, **Gemini** [BCHO22] is able to prove much larger instances than existing proof systems since computations on decentralized ledgers become much more complex.

Within building systems with decentralized trust from decentralized ledgers, **Ghostor** [HKP20]

is a practical data-sharing system with strong security guarantees based on decentralized trust. Existing practical data-sharing systems either cannot provide strong security guarantees as Ghostor or rely on centralized trust. **JEDI** [KHAPC19] provides a secure decentralized messaging system for IoT devices and has been used in UC Berkeley campus [And+19] running for more than two years, with more than 800 IoT devices. Finally, **NIDAR** [BHMS21] is an efficient non-interactive anonymous routing system without centralized assumptions.

1.1 Merkle²: An efficient transparency log system

Transparency logs are proposed to avoid the high overhead of blockchain-based decentralized ledgers and are widely used in building trustworthy certificates [LLK13b] or public key infrastructures [MBBFF15]. Unfortunately, prior transparency log systems suffer from high update latency due to the expensive monitoring algorithm. For example, in key transparency [cite], users may need to wait for an hour to be able to start using the service or revoke compromised keys.

My project Merkle² [HHKYP21] described in Chapter 1 is a low-latency transparency log system. To achieve that, Merkle² first constructs a new multi-dimensional, authenticated data structure that nests two types of Merkle trees. Based on this data structure, Merkle² then builds a transparency log system with efficient monitoring protocols that enables low-latency updates. In particular, all the operations in Merkle² are independent of update latency and are (poly)logarithmic to the number of entries in the log. Merkle² not only has excellent asymptotics when compared to prior work but is also efficient in practice. Our evaluation shows that Merkle² propagates updates in as little as 1 second and can support 100× more users than state-of-the-art transparency logs. Merkle² is open-source [HHKYP] and can be used as a source of decentralized trust for various applications. For example, Merkle² can be used in my projects Ghostor and JEDI as an efficient decentralized ledger.

1.2 Marlin: A more efficient zero-knowledge proof system

Zero-knowledge proofs enable the proof of computation without revealing users' secrets; thus, they are widely used in decentralized ledgers to protect privacy [Ben+14]. However, most prior zero-knowledge proofs may suffer from deployability and usability problems since they require different setups for different applications. For example, in blockchains, whenever people construct a new smart contract, they have to run an expensive setup for it. Recent research also proposed zero-knowledge proofs without a setup; unfortunately, they often result in non-succinct proof size or verification, which are critical in decentralized ledgers since the resources are constrained.

My project Marlin [CHMMVW20] described in Chapter 2 is a universal zero-knowledge proof system in which a single setup suffices to prove various computations. Furthermore, Marlin supports succinct proof size and verification so that it consumes fewer resources of decentralized ledgers. Marlin first proposes a novel protocol enabling fast verification. Then Marlin achieves succinct proof size by leveraging polynomial commitments, which requires only a single setup for multiple

1.3. *GEMINI: AN EFFICIENT ZERO-KNOWLEDGE PROOF SYSTEM FOR LARGE INSTANCES*

applications. Finally, Marlin formalizes the above methodology and constructs a cryptographic compiler, which has been used as a standard tool for building new zero-knowledge proofs in the following works.

Marlin’s implementation is open-source [CHMMVW] and has been widely used in blockchain companies, such as Aleo [Ale], for building decentralized applications. In particular, Marlin can be used in my other project Ghostor to achieve a stronger privacy guarantee for data-sharing. Marlin’s library is further integrated into Arkworks [ark], a rust ecosystem for implementing and developing future zero-knowledge proofs.

1.3 Gemini: An efficient zero-knowledge proof system for large instances

Zero-knowledge proofs can also be used to improve the scalability and efficiency of decentralized ledgers since they enable verifiable computations with succinct proofs and verification. As more and more data are stored on decentralized ledgers, the prover has to prove much larger instances with more complicated computations. However, prior proof systems, including Marlin, suffer from high proving time and memory costs.

To solve this problem, in Chapter 3 we proposed a new type of proof system called Gemini [BCHO22]. When the instance size is small, Gemini achieves a faster prover than Marlin; for a large instance, Gemini provides a “slim” prover that consumes only (poly)logarithmic size of memory and thus supports much larger instances and more complicated computations. At its core is an elastic protocol that can be run in diverse environments. As a result, Gemini can prove several orders of magnitude more complex computations than prior proving systems using the same computation resources. We implement Gemini based on Arkworks libraries and integrate it into the Arkworks ecosystem to support future zero-knowledge proofs design.

1.4 Ghostor: A secure and practical decentralized data-sharing system

Data-sharing systems are used to store and share sensitive data and have seen widespread adoption over the past decade. As a result, both academia and industry have proposed numerous solutions to protect user privacy and data integrity from a compromised server. Decentralized ledgers have been proposed as a possible solution as they eliminate the need of centralized trust. Unfortunately, naively applying decentralized ledgers can introduce extremely high overheads and result in an impractical system. For example, a strawman solution publishing every single operation on ledgers may incur high latency and expensive costs.

My project Ghostor [HKP20] described in Chapter 4 is a practical secure data-sharing system based on decentralized trust. Ghostor leverages a decentralized ledger rarely, publishing only a single hash to the ledger for the entire system once every epoch. Ghostor achieves two strong security

1.5. *JEDI: A SECURE AND EFFICIENT DECENTRALIZED MESSAGING SYSTEM FOR IOT 4*

guarantees, anonymity and verifiable linearizability, to protect sensitive data on an untrusted server. At a high level, anonymity means that the protocol does not reveal directly to the server any user identity with any operation, and verifiable linearizability means users can verify that each write is reflected in later reads, except for benign reordering of concurrent operations as formalized by linearizability. Many prior systems with such strong privacy and integrity guarantees either are too slow to be practical or rely on centralized assumptions, such as trusted hardware [CD16]. In contrast, Ghostor is a practical system, which provides strong privacy and integrity guarantees and uses only decentralized trust. Overall, Ghostor completes a single operation in 30ms, whereas the strawman solution may require one hour for blockchains to confirm the transaction. When compared with a simplistic and completely insecure baseline, Ghostor brings a 4-5x throughput overhead. Although significant, Ghostor's overhead may be worth it for security- and privacy-sensitive applications.

1.5 JEDI: A secure and efficient decentralized messaging system for IoT

Developing secure communication for IoT devices is of paramount importance because these devices collect private information about users. Unfortunately, traditional systems rely on centralized services, which suffer from a central point of attack and are not scalable for IoT-scale systems.

To solve this problem, my project JEDI [KHAPC19] described in Chapter 5 provides a secure decentralized messaging system for IoT devices. The core of JEDI is a new cryptographic encryption protocol supporting many-to-many communication and decentralized key delegations. In JEDI, the namespace administrator can grant other users access by generating the corresponding secret keys. However, JEDI supports decentralized delegations so that a user can further derive and delegate sub-keys to achieve fine-grained access control without help from the administrator. Decentralized delegations are critical for large-scale IoT systems, which may involve thousands of devices. Moreover, JEDI's protocol is lightweight and practical for ultra low-power devices. Finally, JEDI's implementation is open source [KHAPC], and it has been used in UC Berkeley campus [And+19] running for more than two years, with more than 800 IoT devices.

1.6 NIDAR: An efficient anonymous routing based on decentralized trust

Most existing anonymous routing systems rely on centralized assumptions, meaning that at least one trusted service provider exists. One recent work [SW21a] first proposed an anonymous routing with fully untrusted servers. However, this system suffers from quadratic computation cost, which is not scalable for decentralized applications.

My project NIDAR [BHMS21] described in Chapter 6 solves this problem with sub-quadratic computation cost. To achieve that, NIDAR proposes a non-interactive differential private permutation algorithm and combines it with multi-client functional encryption. As a result, NIDAR can be run

1.6. *NIDAR: AN EFFICIENT ANONYMOUS ROUTING BASED ON DECENTRALIZED TRUST*

on a single fully untrusted server and provide network-level protection for various decentralized systems and applications. In particular, my project Ghostor can leverage NIDAR instead of Tor to achieve better network-level anonymity.

Chapter 2

Merkle²: A Low-Latency Transparency Log System

Transparency logs are designed to help users audit untrusted servers. For example, Certificate Transparency (CT) enables users to detect when a compromised Certificate Authority (CA) has issued a fake certificate. Practical state-of-the-art transparency log systems, however, suffer from high monitoring costs when used for low-latency applications. To reduce monitoring costs, such systems often require users to wait an hour or more for their updates to take effect, inhibiting low-latency applications. We propose Merkle², a transparency log system that supports both efficient monitoring and low-latency updates. To achieve this goal, we construct a new multi-dimensional, authenticated data structure that nests two types of Merkle trees, hence the name of our system, Merkle². Using this data structure, we then design a transparency log system with efficient monitoring and lookup protocols that enables low-latency updates. In particular, all the operations in Merkle² are independent of update intervals and are (poly)logarithmic to the number of entries in the log. Merkle² not only has excellent asymptotics when compared to prior work, but is also efficient in practice. Our evaluation shows that Merkle² propagates updates in as little as 1 second and can support 100× more users than state-of-the-art transparency logs.

This work was previously published in [HHKYP21].

2.1 Introduction

Interest in transparency logs [MBBFF15; And+19; Lau14; CSPLM15; HB17; LK12; Bon16; Rya14; TBPPTD19; YCR16; CDGM19] has increased in recent years because they promise a trustworthy PKI [Per99; AL99; Mau96] or certificate infrastructure [Lau14; LKL13]. For example, *certificate transparency* (CT) [Lau14; LKL13] enables building accountable PKIs for web applications and is widely deployed. As of March 2021, CT has publicly logged over 12 billion certificates [Gooa]; Google Chrome requires web certificates issued after April 30, 2018 to appear in a CT log [Chr]. Besides CT, there have been significant efforts into *key transparency* [MBBFF15; Bon16; TBPPTD19; Goob; Gooc] for managing the public keys of end-users and *software transparency* [AM18; FDPFSS14; HC17; Nik+17; Syt+16; TD17] for securing software updates.

Transparency logs provide a consistent, immutable, and append-only log: anybody reading the log entries will see the same entries in the same order, nobody can modify the data already in the log, and parties can only append new data. One of their distinctive features is that they combine aspects of blockchains/ledgers [Bita; Eth; Zcaa; Nak19; Woo14; CL99; YMRGA19; PS18] with aspects of traditional centralized hosting. Like blockchains and ledgers, transparency logs rely on *decentralized verification*, enabling anyone to verify their integrity. At the same time, they are hosted traditionally by a *central service provider*, such as Google [Lau14; HB17]. Due to guarantees provided by the log and decentralized verification by third parties, the service provider cannot modify or fork the log without detection. Additionally, centralized hosting enables these logs to be significantly more efficient than Bitcoin-like blockchains; they provide higher throughput and lower latency while avoiding expensive proof of work or the expensive replication of the ledger state at many users.

Common transparency logs are append-only logs that provide an efficient *dictionary* for key-value pairs stored in the log. State-of-the-art transparency logs like CONIKS [MBBFF15] provide the following crucial properties for applications: efficient membership and non-membership proof, and monitoring proof. In particular, when users look up key-value pairs, the server can provide a succinct proof of membership or non-membership to convince users that it returns the correct lookup result. For example, in CT [Lau14], the browser only downloads logarithmic-sized data to check whether a particular website's certificate is in the log. However, unlike CONIKS, CT cannot provide succinct non-membership proofs, so CT cannot support efficient revocation for certificates.

A major impediment to the wider adoption of transparency logs is their high update latency, precluding their use in many low-latency applications. To understand what impacts update latency, one must first understand *monitoring*, a key component of transparency logs. Monitoring allows other parties to monitor the state of an untrusted server and the results it returns to users. The server periodically – every *epoch* – publishes a *digest* summarizing the state of the system. Transparency logs rely on auditors [MBBFF15; And+19; HB17; TBPPTD19] (third-parties or individual users) to keep track of digests published by the server and gossip with each other to prevent server equivocation. Data owners and users who look up data from the log retrieve the digest from the auditors. Given the digest, data owners can check the integrity of their data, and users can check the correctness of the lookup results from the server.

A key challenge in existing, state-of-the-art, practical transparency logs like CONIKS [MBBFF15] or Key Transparency (KT) [HB17] is that *every* data owner must monitor their data for *every*

published digest. For example, Google’s KT proposal [Gooc; Good] cites 1 second as a desirable epoch interval (enabling a variety of applications). For those transparency logs supporting a desired target of 1 billion users, handling each user monitoring every second is too large of a cost for the server both in bandwidth and computation power. For example, if CONIKS runs with a desirable epoch interval of 1 second [Gooc; Good], every year every user has to download about 65 GB of data to check their data. For 1 billion users, the server has to provide about 65 exabytes of data.

This dependence of the server’s cost proportional to the product of users and epochs is what drives existing proposals to set infrequent epochs, e.g., of hours or days. In turn, long epoch intervals affect responsiveness and user experience in applications requiring low-latency updates [Goob]. We consider PKI as an example. Prior transparency logs [MBBFF15] have users wait **an hour** to be able to start using the service, as it takes an epoch for new public keys to appear on the log enabling other users to look them up. However, a study [Nah04] shows that the tolerable waiting time for web users is only **two seconds**. For example, users may not want to wait an hour before being able to register and set up IoT devices that generate SK-PK pairs [And+19; SRSB15; Net+16; KHAPC19]. Also, key owners may not want to wait an hour to revoke compromised keys as the attacker may use the compromised key to steal data or information. Applications like intrusion detection systems [RWV13; Lee+01; SM12] aim to stop incidents and revoke malicious accounts immediately.

To reduce the monitoring cost, researchers proposed transparency logs based on heavy cryptographic primitives such as recursive SNARKs [BSCTV17; BCCT13; COS20a] or bilinear accumulators [TBPPTD19]. However, these works result in high append latency and memory usage. Other work [And+19; Rya14; YCR16; CDGM19] has auditors check every operation on behalf of users, which results in a high overhead on auditors. We elaborate in Sections 2.8 and 2.9.

Hence, this paper asks the question: Is it possible to build a transparency log system that supports both efficient monitoring and low-latency updates? We propose Merkle², a low-latency transparency log system, to answer this question. At the core of Merkle² is a new data structure designed to support efficient append, monitoring, and lookup protocols. As a result, the server can publish digests frequently, and data owners do not need to check each digest published by the server. Moreover, data owners only download polylogarithmic-sized data for monitoring throughout the system life.

We implemented Merkle² and evaluated it on Amazon EC2. Merkle² can sustain an epoch size of 1 second, enabling low-latency applications. For such an epoch size, Merkle² can support up to 8×10^6 users per server machine (Amazon EC2 r5.2xlarge instance), which is 100x greater than CONIKS. For this significant increase in monitoring efficiency, the cost of append and lookup in Merkle² increases only slightly; as a result, for large epoch size, when the monitoring cost of CONIKS is acceptable, CONIKS may perform slightly better than Merkle². For example, when the epoch size is 1 hour, CONIKS can support 2x more users than Merkle². However, such a large epoch size is difficult to accommodate in various low-latency applications. In Section 2.7, we apply Merkle² to certificate and key transparency applications and show the benefits it brings in these settings compared to existing transparency logs. We compare Merkle²’s asymptotic complexity with prior systems in Table 2.1.

Works	Storage Cost	Append Cost	Monitoring Cost		Lookup Cost
			Auditor	Owner	
AAD [TBPPTD19]	λn	$\lambda \log^3(n)$	$\log(n)$	—	$\log^2(n)$
ECT [Rya14]	n	$\log(n)$	$P \log(n)$	—	$\log(n)$
CONIKS [MBBFF15]	$E \cdot n$	$\log(n)$	—	$E \log(n)$	$\log(n)$
CONIKS*	$n \log(n)$	$\log(n)$	—	$E \log(n)$	$\log(n)$
Merkle ²	$n \log(n)$	$\log^2(n)$	$\log(n)$	$\log^2(n)$	$\log^2(n)$

Table 2.1: Asymptotic costs of the server in Merkle² against other systems. n refers to the number of entries in the log, λ is the security parameter for AAD, E is the number of epochs, and P is the number of appends between epochs. Red indicates the worst performance in the category. For the storage cost, we measure the number of nodes in data structures throughout the system life. For the monitoring cost, the auditor column refers to the size of proof provided to each auditor per epoch; the owner column refers to the size of proof provided to each data owner for each log entry throughout the system life. We do not consider “collective verification” for ECT since it relies on a different threat model. The original CONIKS design copies and reconstructs the whole data structure in each epoch to enable the data owners, who go offline, to verify their data in epochs they missed. With CONIKS*, we optimize CONIKS by leveraging persistent data structures [DSST86], which we discuss in Section 2.8.

2.1.1 Overview of our techniques

Background. Merkle² is built upon Merkle trees [Mer79]. Two types of Merkle trees are common in transparency logs: prefix trees (whose leaves are ordered in lexicographic order) as in CONIKS [MBBFF15] and chronological trees (whose leaves are ordered by time of append) as in CT [Lau14]. We elaborate in Section 2.10.3. In prior transparency logs, the server stores data in Merkle trees and publishes the root hash as the digest in every epoch. When users access the data, the server provides an authentication path of the corresponding leaf as proof.

Our data structure. The advantages and limitations of chronological trees and prefix trees are complementary: CT, which is based on chronological trees, does not require users to monitor each digest but cannot provide efficient lookup or revocation; CONIKS can support efficient lookup, but not efficient monitoring. Therefore, a natural question arises: is there a way to combine them to obtain both benefits?

We solve this problem by leveraging ideas from “multi-dimensional” data structures [IKVR88]. Merkle²’s data structure (Section 2.4) consists of nested Merkle trees, with chronological trees in the outer layer. Each internal node of the chronological layer corresponds to a prefix tree, hence the name for our system, Merkle². The hash of each data block is stored in a leaf node of a chronological tree, and, for each node from that leaf to the root of the chronological tree, in its corresponding prefix tree. We provide a “pre-build” technique (Section 2.4.2) to avoid high append time in the worst case.

We show that Merkle²’s data structure has many convenient properties that enable us to design efficient monitoring and lookup protocols. For example, each data block is stored in only $O(\log(n))$ prefix trees, where n is the number of leaves in the chronological layer. Those prefix trees allow us

to look up data blocks based on indices like CONIKS. Also, the structure of the chronological layer allows us to design a monitoring protocol like CT so that data owners do not have to check every digest, but can simply only check the latest digest.

Transparency log system design. By leveraging Merkle²'s data structure, we design the monitoring (Section 2.5) and lookup (Section 2.6) protocols of our transparency log system.

The first problem we want to avoid is data owners having to monitor their data in every digest. To address this problem, we adapt the consistency proof in CT to provide an extension proof for Merkle²'s nested trees. The consistency proof in CT allows auditors to ensure no certificate is removed or modified when the server publishes a new digest. Using it, domain owners do not need to check every digest in CT. We observe that the consistency proof preserves not only the integrity of leaves, but also the integrity of internal nodes. Because each internal node contains a prefix tree, it is paramount that no leaf node values or internal node contents are changed. Thus, we design an extension proof that allows auditors to ensure the integrity of all existing nodes in future system states. We further show that our extension proof allows data owners to verify *only* the latest digest instead of every digest in history, and be assured that they will see earlier modifications to their data. So far, this proof only ensures that the prefix tree root hash in the internal node remains unmodified, but it does not check the actual content in the prefix tree. Therefore, Merkle² has each data owner check contents of $O(n)$ prefix trees.

To further reduce the monitoring cost of each data owner, we require that data owners check only $O(\log(n))$ prefix trees to ensure the membership of their data. Moreover, because of the extension proof guarantee, the data owner needs to check each of these prefix trees *only once* throughout the life of the system. However, by requiring the data owner to check only $O(\log(n))$ prefix trees, we do not prevent attackers from adding corrupted data blocks for an index that does not belong to them. To solve this problem, we co-design *signature chains*, which enable users to verify data block ownership in lookup results. The security of the signature chain relies on the chronological order maintained by chronological trees.

Finally, we design a lookup protocol for users to verify lookup results efficiently. The protocol involves (non-)membership proofs from only $O(\log(n))$ prefix trees that cover all data blocks in the entire system. Moreover, we design an optimized protocol for looking up only the latest append.

2.2 System overview

In this section, we describe the architecture and API of Merkle². Merkle² is a transparency log system consisting of key-value pairs. We refer to these as **ID-value pairs** to avoid confusion with cryptographic keys. As in CONIKS [MBBFF15] and Google Key Transparency [HB17], Merkle² indexes data blocks on the ledger based on their ID to support efficient ID lookup.

2.2.1 System architecture

Merkle²'s system setup (in Fig. 2.1) is similar to that of prior transparency logs. Recall that in transparency log systems, time is split into *epochs*. The system consists of a logical server, auditors, and clients, whose roles are described below.

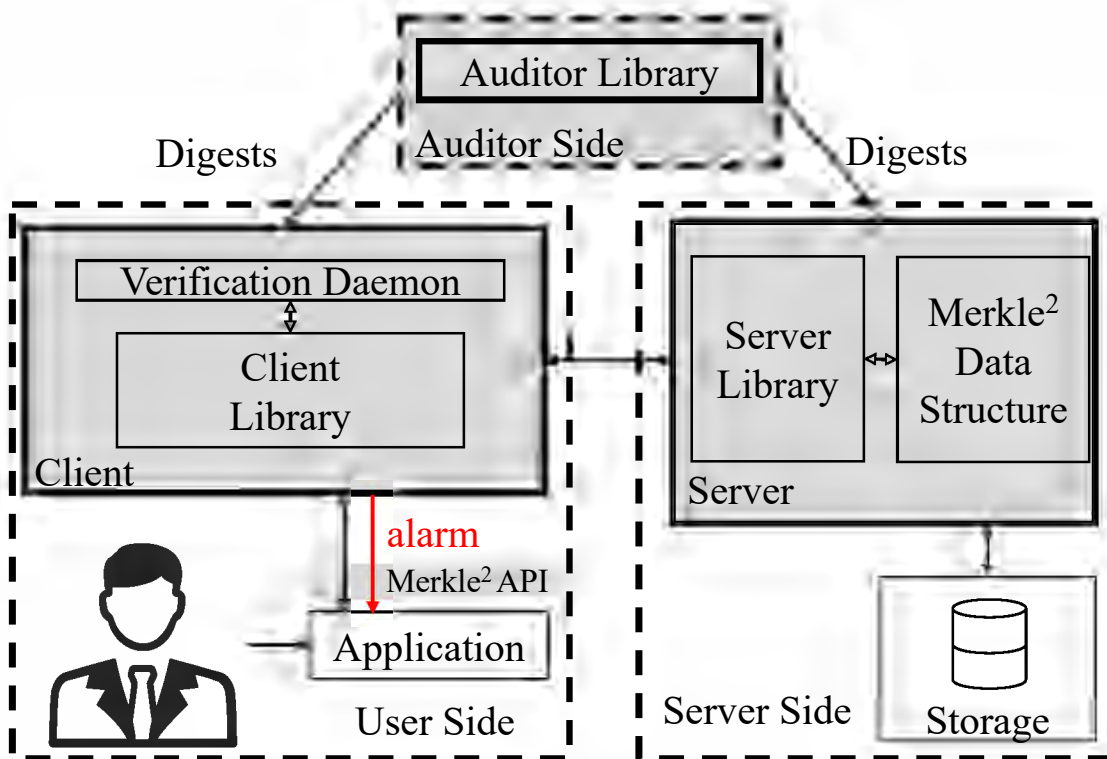


Figure 2.1: System overview of Merkle². Shaded areas indicate components introduced by Merkle².

Server. The (logical) server stores users' ID-value pairs and is responsible for maintaining Merkle²'s data structure and servicing client requests. The server produces proofs for clients and auditors to monitor the system. At the end of each epoch, the server publishes a digest to the auditors summarizing the state of Merkle². Every response provided by the server will be signed.

Auditors. Auditors are responsible for verifying that the server provides to clients and other auditors a consistent view of the state. At the end of every epoch, each auditor requests a server-signed

digest of the overall state from the server along with a short proof. By verifying the proof, the auditor confirms that the state transition from the previous epoch is valid. It is critical that auditors gossip with each other about the digests to make sure they share a consistent view of the system. If two auditors discover different digests for the same epoch, they can prove server's misbehavior by presenting the signature. Anyone can volunteer to serve as an auditor. We present the monitoring protocol for auditors in Section 2.5. Clients fetch digests from multiple auditors and cross check them.

Clients. Users run the client software, including a client library and a verification daemon. Users' applications interact with the client library to append and look up ID-value pairs through the API shown in Section 2.2.2. Each user is responsible for monitoring their own ID-value pairs on the server. We define the owner of an ID as the person who appends the first value for that ID in the system. Only the ID owner is allowed to append new values for that ID. A user can become the owner of multiple IDs. The verification daemon runs as a separate process that regularly monitors the ID-value pairs of the ID owner. After the first append, the daemon will periodically come online and send monitoring requests to the server.

2.2.2 Merkle²'s API

We now explain the API that Merkle² provides to application developers wanting to use a transparency log.

◇ **append**(⟨ID, val⟩): Appends a new value *val* for *ID* to the log. If there does not exist a value for *ID* before the append, the user will be identified as the owner of *ID*. Otherwise, only the owner of *ID* is allowed to append a new value. We describe in Section 2.5.3 how Merkle² enforces this condition. The server adds ⟨*ID*, *val*⟩ to both persistent storage and Merkle²'s data structure. We discuss how to maintain Merkle²'s data structure in Section 2.4.2. The append will not take effect until the server publishes a new digest in the next epoch. For each append, the verification daemon will periodically come online and monitor ⟨*ID*, *val*⟩ on behalf of the *ID* owner to ensure its integrity. Section 2.5 explains the monitoring protocol in detail.

◇ **lookup**(*ID*): Looks up all values for *ID*. The server should return all values for *ID* appended by the *ID* owner so far, sorted in chronological order by append time. The lookup result does not contain values appended after the current epoch; but, in this case, the server can notify the user to send another lookup request in the next epoch. We also introduce an extended API that supports **lookup_latest**, which allows users to look up the latest value for *ID*, because for many applications the most recent value is the only one of interest. This value is the latest append for *ID* in epochs before the lookup operation. To verify a lookup result, clients must fetch the latest digest from auditors and the lookup proof from the server. We discuss the lookup protocol in Section 2.6.

2.3 Threat model and security guarantee

Now, we describe Merkle²'s threat model and guarantees intuitively and then formalize them in Guarantee 1. Merkle² protects the system against a malicious attacker who has compromised the server. A hypothetical attacker can modify data or control the protocol running at the server. In particular, it can fork the views of different users [LKMS04] or return old data. Merkle² guarantees that honest users and auditors will detect these kinds of active attacks.

As in most transparency logs [Lau14; And+19; MBBFF15], we assume that at least some of the auditors are trusted, which means they will verify the digest and proof published at the end of each epoch and detect forks. Intuitively, the requirement is that any user should be able to reach at least one connected and trusted auditor. Each ID owner is responsible for monitoring its own ID-value pairs. Unlike systems that rely on strong ID owners [MBBFF15], ID owners in Merkle² do not need to monitor every epoch. They can choose a monitoring frequency, like once per day or once per week, and monitor updates in between monitoring periods by checking only the latest epoch. If an ID owner monitors the digest in epoch p without detecting a violation, honest users performing a lookup for ID before epoch p (inclusive) are guaranteed to receive the correct values or to detect a violation.

To define Merkle²'s guarantee, we introduce the following conventions. If the owner appends an ID-value pair in epoch E , other users can look it up starting with epoch $E + 1$. The server assigns a *position number* to each ID-value pair to indicate its global order in the system. When users append or lookup, they also receive the position number from the server.

We further define the notion of a *correct lookup result*, as follows. We denote by S_E^{ID} the ordered list of ID-value pairs and corresponding position numbers that the ID owner appended for ID before epoch E , and $S'_E{}^{\text{ID}}$ the ordered list that the user received as the lookup result for ID in epoch E . A correct lookup result for ID in epoch E means that the two lists S_E^{ID} and $S'_E{}^{\text{ID}}$ are identical.

Merkle² provides the following guarantee, for which we provide a proof sketch in Section 2.10.

Guarantee 1. *Assume that the hash function used by Merkle² is a collision-resistant hash function [Gol07] and that the signature scheme is existentially unforgeable [Gol07].*

For any set of users U , for any set of honest auditors A , for any set of append, lookup and monitoring operations by users in U , for any set of honest users $C \in U$, for any ID whose owner is in C , let E_1 be the first epoch in which ID's owner appends the first value for ID, and let $E_2 > E_1$, for each lookup operation for ID performed by a user u in C during epochs $E_1 + 1 \dots E_2$, if

Connectivity conditions:

1. *users in C can reach the server and at least one of the auditors in A ;*
2. *auditors in A can reach the server and all other auditors in A ;*
3. *in every epoch $e \geq 1$, the server outputs the digest of epoch $e - 1$ to all the auditors in A .*

Honesty conditions:

4. *the auditors in A follow the monitoring protocol in Section 2.5 to gossip with each other and check digests in epochs $1 \dots E_2$;*
5. *the owner of ID follows the monitoring protocol in Section 2.5 to check its ID -value pairs in epoch E_2 .*
6. *whenever a user in C looks up ID , it follows the lookup protocol in Section 2.6 for ID ;*

then, if user u did not receive the correct lookup result for ID during $E_1 + 1 \dots E_2$, then at least one of the following parties has detected a server violation: users in C (including the ID owner), or auditors in A .

2.4 Merkle²'s data structure

In this section, we explain Merkle²'s data structure.

2.4.1 Data structure layout

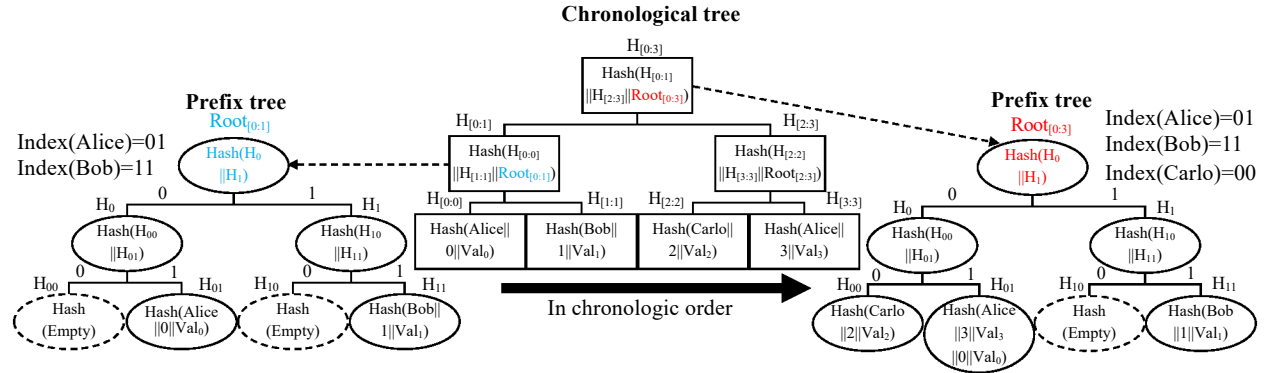


Figure 2.2: The center tree is the chronological tree in which internal nodes store root hashes of prefix trees. We denote H the hash value of the node, and Root the root hash of the Merkle prefix trees. Each leaf in the chronological tree has a position number (from 0 to 3). We denote by $[X:Y]$ the node that covers leaves with position number from X to Y . The other two trees are prefix trees with the Merkle root $\text{Root}_{[0,1]}$ and $\text{Root}_{[0,3]}$ respectively. We denote by Index the index of leaves in prefix trees.

Fig. 2.2 depicts Merkle²'s data structure. For clarity, we use usernames as IDs, so users append values associated with their usernames in the form of ID-value pairs. Merkle²'s data structure consists of several top-level trees called chronological trees (sorted by time), and for every internal node in this tree, there exists a prefix tree (sorted by IDs). We now elaborate on each type of tree and on how they are nested.

The **Chronological Tree** stores users' ID-value pairs at its leaves from left to right in chronological order. For example, if Alice adds an ID-value pair into the system before Bob, her pair will appear on the left side of Bob's pair. Each ID-value pair is given a *position number*, which indicates the position of the leaf in the tree. For example, in Fig. 2.2, Alice has two values Val₀ and Val₃, which are assigned to leaves with position numbers 0 and 3, respectively. The ID-value pair can later be referenced by its position number as a leaf node. Each internal node of the chronological tree has a corresponding prefix tree, as shown in Fig. 2.2. The hash of each internal node in the chronological tree is the hash of the following triple:

- the hashes of its two children in the chronological tree, and
- the root hash of the prefix tree corresponding to this node.

This allows the root hash of a chronological tree to summarize the states of all the prefix trees within that chronological tree.

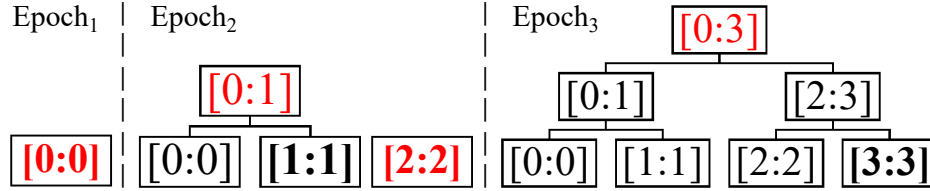


Figure 2.3: A forest transition starting from one leaf to four leaves. The red nodes indicate the Merkle roots in the digest. Leaves in bold indicate ID-value pairs added in each epoch.

The **Prefix Tree** stores users' ID-value pairs, arranged in lexicographic order by ID. The prefix tree associated with an internal node stores all ID-value pairs that appear inside the subtree rooted at that node. For example, in Fig. 2.2, the prefix tree corresponding to node [0:1] of the chronological tree stores all children in the subtree of [0:1]. Thus, the prefix tree of [0:1] (depicted on the left of Fig. 2.2) stores ID-value pairs Alice||Val₀ and Bob||Val₁. Meanwhile, the prefix tree on the right in Fig. 2.2 stores all of the ID-value pairs, because it is associated with the root node [0:3]. If a user appends multiple values, they will all be stored under the same leaf node in the prefix tree because they share the same ID. Since Alice appends two values (Val₀ and Val₃), both are stored in the same leaf node of the prefix tree. The hash of the internal node in a prefix tree is computed as in a typical Merkle tree, where the parent node hash is the hash of left and right child hashes concatenated together ($H(\text{leftChildHash}||\text{rightChildHash})$).

The reason why we chose the chronological tree as the outer layer is that auditors can check a succinct proof for all the appends between epochs. We elaborate in Section 2.5.1. In contrast, if we use the prefix tree as the outer layer, the size of the proof, which auditors need to check, might be linear to the number of appends between epochs. That is also the reason why CT [Lau14] chooses the chronological tree. Other systems such as CONIKS [MBBFF15] avoid this overhead by asking each ID owner to monitor every epoch.

The forest of chronological trees. Merkle²'s data structure consists of a forest of chronological trees. Each such tree is full; that is, no leaf is missing. This property is maintained so that as ID-value pairs get appended in epoch_{*i*+1}, we preserve the structure of the chronological trees from epoch_{*i*}. We ensure that in epoch_{*i*+1}, we only add parents to the existing trees of epoch_{*i*} or add separate trees altogether. This helps greatly with our extension proof in Section 2.5.1. Thus, leaves are added by extending Merkle²'s data structure to the right; as more leaves are added, new internal nodes (and therefore new roots) are created whenever we can construct a larger, full binary tree (as discussed in Section 2.4.2).

Fig. 2.3 illustrates the transition of Merkle²'s data structure after a set of appends. The internal node [0:1] is automatically created because leaves [0:0] and [1:1] can be stored under a full binary tree. Note that each internal node ([0:1], [2:3], [0:3]) contains the root hash of a prefix tree as shown in Fig. 2.2. This forest design ensures that the roots of an old forest remain as internal nodes in any later version of the forest, and their hashes will still be the same. For example, the hashes of [0:1] and [2:2] will not change between epoch₂ and epoch₃.

This construction enables us to design a succinct global extension proof and, as shown later,

enables users to check *only* the latest version. Since the latest version is an extension of all of the versions before it, no others need to be checked. The **digest** of Merkle²'s data structure contains: (1) the root hashes of the chronological trees in the forest; (2) the total number of leaves. The digest stores $O(\log(n))$ root hashes because n leaves can be stored under $O(\log(n))$ full binary trees, and there are $O(\log(n))$ chronological trees in the forest.

Storage complexity analysis. At first glance, the fact that each internal node of the chronological tree is associated with a prefix tree implies $O(n^2 \log(n))$ storage as there are $O(n)$ internal nodes in the chronological tree and each prefix tree may require $O(n \log(n))$ nodes. However, we observe that the size of most prefix trees is much smaller than $O(n \log(n))$ since each prefix tree only stores a small portion of ID-value pairs. Moreover, each prefix tree can be compressed to require only $O(p)$ nodes where p is the number of ID-value pairs in the prefix tree. We describe the compression algorithm in Section 2.11.2. Now, we consider the storage costs for all prefix trees at each height of the chronological forest. The number of ID-value pairs of all the prefix trees in the same height is $O(n)$; thus, there are $O(n)$ prefix tree nodes at each height of the chronological forest. And there are $O(\log(n))$ levels in total. Therefore, Merkle²'s data structure with n ID-value pairs requires only $O(n \log(n))$ storage.

2.4.2 Appending ID-value pairs

To append a new ID-value pair, Merkle²'s data structure first extends the forest by creating a new leaf node containing the ID-value pair. As mentioned above, the ID-value pair is assigned a position number according to the leaf position. Then, Merkle²'s data structure recursively merges the rightmost two chronological trees into one big chronological tree if they have the same size. This process repeats until it is no longer possible to merge the last two trees. For each root node created in the merging process, a corresponding prefix tree must be built by inserting all the ID-value pairs that occur under that root node.

For example, in epoch₃ of Fig. 2.3, the leaf [3:3] is added and the ID-value pair is assigned a position number 3. Nodes [2:3] and [0:3] are created to merge equally-sized chronological trees. The prefix tree of [2:3] is created by adding the ID-value pairs of [2:2] and [3:3]. And, the prefix tree of [0:3] is created by adding all ID-value pairs appended so far.

We now analyze the complexity of appending an ID-value pair. A single append results in the creation of only $O(\log(n))$ internal nodes, but Merkle²'s data structure has to build a new prefix tree for each new internal node. The bottleneck is building new prefix trees, since some prefix trees may have $O(n)$ leaves. However, every leaf node has at most $O(\log(n))$ ancestor nodes, which means each ID-value pair is inserted into $O(\log(n))$ prefix trees. The cost of inserting an ID-value pair into a prefix tree is $O(\log(n))$. Thus, the amortized cost of appending a new ID-value pair is $O(\log^2(n))$.

However, this solution is still impractical for a low-latency system. Suppose there are $2^{20}-1$ ID-value pairs in the system; the next append combines all roots into a singular chronological tree, under node [0: $2^{20}-1$]. Building the corresponding prefix tree, which contains at most 2^{20} leaves, incurs a $O(n)$ cost. Thus, although the amortized cost is $O(\log^2(n))$, some appends results in linear-time operations in the worst case.

To solve this problem, we introduce the pre-build strategy. We first fix the maximum number of ID-value pairs supported by Merkle²'s data structure. This number can be sufficiently large, such as 2^{32} . To append an ID-value pair, Merkle²'s data structure inserts it into all possible prefix trees, including those that do not exist yet but are supposed to be built in the future. In other words, we pre-build prefix trees that may be used in the future. For example, if Merkle²'s data structure supports at most 2^{32} ID-value pairs, we will add an ID-value pair to 32 prefix trees, which correspond to all the existing and future ancestor nodes of the leaf. The cost of each append is still $O(\log^2(n))$, where n now is the maximum number of ID-value pairs, but we avoid the high latency of the worst case operations. We provide more details in Section 2.11.1.

2.5 Monitoring protocol in Merkle²

In this section, we show how Merkle² performs monitoring efficiently. ID owners are responsible for monitoring their own ID-value pairs, while auditors keep track of digests published by the server in each epoch. The goal of our monitoring protocol is to:

- avoid the need to monitor in every single epoch;
- enable efficient monitoring in any given epoch.

Intuition. To address the former problem, we design an *efficient extension proof* that allows auditors to prove that every epoch is an extension of the previous one. This way, when an ID owner verifies a monitoring proof for epoch t , the ID owner is implicitly also verifying epochs $t - 1, \dots, 1$. Thus, the ID owner need not monitor each digest, only the latest one.

For the latter property, we carefully design a monitoring proof and co-design *signature chains* that enable an ID owner to verify that their values have not been tampered with.

2.5.1 Extension proofs

In prior transparency log [MBBFF15], each ID owner must monitor their ID-value pairs in every epoch as there is no guaranteed relationship between the server's state in different epochs. For example, at epoch _{t} , the server could switch to a corrupted state s' (having some corrupted value for some ID) for this epoch alone and then switch back to the correct state s in epoch _{$t+1$} . Thus, the server is able to equivocate, and ID owners will never detect it if they do not audit the equivocated epoch _{t} .

As a first step in solving this problem, Merkle² maintains the invariant that a system state in epoch _{t} is an extension of the state in epoch _{$t-1$} . In other words, every epoch is an extension of those before it, with the existing ID-value pairs in the same chronological order as before with all the new ID-value pairs occurring after the existing ones. Our extension proof, thus, is designed to be used by auditors to verify these requirements between system states in different epochs.

Each state of the system s_x can be summarized by the root hashes of the trees in its chronological forest. For example, in Fig. 2.3, the state of the system in epoch₂ can be represented by the hashes of nodes [0:1] and [2:2]. To prove that a state s_y is an extension of a state s_x , we must prove that all chronological roots of s_x are contained within those of s_y . By providing the minimum set of hashes necessary, it is possible to compute the root hashes of s_y from those of s_x , thereby proving the extension relationship between the two states.

For example, in Fig. 2.3, the extension proof between epoch₂ and epoch₃ contains the following hashes: the chronological tree node hash, $H_{[3:3]}$ and the prefix tree root hashes, $\text{Root}_{[2:3]}$, $\text{Root}_{[0:3]}$. Given the hashes $H_{[0:1]}$, $H_{[2:2]}$ from the old epoch's digest, the auditor can check if the hash $H_{[0:3]}$ in the new digest is computed correctly as follows:

1. compute $H_{[2:3]}$ using $H_{[2:2]}$, $H_{[3:3]}$, $\text{Root}_{[2:3]}$;
2. compute $H_{[0:3]}$ using $H_{[0:1]}$, $H_{[2:3]}$, $\text{Root}_{[0:3]}$;
3. check if $H_{[0:3]}$ matches the root hash in the new digest.

At the end of each epoch, auditors receive the new digest and the extension proof from the server. After verifying the extension proof, auditors gossip with each other to ensure that they share

a consistent view of the new digest. To reduce server load, auditors can also share extension proofs amongst themselves, since they are checking the same extension proofs.

Extension proofs prevent attackers from removing or modifying existing nodes in Merkle²'s data structure. Once an internal node is created, it will be a part of all future epochs, as they are extensions of the current state. Moreover, because each internal node contains a corresponding prefix tree, all prefix trees that exist in an earlier epoch will remain the *same* for all future epochs. Thus, once an ID owner monitors epoch_{*t*} and goes offline for *m* epochs, the ID owner only has to monitor the latest epoch (epoch_{*t+m*}), and implicitly verifies the system states in epoch_{*t+m*}, ..., epoch_{*t+1*}. In contrast, CONIKS requires owners to monitor system states in all epochs.

The Merkle² extension proof is similar to the consistency proof in CT [Lau14]; thus, the extension proof provides a similar property. There are a few key differences though. A difference is that the extension proof contains root hashes of prefix trees in the path due to the nested Merkle tree design. Another difference is that the consistency proof ensures that leaves are append-only and thus guarantees CT's security. Still, the extension proof alone does not suffice for Merkle²'s security goal because the attacker might compromise newly added prefix trees. Thus, we need monitoring proofs for ID owners to check prefix trees as explained in Section 2.5.2.

Complexity analysis. The size of the extension proof from a state s_x into a state s_y is dependent on the number of hashes required to construct the root hashes of s_y from those of s_x . Because the depth of any chronological root is $O(\log(n))$, there are $O(\log(n))$ ancestor node hashes required to prove the inclusion of the roots of s_x in those of s_y . Thus, the extension proof between two epochs contains $O(\log(n))$ hashes.

2.5.2 Monitoring proofs

Monitoring proofs enable ID owners to check contents of prefix trees in Merkle²'s data structure. For concreteness of exposition, consider that Bob wants to monitor his ID, denoted ID_{Bob}. A strawman design for the monitoring proof has Bob check every prefix tree for ID-value pairs matching ID_{Bob}. This way, for each prefix tree, the server provides a (non-)membership proof for ID_{Bob}, which can convince Bob that there are no unwanted changes. Unfortunately, the cost of this strawman is quasilinear in the number of ID-value pairs.

Instead, Merkle² requires ID owners to check only the prefix trees that are supposed to store their ID-value pairs. ID owners keep track of position numbers assigned to their ID-value pair; thus, they can infer which prefix trees to check. Given an ID-value pair $\langle \text{ID}, \text{val} \rangle$, we denote by v the leaf node that stores it in the chronological tree. The only prefix trees that will contain $\langle \text{ID}, \text{val} \rangle$ are those that correspond to the ancestors of v in the chronological tree. Thus, for each of these prefix trees, the server generates a membership proof for ID to ensure it exists within the prefix tree. Once checked, each membership proof will generate the prefix root hash it belongs to. Note that this mechanism by itself does not prevent a compromised server from adding ID-value pairs (the attacker can add values to the prefix trees that ID owners do not check); our signature chain co-design (Section 2.5.3) addresses this aspect.

The ID owner is not done yet, however, because she must still verify that the generated prefix root hashes are correct. The digest provided by the server only contains the root hashes of the

chronological forest. Thus, we must provide the minimum set of hashes so that the ID owner can reproduce the chronological root hash and compare it with the digest. Notice that the generated prefix tree root hashes each correspond to an ancestor of v in the chronological tree. If we provide the authentication path for v , the ID owner will have enough information to reproduce a digest root hash. Therefore, the monitoring proof for $\langle \text{ID}, \text{val} \rangle$ corresponding to leaf node v in the chronological tree consists of the following:

- membership proofs for prefix trees in ancestor nodes of v ;
- the authentication path in the chronological tree for v .

The verification process works as follows. Given $\langle \text{ID}, \text{val} \rangle$, its monitoring proof, and the digest, the ID owner begins by computing the root hashes of the prefix trees using the membership proofs. In conjunction with the authentication path, the verifier can reconstruct the hashes of every ancestor of node v until eventually reaching the root of the chronological tree. Then, the verifier can compare the computed root hash with the corresponding root hash in the digest. For example, in Fig. 2.2, Bob wants to monitor the ID-value pair $\text{Bob}||\text{Val}_1$. First, Bob computes the prefix root hashes $\text{Root}_{[0:1]}$, $\text{Root}_{[0:3]}$ using the membership proofs for the prefix trees corresponding to nodes $[0:1]$, $[0:3]$. Then, together with the authentication path $(H_{[0:0]}, H_{[2:3]})$, Bob can reconstruct and verify the chronological tree root hash $H_{[0:3]}$ with the corresponding hash in the digest.

The protocol described thus far allows ID owners to efficiently monitor Merkle² in a particular epoch. Because of the extension proofs in Section 2.5.1, the server cannot modify existing content of Merkle² in future epochs. Therefore, if a prefix tree has already been verified by an ID owner, it does not need to be verified again. When ID owners come online and request a monitoring proof, they can specify the latest epoch they have already monitored and only download membership proofs from prefix trees added since that should contain that ID-value pair. This way, a single prefix tree is only checked once by the same ID owner.

Complexity analysis. We now analyze the size of monitoring proofs. For each ID-value pair, we observe that the ID owner only needs to check $O(\log(n))$ prefix trees, because each leaf node in the chronological tree has at most $O(\log(n))$ ancestor nodes. For each of those prefix trees, the membership proof is of size $O(\log(n))$. Note that the ID owner can skip and cache prefix trees if they have been checked before. Overall, for each ID-value pair, the ID owner downloads monitoring proofs of total size $O(\log^2(n))$ throughout the system's life.

2.5.3 Signature chains design

The monitoring proof discussed in Section 2.5.2 only guarantees that the attacker cannot remove ID-value pairs from Merkle²'s data structure. It does not prevent the compromised server from adding ID-value pairs (e.g. to prefix trees the ID owners never check). For example, in Fig. 2.4, Alice cannot detect that the attacker inserted $\text{Alice}||\text{Val}'$ at position 25, because Alice does not have a value inside the chronological tree rooted at node $[24:27]$, so the monitoring proof will not include a membership proof for the prefix tree at node $[24:27]$.

To prevent attackers from adding corrupted ID-value pairs, Merkle² co-designs signature chains as follows. The ID owner attaches a verifying key to each ID-value pair in Merkle². And, on append, each new ID-value pair, its position, and the new verifying key are signed by the verifying key

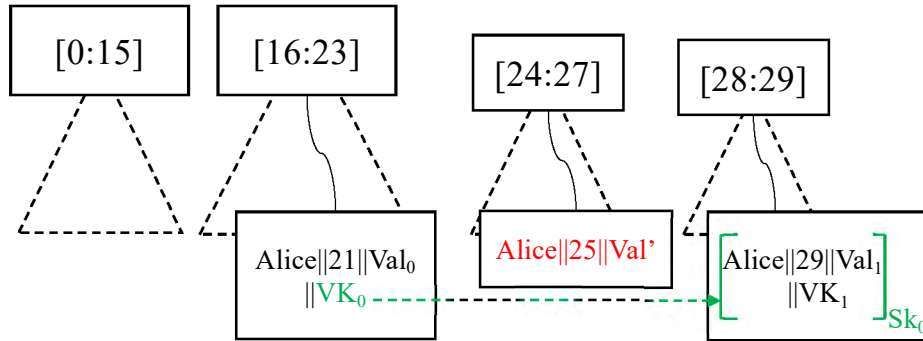


Figure 2.4: Alice’s values are Val_0 and Val_1 . The attacker may add values for the ID “Alice” in the chronological trees [24:27]. The signature chain prevents such attacks.

attached to the previous ID-value pair for the owner. Users can then verify the signature on an ID-value pair with the same verifying key. By verifying the signature chain, users can confirm that all the ID-value pairs are indeed appended by the ID owner. In the example of Fig. 2.4, although the attacker can still add $Alice||Val'$ without being caught by Alice, other users will not accept the corrupted pair because the attacker cannot produce a valid signature. Notice that the attacker may try to hide the end of the chain during lookup. The monitoring proof ensures that the attacker cannot hide the owner’s values without detection.

The protocol described thus far is still insecure because the first value is not signed. An attacker may insert a corrupt ID-value pair and try to convince users that it is the first value for that ID; thus, the attacker could circumvent the signature chain. We observe, however, that if the honest ID owner already has inserted values for that ID in Merkle², the attacker cannot convince other users that the falsified value is the first for that ID. This holds true because the monitoring proof ensures that the attacker cannot remove existing values without being detected. For example, in Fig. 2.4, the attacker cannot hide $Alice||21||Val_0$ and claim $Alice||25||Val'$ as the first pair for “Alice”; as shown in our lookup protocol in Section 2.6, other users looking up Alice’s values will verify the non-membership proof for the prefix tree in [16:23]. Meanwhile, Alice will also check the membership proof for [16:23] by verifying the monitoring proof. The server cannot provide both a membership and non-membership proof for a leaf node associated with the ID “Alice” for the same prefix tree.

First-value checking. The ID owner must ensure that it indeed appends the first ID-value pair for that ID in Merkle², otherwise others may have obtained ownership for that ID already. It is not feasible to check all the leaves appended before it. Instead, we can leverage non-membership proofs from prefix trees to prove that no value exists for that particular ID. For example, if the ID-value pair is added at position x , there exists a minimum set of chronological trees Ct_1, \dots, Ct_n that cover the previous $x - 1$ leaves (there are only $O(\log(x))$ chronological trees in this covering set). For the prefix tree corresponding to every chronological root of Ct_i , we can generate a non-membership proof for the ID. The non-membership proofs allow the ID owner to compute the root hashes of the prefix trees; in order to compute the root hashes of the chronological roots, Ct_1, \dots, Ct_n , the first-value proof also contains the minimum set of node hashes needed to do so. This way, the ID

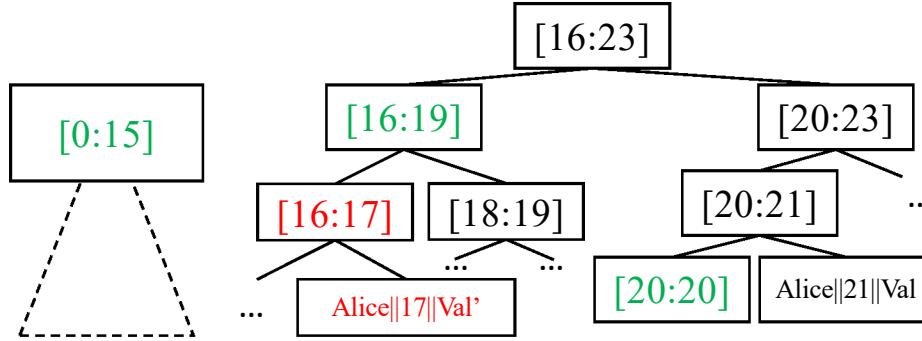


Figure 2.5: Alice appends the first value Val for her ID at position 21. Someone (either another honest user or the attacker) has appended Val' for ID “Alice” at position 17 already. Alice will verify the first-value proof, which contains non-membership proofs for ID “Alice” in the prefix trees at the green nodes.

owner can compute the hashes of the chronological roots and compare them against the digest.

For example, in Fig. 2.5, Alice verifies the non-membership proofs of the prefix trees for nodes [0:15], [16:19], and [20:20] to ensure that they do not contain values for “Alice.” If the non-membership proofs are not valid and there exists a value for “Alice,” Alice will know that she does not have ownership of that ID. If another honest user appends Alice||17||Val' before Alice appends Alice||21||val, then the attacker cannot hide Alice||17||Val' from Alice because when the honest user performs the monitoring protocol, it verifies the membership proof of the prefix tree of node [16:19]. Since the server cannot provide both a membership and non-membership proof for the same leaf node in the same prefix tree, the misbehavior will be detected. In addition to the non-membership proofs, the server must also provide the hashes necessary for Alice to compute the root hashes of the digest. For example, Alice can compute prefix root hash of node [16:19] using the non-membership proof provided, but she still needs $H_{[16:17]}$, $H_{[18:19]}$ to compute $H_{[16:19]}$. With the other hashes, Alice can finally compute the chronological root hash, $H_{[16:23]}$, to see if it matches the digest.

It is possible for a malicious server to add Alice||17||Val' to the chronological tree but remove it from the prefix tree at node [16:19]. However, once Alice appends the first value and successfully verifies the first-value proof, the attacker can no longer add Val' back to any prefix trees associated with its ancestor nodes without detection. New ancestor nodes added in the future will also be ancestors of Alice||21||val, so they will be checked in monitoring proofs by Alice. Thus, the attacker may be the owner of ID “Alice” before Alice joins the system; but, after Alice appends her first value, any value appended previously by the attacker will not be accepted by honest users, and the attacker no longer has the ability to append values for “Alice” without a valid signature.

Complexity analysis. If there are $O(n)$ ID-value pairs in the system, the covering set contains $O(\log(n))$ trees. Thus, there are $O(\log(n))$ total non-membership proofs to verify, each with a size of $O(\log(n))$, so the total cost of the first-value proof is $O(\log^2(n))$. ID owners only need to check the first-value proof once due to extension proofs.

2.6 Lookup protocol in Merkle²

In this section, we present the construction of lookup proofs in Merkle². We denote by Ct_1, \dots, Ct_n the chronological trees in the forest of Merkle²'s data structure, and Pt_1, \dots, Pt_n the prefix trees at their roots, which we also refer to as *root prefix trees*. The lookup proof must be able to convince users that the lookup result contains *all* the values for a given ID. By providing (non-)membership proofs for all the root prefix trees, the server can prove the (non-)membership of ID-value pairs for the given ID in all of Merkle². Further, the signature chain helps users verify the authenticity of the lookup result.

Based on these ideas, the lookup protocol for an ID works as follows. For each ID-value pair except the first one, the lookup proof contains a signature signed by the ID owner. For each prefix tree Pt_i , we generate a proof, π_i ; if there exists a value for ID in Pt_i , π_i is a membership proof. Otherwise, π_i is a non-membership proof. The lookup proof also contains the hashes LH_i, RH_i of the left and right children of root node Ct_i , which allows users to compute the root hashes and compare them to those in the digest. Finally, the lookup proof contains the signature chain and the following tuples: $(\langle \pi_1, LH_1, RH_1 \rangle, \dots, \langle \pi_n, LH_n, RH_n \rangle)$. Intuitively, the lookup proof captures which chronological roots contain values for ID and which do not.

In the example of Fig. 2.4, the lookup proof for Alice's values contains the signature chain and (non-)membership proofs for prefix trees [0:15], [16:23], [24:27], and [28:29]. The proofs for [16:23] and [28:29] are membership proofs of the leaf values associated with Alice. The proofs for [0:15] and [24:27] are used to prove non-membership of Alice's ID. Finally, the lookup proof also contains any hashes which are necessary to compute the root hashes; for example, $H_{[0:7]}$ and $H_{[8:15]}$ are used to compute the root hash $H_{[0:15]}$.

During lookup proof verification, the user possesses the following: 1) the lookup result, which contains all ID-value pairs for the target ID; 2) the latest digest from auditors that contains the root hashes of each chronological tree in the forest; 3) the lookup proof corresponding to the lookup result.

The user proceeds as follows:

1. verifies the signature chain using the verifying keys provided in each ID-value pair;
2. for each ID-value pair, finds which chronological tree Ct_i and corresponding prefix tree Pt_i it belongs to;
3. computes the root hash $Root_i$ for each Pt_i using π_i and the ID-value pairs in Pt_i based on the results from step 2;
4. computes the root hash for Ct_i using $Root_i$ and $\langle LH_i, RH_i \rangle$

If the signature chain is valid and the computed root hashes match those in the digest, the user can accept the lookup result.

Complexity analysis. The signature chain is of length $O(\ell)$ where ℓ is the number of values. The lookup proof contains (non-)membership proofs from $O(\log(n))$ prefix trees, and each (non-)membership proof is of size $O(\log(n))$. Therefore, the overall lookup proof is of size $O(\ell + \log^2(n))$.

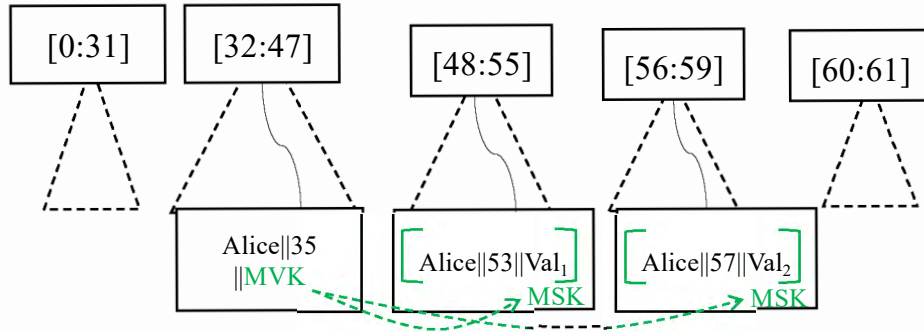


Figure 2.6: Alice appends the master verifying key as the first value. Following values are signed by the master signing key.

2.6.1 Lookup for the latest ID-value pair

We provide an optimized protocol to look up the *latest* (i.e., most recent) value for an ID. In many applications, users may want the latest value instead of all values. We make use of master keys, shown in Fig. 2.6, to replace signature chains since the size of the chain is linear to the number of values. ID owners generate a pair of master keys and append the master verifying key as their first ID-value pair in the log. ID owners must also ensure the master verifying key is the first value by verifying the first-value proof. All future pairs appended by the ID owner will be signed using the master signing key.

Instead of downloading all the ID-value pairs and signature chain for a given ID, users can download only the latest value and the master verifying key to verify the signature. Users also need (non-)membership proofs from prefix trees to ensure that the master verifying key is in fact the first value for the given ID and that the lookup result is in fact the latest one.

For example, in Fig. 2.6, users can verify that Alice’s master verifying key MVK is the first value for “Alice” by verifying the non-membership proof for the prefix tree [0:31] and the membership proof for the prefix tree [32:47]. Similarly, users can also verify that Alice||57||Val₂ is the latest value by verifying the membership proof for the prefix tree [56:59] and the non-membership proof for the prefix tree [60:61].

We assume ID owners will not change their master keys in the system. If required, Merkle² can permit ID owners to change their master keys as follows. ID owners can revoke master keys by appending and signing the new master key using the old master key. However, allowing changes to master keys will increase the lookup cost because the other users now also need to check the signature chain of the master keys. Instead, users can use existing techniques for backing up master keys on multiple devices securely just as they would do for their secret keys for end-to-end encrypted services, for example by using secret sharing [Sec; Sha79; WMZV16]. The cost of this modified lookup protocol is $O(\log^2(n))$ since there is no longer a signature chain cost. Users can also cache master keys to further reduce the cost of a lookup.

2.7 Applications of Merkle²

In this section, we discuss two applications of Merkle²: a ledger for web certificates and for a public key infrastructure.

2.7.1 Transparency log for web certificates

The security issues, where CAs have been compromised and issued certificates incorrectly [Lau; Adk; Wha; Man; NB13], prompted the design of web certificate management systems using transparency logs, where the owner of a certificate can verify the integrity of their own certificate and hold CAs accountable for corrupted certificates [LKL13; Chr; Rya14; Lau14; CSPLM15].

We now describe how to use Merkle² for certificate management in place of existing systems [LKL13; Lau14]. The log server runs Merkle²'s server to manage certificates for each domain name. The ID is the domain name and the values are the web certificates for that domain. There may be more than one certificate for the same domain. Instead of storing certificates as different domain-certificate pairs, we bundle multiple certificates together as a single value, and each append will be the hash of all the certificates for the domain name. Merkle² also supports revocation efficiently by allowing the domain owner to append the hash of all the unrevoked certificates for the same domain name. If a certificate is not in the latest append for this domain name, it is not valid (e.g., it was revoked). In the end, web browsers can simply retrieve the latest value for the domain to check whether a certificate is valid.

Benefits of our system. We compare Merkle² to existing proposals. Deployed CT systems [LKL13; Lau14] do not support revocation. Enhanced certificate transparency (ECT) [Rya14] aims to solve this problem. ECT also uses prefix trees (they use a similar design called lexicographic trees) and chronological trees, but ECT keeps these trees *separate*; thus, ECT requires auditors to verify the relation between these two trees by scanning linearly through all entries in the log. Hence, auditors require $O(n)$ time and space to perform their monitoring. In contrast, Merkle² provides a way to *nest* the two trees to reap their benefits *simultaneously* through our “multi-dimensional” design, which reduces the cost of auditors to only $O(\log(n))$. We give a concrete performance comparison in Section 2.8.

2.7.2 Transparency log for public keys

Merkle² can be used for a public-key infrastructure as an alternative to CONIKS [MBBFF15] or KT [HB17]. We use end-to-end encrypted email systems [ema] as a real-world example. In this application, an ID in Merkle² corresponds to a user's email, e.g. `alice@org.com`, and the value corresponds to the public key of the user, e.g., PK_{Alice} . To join the system, Alice appends the first public key for her email address `alice@org.com`. To revoke a public key, Alice appends a new public key for her email address. If Bob wants to send an encrypted email to Alice, he looks up the latest public key for `alice@org.com` and uses it to encrypt and send the email. For company communications, the organization can monitor all the email-PK pairs. For a personal account, the client running on the user's devices can monitor the transparency log regularly.

Benefits of our system. We compare Merkle² to CONIKS [MBBFF15], a state-of-the-art key transparency system. Monitoring in CONIKS is inefficient as key owners must check each digest published by the server. In contrast, each key owner in Merkle² only monitors $O(\log^2(n))$ data throughout the system's life, where n is the number of keys in the log. As we will show in Section 2.8, Merkle² can support short enough epoch periods (such as 1 second) to be considered low-latency [Gooc; Good].

Privacy concerns. CONIKS [MBBFF15] shows that verifiable random functions (VRFs) [MRV99] can reduce the leakage of IDs to malicious users. The same technique can be applied to Merkle². Instead of using an ID directly, users compute indices using the output of VRFs on the ID; the server also includes the information needed to verify the VRF result in the reply.

2.8 Implementation and evaluation

We implement a prototype of Merkle² in Go. It consists of four parts, as in Fig. 2.1: the server (≈ 800 LoC), auditor (≈ 200 LoC), client library (≈ 450 LoC), and verification daemon (≈ 600 LoC), which all depend on a set of core Merkle² data structure libraries (≈ 2400 LoC). The Merkle² library is available at <https://github.com/ucbrise/MerkleSquare>.

Our server implementation backs up ID-value pairs in persistent storage in case the server fails using LevelDB [GD], which has been used in previous transparency log systems [MBBFF15; HB17]. To provide a 128-bit security level, we used SHA-3 [Dwo15] as the hash function and Ed25519 signatures [BDLSY12] (this is the only public key operation in the system). We did not implement VRFs [MRV99] since privacy is not the focus of this paper. We limit the chronological tree height in Merkle² to 31 to support the pre-build strategy, which means it can store up to 2^{31} ID-value pairs and each append will be added to 31 prefix trees.

Concurrency control. Merkle²'s server can serve requests in parallel, relying on concurrency control of Merkle²'s data structure and LevelDB. Merkle²'s data structure prohibits concurrent appends since two appends may affect the same prefix tree. For each append, the server sends a position number to the user, and the user should reply with the signature. If the user withholds or does not reply with the signature within a short time bound, the server rejects the append. Lookups and monitoring can be concurrent with appends because required hashes have been computed in the past epochs.

2.8.1 Setup

Experiment setups. We ran our experiments on Amazon EC2 instances; the microbenchmarks and system server were run on a r5.2xlarge instance. The auditor and client services were run on a r5a.xlarge and c4.8xlarge instance, respectively.

Baselines. We chose three state-of-the-art transparency logs to compare with: CONIKS [MBBFF15], AAD [TBPPTD19], and ECT [Rya14]. We compare Merkle²'s complexity with these baselines in Table 2.1.

We compare Merkle² with CONIKS via both microbenchmarks and end-to-end system performance. The original CONIKS implementation [Con] is quite incomplete; for example, it does not support monitoring or persistent storage. Moreover, since CONIKS is not designed for short epochs, it copies and reconstructs the entire Merkle tree in each epoch; this incurs a large time and space cost, which would unfairly disadvantage CONIKS in comparison to Merkle². To produce a fair comparison, we enhanced the CONIKS design to use persistent data structures [DSST86], avoiding the overhead of copying the entire tree. We also implemented the monitoring functionality from its paper, and wrapped the CONIKS data structure into a server system. We disabled VRFs [MRV99] in the CONIKS implementation since we do not focus on privacy. Similarly to Merkle², the modified CONIKS implementation can process lookups and monitoring during appends, but does not allow concurrent appends.

AAD is an asymptotically efficient transparency log built on top of bilinear accumulators [Ngu05], but its constants are large. We compare with AAD's microbenchmarks results from their paper and

repository [TBPPTD19; Aad] because our setup cannot support running experiments for AAD. For example, it takes more than 20 hours to generate the public parameters necessary, and as shown in [TBPPTD19], the experiments were run on a r4.16xlarge instance, which is much more powerful than our machines.

We also compare Merkle² with ECT, for the use case of transparent web certificates. Since there is no ECT implementation available, we use the numbers provided in their paper and other online statistics for the comparison.

2.8.2 Microbenchmarks

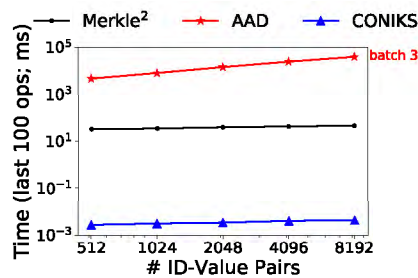


Figure 2.7: Append time.

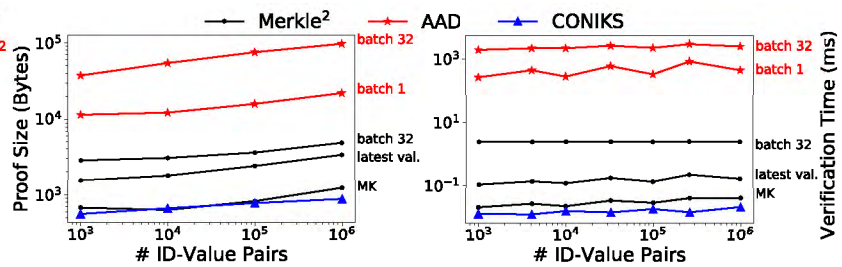


Figure 2.8: Lookup proof size and verification time.

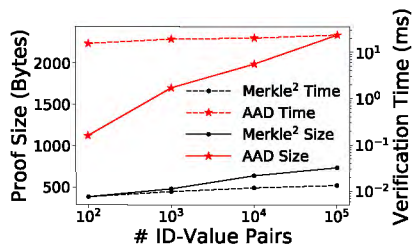


Figure 2.9: Monitoring cost for auditors in AAD and Merkle².

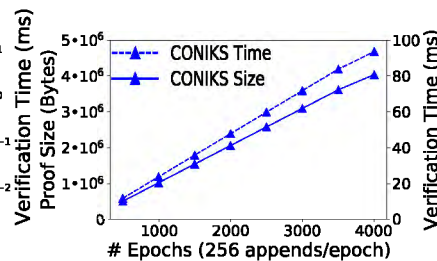


Figure 2.10: Monitoring cost for ID owners in CONIKS.

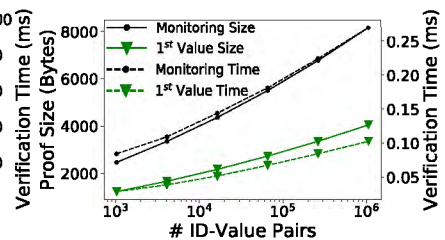


Figure 2.11: Monitoring cost for ID owners in Merkle².

We compare Merkle²'s core protocol to AAD [TBPPTD19] and CONIKS [MBBFF15] via microbenchmarks for individual operations.

Append time. The append time comparison (depicted in Fig. 2.7) measures the total time taken for the last 100 appends for a given number of ID-value pairs. AAD only provides benchmarks up to $2^{13} = 8192$ ID-value pairs because of how long larger scale appends would take [TBPPTD19]. AAD supports a batching mechanism to speed up append operations; in our graph, we include only the result for the 32 ID-value pair batch as the time for a single append is too high. In spite of the batching, AAD is still significantly slower than Merkle² and CONIKS because it uses bilinear

accumulators. CONIKS, on the other hand, is faster than Merkle²; a single append in CONIKS inserts only an ID-value pair into a single prefix tree. In Merkle², we measured the append cost with the pre-build strategy. Because we set the maximum chronological tree height to 31, each append is added to 31 prefix trees. If there are 2^{20} ID-value pairs in the system, it takes 3 ms for the last 100 appends in CONIKS, while in Merkle² it takes 151 ms.

Monitoring cost. In Fig. 2.9, we contrast the monitoring costs of auditors in AAD and in Merkle². We compare proof size and verification time for extension proofs between system states containing 10^i and 10^{i+1} ID-value pairs. Clearly, the monitoring costs of auditors in AAD is higher than those of auditors in Merkle² for both proof size and verification time. Because system states in different epochs in CONIKS share no defined relationship, CONIKS does not have extension proofs, so it was excluded from this experiment.

Now, we compare the monitoring costs of ID owners in CONIKS with those of ID owners in Merkle². In Fig. 2.10, we add 2^{20} ID-value pairs into a CONIKS system. Then, we vary the number of epochs the ID owner must monitor because CONIKS requires each ID owner to monitor every epoch. The monitoring costs of CONIKS grow linearly with the number of epochs. In contrast, Merkle²'s monitoring proof depends only on the number of existing ID-value pairs; it is independent of the number of epochs since ID owners can simply monitor the latest epoch. In Merkle², an ID owner must check the first-value proof when they join the system and then regularly monitor their ID-value pairs; the costs are depicted in Fig. 2.11. As discussed in Section 2.5.2, ID owners can cache monitoring proofs and need to verify the first-value proof only once; thus, the monitoring cost of ID owners in Merkle² is significantly lower than those of ID owners in CONIKS.

Lookup cost. In Fig. 2.8, we compare the average lookup proof size and verification time for all three systems. The lookup cost in Merkle² and AAD may increase when there are more values for the target ID, so we measure the results for both single ID-value pairs and batches of 32. Lookup costs fluctuate as performance depends on the underlying structure of the system. In Merkle², for example, $2^{12} - 1$ ID-value pairs result in more root prefix trees than 10^4 ID-value pairs. What first appears to be a discrepancy is actually an odd feature of the system. The batched lookup proof, however, is not affected because the cost is dominated by the signature chain.

In Merkle², the master key and latest value lookups are more efficient than the batched lookup because they do not require signature chains. Note that the master key lookup cost of Merkle² is close to that of CONIKS. This occurs due to the forest design in Merkle², where the root prefix trees at the beginning of the forest span more ID-values than the others. Because master keys require non-membership proofs beginning from the left of the forest, there are fewer (non-)membership proofs in the master key proof than in the latest value proof. In most cases, the master key is covered by the largest chronological tree. Thus it contains only a single membership proof and has a cost close to that of CONIKS.

Memory usage. Fig. 2.12 compares the memory cost of Merkle² and CONIKS. We fix the number of appends per epoch to be 256. We also limit the chronological tree height in Merkle² to 31 to support the pre-build strategy.

For 2^{20} ID-value pairs, Merkle² consumed about 22 GiB of RAM, and CONIKS consumed about 6.3 GiB of RAM. We obtain the memory usage of AAD from its paper [TBPPTD19], and it consumed 263 GiB of RAM. AAD additionally required 64 GiB of RAM for public parameters.

Operation		Merkle ²	CONIKS
Append		4.39	3.02
Lookup	Master key	1.62	1.28
	Latest value	2.31	
Owner Monitor	1 epoch	2.33	1.29
	10 epochs		2.32
	100 epochs		10.62
	1000 epochs		88.03
Auditor Monitor		0.25	0.22

Table 2.2: Latency (in ms).

Operation		Merkle ²	CONIKS
Append		42B	42B
Lookup	Master key	3.8KB	1.6KB
	Latest value	9.8KB	
Owner Monitor	1 epoch	22.9KB	2.1KB
	10 epochs		21KB
	100 epochs		209.6KB
	1000 epochs		2.1MB
Auditor Monitor		654B	370B

Table 2.3: Message size of server’s responses.

The original CONIKS implementation copies the prefix tree for each epoch and results in $O(E \cdot n)$ nodes where E is the number of epochs, which is prohibitively large for shorter epochs. Instead, we improve CONIKS by leveraging persistent data structures [DSST86] to avoid copying prefix trees in each epoch. Each insertion in the persistent prefix tree creates $O(\log(n))$ nodes; thus, CONIKS has to store $O(n \log(n))$ nodes in total. Merkle²’s asymptotic storage cost is the same as CONIKS, but the memory usage is higher due to the larger constants. We discuss how to optimize Merkle²’s storage in Section 2.8.5.

2.8.3 End-to-end system evaluation

In this section, we evaluate Merkle²’s system-level performance with that of CONIKS. Note that we do not use VRFs [MRV99] in either Merkle² or CONIKS as we are interested in the main system cost. We also limit the chronological tree height in Merkle² to 31. In the experiment, we insert 10^6 ID-value pairs into each system before running the benchmarks.

End-to-end performance. We analyze the end-to-end performance from the client’s perspective, which includes the proof verification and communication with the server and auditor. Table 2.2

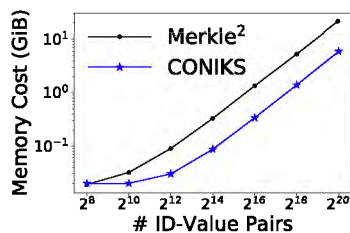


Figure 2.12: Memory cost.

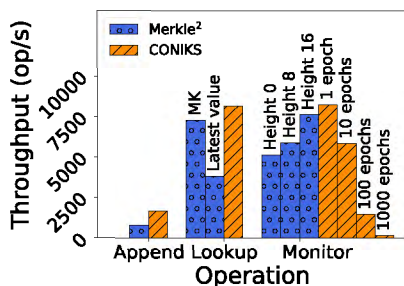


Figure 2.13: Server throughput.

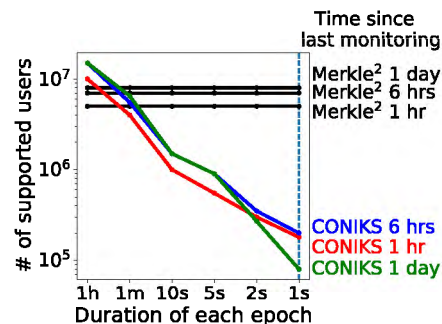


Figure 2.14: Users supported by one server.

shows the average latency and Table 2.3 shows the message size of server responses for 10^3 operations. For Merkle², we measure only the latest-value lookup protocol as it is more efficient and useful in real-world applications. And, we separate the cost of looking up the master verifying key and latest ID-value pair, since users can store and reuse master keys as discussed in Section 2.6.1. The result shows that appends and lookups in Merkle² are more expensive. The cost for the master key lookup is cheaper than that of the latest-value lookup; this occurs because the proof for the latter is smaller due to the forest design in Merkle².

To measure the cost for ID owners to monitor, we vary the number of epochs the user is offline since last monitoring. As mentioned, ID owners in CONIKS must check every digest published, which makes the cost grow significantly when a user is offline for some time. In contrast, Merkle² does not see the same increase because ID owners can check only the latest digest. Based on results in Table 2.2 and Table 2.3 for CONIKS, we can estimate the cost of an ID owner who wishes to verify all digests in a given month. Suppose the epoch is 1 second, as is desired by Google KT [Gooc]; in a month, there will be $30 \cdot 24 \cdot 60 \cdot 60 = 2592000$ epochs. Then, the ID owner needs to download $\frac{2592000}{1000} \cdot 2.1\text{MB} \approx 5.4\text{GB}$ of data and spend $\frac{2592000}{1000} \cdot 88\text{ms} \approx 3.8\text{mins}$ to verify it. This cost is problematic for the server who has to incur this cost for every data owner. In contrast, ID owners in Merkle² only download 22.9KB of data, which is independent of the number of epochs.

We measure the cost for auditors to check the new digest. Auditors in Merkle² check the extension proof, whereas auditors in CONIKS fetch only the digest from the server. The result shows that if there are 2^8 appends between epochs, auditors in Merkle² are as lightweight as those in CONIKS. Note that auditors in Merkle² can gossip the extension proof with each other to further reduce the bandwidth of the server.

Throughput. Next, we measure the throughput of frequent operations in Merkle² and CONIKS, depicted in Fig. 2.13. In the experiments, we randomly choose an ID-value pair for lookup and monitoring. CONIKS can support more append and lookup operations because its append and lookup are more efficient than those of Merkle². The throughput of the master key lookup is much higher than that of the latest-value lookup because the master key lookup proof is smaller than the latest value lookup proof, as shown in Table 2.3.

For the monitoring throughput benchmark, we fix every ID owner’s monitoring frequency; for

CONIKS, there is a fixed number of digests the ID owners must monitor. The results clearly show that the server’s performance decreases significantly when ID owners monitor larger numbers of epochs. In Merkle², the monitoring cost is independent of the number of epochs, and ID owners can cache and skip membership proofs of prefix trees that are checked in the past. To illustrate the saving provided by this caching mechanism, let i be the height of the highest node associated with the prefix tree that is in ID owners’ cache. ID owners do not need to download membership proofs for prefix trees at nodes below height i . In Fig. 2.13, we vary the height i to show the throughput. “Height 0” shows the worst case result, as it means ID owners need to check all the prefix trees. The results show that the throughput increases when ID owners cache more monitoring proofs.

2.8.4 Performance in applications

In this section, we compare Merkle²’s performance in the applications described in Section 2.7.

Web certificate management. To compare with ECT, we estimate the cost of ECT based on numbers in their paper and with the help of online statistics. Recent certificate statistics [Li+19] show that about 5,002,599 certificates are appended every day since June 2018. Additionally, auditors in ECT require 2 KB of data for each append [Rya14]. Thus, ECT auditors have to download $5002599 * 2\text{KB} \approx 9.5\text{GB}$ of data per day for monitoring. In contrast, Merkle² auditors require only $\log(10^9) * 32\text{B} * 2 \approx 1.9\text{KB}$ of data for monitoring.

Public keys management. As shown in previous sections, Merkle² supports efficient monitoring but sacrifices some performance for append and lookup operations. To understand the benefits of Merkle², we must run our benchmarks under workloads matching the target application. Thus, we compare Merkle² and CONIKS in the following scenario.

We consider the real-world application of encrypted emails using available statistics; in particular, there are 200 appends per second for 1 billion users [Gooc; Good], and each user sends 42 emails per day [Rad]. Users may also cache public keys for emails sent within a short period of time, such as 1 hour. Using the Enron email dataset [Coh], we find that users need to perform a key lookup for about 62% of emails. In summary, each user may send $42 * 62\% = 26$ lookup requests per day. We control the number of monitoring requests by adjusting the average monitoring frequency. Based on these results, we generate the workload for different numbers of users under different epoch intervals and monitoring frequencies. To keep the experiment time reasonable, we add only 10^6 ID-value pairs into the system before running them.

Fig. 2.14 depicts the number of supported users by a single server machine for both systems. The result shows that the epoch interval does not affect the performance of Merkle². When users monitor the server more frequently, Merkle²’s performance decreases because the server has to serve more requests. CONIKS’s performance is significantly worse than Merkle² when the epoch interval is short because of its expensive monitoring protocol. Users may increase their monitoring frequency to avoid larger computations as a result of being offline for many epochs. However, this increases the number of monitoring requests that the server must handle, which may decrease its performance. On the other hand, when the epoch interval becomes 1 hour, CONIKS starts to outperform Merkle² because its appends and lookups are more efficient. We find that the monitoring caching mechanism does not improve the performance of Merkle² when the monitoring interval is long (more than

1 hour) because lookups dominate the workload in this application. The caching mechanism is more helpful in applications with more frequent monitoring. In conclusion, Merkle² significantly outperforms CONIKS when the epoch interval is small (for example, one second), which is desirable for a low-latency key transparency system.

2.8.5 Limitations and future work

These experiments suggest Merkle² cannot outperform CONIKS when the epoch interval is long because appends and lookups become the bottleneck. Merkle² leaves as future work improving its append and lookup efficiency. For example, we can use one server to pre-build large prefix trees that are supposed to be used in the future; thus, another server can cache smaller prefix trees and serve users' requests more efficiently. Also, we can batch lookup proofs for different users and leverage proxies to save the bandwidth of the server.

Given that data is replicated in Merkle²'s data structure, memory usage is also a concern. We notice that prefix trees associated with the right children in chronological trees are not needed; thus, the server can save half of the space by skipping these prefix trees. We leave these optimizations to future work.

2.9 Related works

Transparency logs. We have already compared extensively with CT [LKL13; Lau14], ECT [Rya14], CONIKS [MBBFF15], and AAD [TBPPTD19].

Recently, there has been extensive research into improving the performance of transparency logs. One school of thought [And+19; Rya14; YCR16], attempts to use prefix trees and chronological trees in parallel for efficient lookup and state monitoring, respectively. Unfortunately, as in ECT, auditors and owners must still verify all operations in the chronological tree to verify that the prefix tree is built correctly. WAVE [And+19] relies on strong auditors to monitor on behalf of users, which incurs a large burden on the auditors. SEEMless [CDGM19] is the first for proposing persistent data structures to optimize CONIKS. However, it relies on strong auditors to monitor the append-only property of the Persistent Patricia Trie, which still incurs a super-linear cost on the auditors. ECT and DTKI [YCR16] also rely on users to collectively verify server states; collective verification, however, assumes enough honest users in the system, which limits its use in real-world applications. Google KT [Gooc] recently proposed a new design that requires users performing a lookup on a value to only perform verification of the same digest as the value owner. To ensure that the value owner and user both verify the same digest, KT uses a meet-in-the-middle algorithm. As a result, the monitoring cost of the owner becomes $O(\log(E) \log(n))$, where E is the number of epochs. However, the cost of a lookup also increases to $O(\log(E) \log(n))$. We do not perform any comparison to KT in our evaluations as it is still in its early stages, and there is no protocol detail or implementation available for benchmarking. AKI [KHPJG13] has the server maintain prefix trees, as in CONIKS; it distributes the monitoring workload to auditors, ID owners, users, and other third parties. Unfortunately, AKI operates on the assumption that no parties collude. ARPKI [BCKPSS14] and PoliCert [SMP14] extend the security of AKI by protecting against attackers controlling $n-1$ out of n parties.

Another approach is to use recursive SNARKs [BSCTV17; BCCT13; COS20a; CHMMVW20] or cryptographic accumulators [Ngu05; BBF19]. However, these solutions are too expensive to be practical.

Several gossip protocols [Syt+16; CSPLM15; DPVHJK19; TD17] are designed to ensure a consistent view of digests among users and auditors. Merkle² can use these to share digests and extension proofs. Software transparency logs [AM18; FDPFSS14; HC17; Nik+17; Syt+16; TD17] are designed for securing software updates. Merkle² can be used to improve the performance of these systems.

Previous works also formalize the security guarantees of CT [CM16; DGHS16] and those of general transparency logs [CM16; TBPPTD19]. The security guarantees of Merkle² can be analyzed under the same model; we leave this for future work.

Authenticated data structures. Authenticated data structures [Mer87; TBPPTD19] are at the core of transparency logs. Some works [DPP16; KKKP] focus on improving the performance of Merkle tree implementations, which can also be applied to Merkle². Cryptographic accumulators [Ngu05; BBF19; BBF19] can also be used for building authenticated data structures; however, these cryptographic primitives have a large overhead.

Blockchains and consensus protocols. Decentralized ledgers can be built on top of blockchains using consensus protocols [Nak19; Woo14; CL99; YMRGA19; PS18; Lok+19], which have seen

widespread adoption in cryptocurrencies [Bita; Eth; Zcaa; Lok+19]. Merkle² is more efficient and lightweight than blockchain-based systems because it is hosted centrally; this way, there is no need for expensive consensus protocols or data replication. At the same time, Merkle² loses the availability guarantee of blockchain systems, since a malicious server can deny service.

Another approach is to use blockchains to provide efficient auditing mechanisms for transparency logs [Bon16; TD17; ANSF16; MR17; WLCWZJ20]. However, the performance of this approach is limited by the underlying blockchain protocol. Some works [MHWK16; ZEEJVR17; Che+19] leverage trusted hardware to improve the performance of blockchain systems. However, existing hardware, like Intel SGX, are susceptible to side-channel attacks [VB+18].

File sharing with an untrusted server. Many systems [LKMS04; KFPC16; HKP20; SCCKMS10] allow users to share files on untrusted storage. The focus of these works is to provide a file sharing functionality instead of an immutable append-only log. SUNDR [LKMS04] and Venus [SCCKMS10] achieve weaker consistency guarantee than Merkle². Verena [KFPC16] relies on two “non-colluding” servers. Ghostor [HKP20] relies on either a blockchain or transparency log, so Merkle² can be used as a foundation for Ghostor.

2.10 Security of Merkle²

Before we can prove the security of Merkle², we must first consider the properties of the data structure. Our first step is thus to define the security properties of the prefix trees and chronological trees that make up Merkle²'s data structure in Section 2.10.1 and Section 2.10.2. Armed with these properties, we prove Guarantee 1 in Section 2.10.3.

2.10.1 Prefix trees

In this section, we briefly introduce prefix trees [MBBFF15] used in Merkle². The prefix tree is a binary trie [DLB59] built over a set of ID-value pairs S . Each node in the prefix tree is labeled with an index that is defined recursively: the root node is labeled with an empty index string; given a node with index p , its left and right children are labeled with indices $p + '0'$ and $p + '1'$, respectively. The leaf node with prefix p stores the hash of all the ID-value pairs $\langle \text{ID}, \text{val} \rangle$ in S for which ID is equal to the index p . The internal node hash will be the hash of its left and right children hashes. We use the term *empty node* to denote any node with an index that is not a prefix of any ID in S . The empty node has a special hash to avoid a collision in the same location [MBBFF15]. We define the authentication path in prefix trees below.

Definition 2.10.1 (Authentication paths in prefix trees). *Given a prefix tree \mathcal{T} and a node v with index \mathcal{I} , the authentication path of v contains node hashes that are on the co-path of v .*

The server proves membership by presenting the authentication path for the leaf node with index matching ID. If the user successfully recomputes the root hash, it ensures that the server returns all the values for ID in S . The server can prove that there is no value for ID in S by presenting the authentication path for an empty node, whose index is the prefix of ID.

Given a prefix tree root hash Root , an ID, the node hash H (H can be the hash of the empty node with index that is the prefix of ID), and an authentication path Path , we denote by $\text{PrefixTree.Check}(\text{Root}, \text{ID}, H, \text{Path})$ the process to check the authentication path.

$\text{PrefixTree.Check}(\text{Root}, \text{ID}, H, \text{Path}) = 1$ means that users successfully recompute the root hash Root . We define (non-)membership security of prefix trees below.

Guarantee 2.10.2 ((Non-)Membership security of prefix trees). *Assume that the hash function used in the prefix tree is a collision-resistant hash function [Gol07]. For all polynomial-time adversaries \mathcal{A} there exists a negligible function $\nu(\cdot)$ such that:*

$$\Pr \left[\begin{array}{l} (\text{Root}, \text{ID}, H_1, \text{Path}_1, H_2, \text{Path}_2) \leftarrow \mathcal{A}(1^\lambda) : \\ H_1 \neq H_2 \\ \wedge \\ \text{PrefixTree.Check}(\text{Root}, \text{ID}, H_1, \text{Path}_1) = 1 \\ \wedge \\ \text{PrefixTree.Check}(\text{Root}, \text{ID}, H_2, \text{Path}_2) = 1 \end{array} \right] = \nu(\lambda) .$$

This is a standard security property of Merkle prefix trees [MBBFF15]; thus, we do not prove it here due to space constraints.

2.10.2 Chronological trees

In this section, we introduce chronological trees in Merkle². Our chronological trees are different from those in CT [Lau14] since the internal nodes contain root hashes of prefix trees. A chronological tree is a full binary tree, whose leaves are created chronologically. A chronological forest is a set of chronological trees as discussed in Section 2.4.1. We define the index for a node v as a tuple $[L:R]$, where the subtree rooted at v includes all leaves with position numbers L to R . If v is a leaf, L and R will be the position number of v . The node index enables us to specify the location of node in the chronological forest. We define the authentication path below.

Definition 2.10.3 (Authentication path in chronological trees). *Given a chronological forest \mathcal{F} , a chronological tree \mathcal{T} in \mathcal{F} , and a node v with index $[L : R]$ in \mathcal{T} , the authentication path of v consists of the following hashes:*

- *prefix tree root hashes in all the ancestor nodes of v in \mathcal{T} ;*
- *the node hashes that are on the co-path of v in \mathcal{T} .*

The server uses the authentication path to prove the membership of both leaf and internal nodes. That is, given a prefix tree root hash, the server can prove that it is stored in a node in the chronological forest. For example, the server first proves a node v has the node hash H_v by presenting the authentication path of v . Then, the server can provide the node hashes $H_{\text{Left}}, H_{\text{Right}}$ of left and right children of v , and the user checks by recomputing H_v using the prefix tree root hash.

Given a digest \mathcal{D} of a chronological forest (which contains hashes of all the root nodes in the forest and the forest size), a node index \mathcal{I} , the node hash H , and an authentication path Path , we denote by $\text{ChronForest.Check}(\mathcal{D}, \mathcal{I}, H, \text{Path})$ the process to check the authentication path. The user first finds the chronological tree root hash Root in \mathcal{D} based on the forest size and the index \mathcal{I} . If $\text{ChronForest.Check}(\mathcal{D}, \mathcal{I}, H, \text{Path}) = 1$, then the user successfully recomputed the root hash Root and ensured that the node with index \mathcal{I} has node hash H . If the node with index \mathcal{I} is the root node, then Path is empty, and the user will compare H with Root directly. We define membership security of chronological trees below.

Guarantee 2.10.4 (Membership security of chronological trees). *Assume that the hash function used in chronological trees is a collision-resistant hash function [Gol07]. For all polynomial-time adversaries \mathcal{A} there exists a negligible function $\nu(\cdot)$ such that:*

$$\Pr \left[\begin{array}{l} (\mathcal{D}, \mathcal{I}, H_1, \text{Path}_1, H_2, \text{Path}_2) \leftarrow \mathcal{A}(1^\lambda) : \\ H_1 \neq H_2 \\ \wedge \\ \text{ChronForest.Check}(\mathcal{D}, \mathcal{I}, H_1, \text{Path}_1) = 1 \\ \wedge \\ \text{ChronForest.Check}(\mathcal{D}, \mathcal{I}, H_2, \text{Path}_2) = 1 \end{array} \right] = \nu(\lambda) .$$

One can prove the above security guarantee via a straightforward extension to the membership security proof in CT [Lau14]. We do not include it here due to space constraints.

The chronological tree in Merkle² can also provide the extension proof for updates. At the end of each epoch, the server provides the extension proof to auditors to show that the update does not modify existing node hashes. Given two digests $\mathcal{D}_1, \mathcal{D}_2$ and an extension proof π , we denote by $\text{Extension.Check}(\mathcal{D}_1, \mathcal{D}_2, \pi)$ the process to check the extension proof. Auditors first check the forest size in \mathcal{D}_2 is greater than the size in \mathcal{D}_1 , and then try to recompute root hashes in \mathcal{D}_2 using \mathcal{D}_1 and π . If $\text{Extension.Check}(\mathcal{D}_1, \mathcal{D}_2, \pi) = 1$, then auditors will accept the update to the chronological forest. We define the append-only property of chronological trees below.

Guarantee 2.10.5 (Append-only security of chronological trees). *Assume that the hash function used in chronological trees is a collision-resistant hash function [Gol07]. For all polynomial-time adversaries \mathcal{A} there exists a polynomial-time extractor \mathcal{E} and a negligible function $\nu(\cdot)$ such that:*

$$\Pr \left[\begin{array}{l} (\mathcal{D}_1, \mathcal{D}_2, \pi, \mathcal{I}, H, \text{Path}_1) \leftarrow \mathcal{A}(1^\lambda); \\ \text{Path}_2 \leftarrow \mathcal{E}(1^\lambda, \mathcal{D}_1, \mathcal{D}_2, \pi, \mathcal{I}, H, \text{Path}_1) : \\ \text{Extension.Check}(\mathcal{D}_1, \mathcal{D}_2, \pi) = 1 \\ \quad \wedge \\ \text{ChronForest.Check}(\mathcal{D}_1, \mathcal{I}, H, \text{Path}_1) = 1 \\ \quad \downarrow \\ \text{ChronForest.Check}(\mathcal{D}_2, \mathcal{I}, H, \text{Path}_2) = 1 \end{array} \right] = 1 - \nu(\lambda) .$$

Proof. We show how to construct the authentication path Path_2 by using Path_1 and π . Root_1 and Root_2 each denotes the root hash of the tree, from \mathcal{D}_1 and \mathcal{D}_2 respectively, that contains the node v with index \mathcal{I} . There are two cases:

Case 1. $\text{Root}_1 = \text{Root}_2$. We can use Path_1 as the authentication path between the node v and the root hash Root_2 .

Case 2. $\text{Root}_1 \neq \text{Root}_2$. The extension proof π between \mathcal{D}_1 and \mathcal{D}_2 must contain necessary hashes that enable us to compute Root_2 from \mathcal{D}_1 ; thus, we can construct an authentication path Path' between Root_1 and Root_2 by obtaining hashes from \mathcal{D}_1 and π . Finally, we can construct Path_2 by merging Path_1 and Path' since Path_1 enables us to compute Root_1 from H , and Path' enables us to compute Root_2 from Root_1 . \square

We prove a lemma that is useful in Merkle². Although auditors only check the extension proof between neighboring epochs, there exist a extension proof between any two epochs.

Lemma 2.10.6. *Assume that the hash function used in chronological trees is a collision-resistant hash function [Gol07]. For all polynomial-time adversaries \mathcal{A} there exists a polynomial-time extractor \mathcal{E} and a negligible function $\nu(\cdot)$ such that:*

$$\Pr \left[\begin{array}{l} (\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \pi_{12}, \pi_{23}) \leftarrow \mathcal{A}(1^\lambda); \\ \pi_{13} \leftarrow \mathcal{E}(1^\lambda, \mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \pi_{12}, \pi_{23}) : \\ \text{Extension.Check}(\mathcal{D}_1, \mathcal{D}_2, \pi_{12}) = 1 \\ \quad \wedge \\ \text{Extension.Check}(\mathcal{D}_2, \mathcal{D}_3, \pi_{23}) = 1 \\ \quad \downarrow \\ \text{Extension.Check}(\mathcal{D}_1, \mathcal{D}_3, \pi_{13}) = 1 \end{array} \right] = 1 - \nu(\lambda) .$$

Proof. The proof is similar to the proof of Guarantee 2.10.5. We can construct the extension proof π_{13} by obtaining necessary hashes from π_{12} and π_{23} . \square

2.10.3 Security proof of Guarantee 1

We will perform a reduction to show that if there exists an adversary \mathcal{B} that can compromise lookup results without being detected, then there exists an adversary \mathcal{A} that can violate the security properties of prefix trees or chronological trees, the collision-resistance of the hash function, or the existential unforgeability of the signature scheme.

Since \mathcal{B} is not detected, \mathcal{B} can violate the property in Guarantee 1 for a user u in C who looks up ID in epoch e , where $E_1 < e \leq E_2$, while remaining undetected by the owner r of ID, the user u , and auditors in A . We denote by S_e the ordered list of ID-value pairs and their position numbers that are appended by r before epoch e , and S'_e the ordered list received by u as the lookup result in epoch e . Because $S_e \neq S'_e$, \mathcal{B} 's attack must fall into one of three cases.

1. The first element in S_e and S'_e are identical, but there exist an element that is in S'_e but NOT in S_e .
2. The first element in S_e and S'_e are identical, but there exist an element that is NOT in S'_e but in S_e .
3. The first element in S_e and S'_e are NOT identical.

We will show that no matter which of the above three cases describes \mathcal{B} 's attack, \mathcal{A} can either break the security of Merkle trees, find a hash collision, or forge the signature. We denote by p the position number of the first element elem in S_e , and p' the position number of the first element elem' in S'_e .

Case 1. In this case, the first ID-value pair in the lookup result is correct ($p = p'$ and $\text{elem} = \text{elem}'$), but there exist other ID-value pairs that are not appended by the ID owner r . Due to the signature chain design, each ID-value pair is associated with a verifying key.

The fact that the first elements of S_e and S'_e are identical does not imply that the user can retrieve the correct verifying key associated with the first element in the Merkle²'s design; thus, we begin by proving that the verifying key that the user u received for the first ID-value pair is the one appended by the owner, r . The lookup protocol has the user u check the prefix tree of a node in the chronological forest with the index $[L:R]$ in epoch e , where $L \leq p \leq R$ and $[L:R]$ is the root node in epoch e (if $L = R$, the user u will check the node hash directly). The monitoring protocol also has the owner r check the prefix tree of the node $[L:R]$ in epoch E_2 .

We denote by $H_{[L:R]}$ the hash of the node in the chronological forest with index $[L:R]$. If $e = E_2$, the hash $H_{[L:R]}$ is the root hash in the digest $\mathcal{D}_{E_2}(\mathcal{D}_e)$. By our assumptions in Guarantee 1, both the owner r and the user u retrieve the same digest of epoch E_2 (which is equal to e) from honest auditors in A ; thus, they will see the same hash for $[L:R]$. If $e < E_2$, because auditors in A check digests in epochs $e \dots E_2$, by Guarantee 2.10.5 and Lemma 2.10.6, there exists an authentication path for the node $[L:R]$ and digest \mathcal{D}_{E_2} . By Guarantee 2.10.4, the owner r and the user u will see the same node hash $H_{[L:R]}$.

If the user u and the owner r see different verifying keys, they will obtain different prefix tree root hashes for $[L:R]$; otherwise, Guarantee 2.10.2 will be violated. Recall that they also obtain the

same node hash $H_{[L:R]}$. Therefore, if \mathcal{B} cheated without being detected by the user u and the owner r , we will successfully find a collision for $H_{[L:R]}$.

Now, we conclude that the user u received the correct verifying key associated with the first ID-value pair. We denote by elem'' the first element in S'_e but not in S_e . The user u will check the signature chain when receiving the lookup result; thus, elem'' must be associated with a signature that can be verified using the verifying key associated with the previous element. Because elem'' is the first compromised element in S'_e , the attacker must provide a valid signature without knowing the signing key associated with the previous element. This violates the existential unforgeability of the signature scheme.

Case 2. In this case, the first ID-value pair in the lookup result is correct, but there exists an ID-value pair that the user u did not receive but was appended by the owner, r . We denote by p the position number of the ID-value pair $\langle \text{ID}, \text{val} \rangle$ that is in S_e but not in S'_e . Similar to the previous case, both the user u and the owner r are supposed to check the prefix tree associated with the node $[L:R]$ where $L \leq p \leq R$ and $[L:R]$ is the root node in epoch e . Using the same argument, the owner r and the user u will see the same node hash $H_{[L:R]}$. However, because the user u did not receive $\langle \text{ID}, \text{val} \rangle$, u must check the non-membership proof of the prefix tree $[L:R]$ as specified in the lookup protocol. u and r will obtain different prefix tree root hashes for $[L:R]$ due to Guarantee 2.10.2. Therefore, we can find a collision for $H_{[L:R]}$.

Case 3. In this case, the attacker \mathcal{B} circumvents the signature chain by forging the first ID-value pair in S'_e ($\text{elem} \neq \text{elem}'$). If $p \leq p'$, the server must hide elem , otherwise the user u will notice elem for the same ID before elem' . Then, we can use a similar argument as in case 1 to find a collision for the hash $H_{[L:R]}$, where $L \leq p \leq R$ and $[L:R]$ is the root in epoch e .

We denote by $[L':R']$ the node index where $L' \leq p' \leq R'$ and $[L':R']$ is the root in epoch e . We now consider the case where $p' < p \leq R'$. The monitoring protocol has the owner check the prefix tree of $[L':R']$, which is supposed to contain elem . However, elem is not in S'_e ; thus, we can apply the same argument as in case 1 to find a collision for $H_{[L':R']}$.

What about $p' \leq R' < p$? The owner r will check the prefix tree associated with $[L':R']$ in the first-value proof. For simplicity, we assume the owner r checks the first-value proof in epoch $E_1 + 1$. The owner will choose a minimum set of chronological trees $\text{Ct}_1, \dots, \text{Ct}_n$ to cover leaf nodes from 0 to $p - 1$. Because $[L':R']$ is the root node in epoch e and $R' < p$, there must exist a chronological tree Ct_i whose root node is $[L':R']$. Through the first-value proof, the owner r will check the prefix tree associated with the root node of Ct_i ; it is supposed not to contain elem' . However, the user u can find elem' in the prefix tree of $[L':R']$. By Guarantee 2.10.2 and Guarantee 2.10.5, the user u and the owner r cannot obtain the same prefix tree root hash. But, the user u and the owner r obtain the same node hash $H_{[L':R']}$. Therefore, we find a collision for $H_{[L':R']}$.

2.11 Optimizations

2.11.1 Pre-build strategy

In this section, we describe the pre-build strategy in Merkle². The pre-build strategy amortizes the construction of prefix trees so that they are incrementally built as ID-value pairs are appended to Merkle². In other words, before a chronological tree internal node N_i is even created, Merkle² maintains its prefix tree P_i ; as ID-value pairs are appended to Merkle², they are appended to P_i if they will be in the subtree rooted at N_i . In the process, the cost of building any prefix tree P_i is amortized across all appends to Merkle². Thus, when creating N_i , there is no additional overhead for building P_i .

For every internal node N_i —even those not yet created—in the ancestor chain of the ID-value pair, we must append the ID-value pair to the corresponding prefix tree P_i . Because we have not set a limit on the maximum height of Merkle², each ID-value pair would have infinite ancestor nodes; thus, we choose a maximum height H so that the root to leaf path in Merkle² may not exceed H . Now, each ID-value pair will be a part of at most H prefix trees; so, we perform H appends to pre-built prefix trees in the ancestor chain, resulting in $O(H \log n)$ operations on each append. Because $H = O(\log(n))$, the cost of each append is still $O(\log^2(n))$.

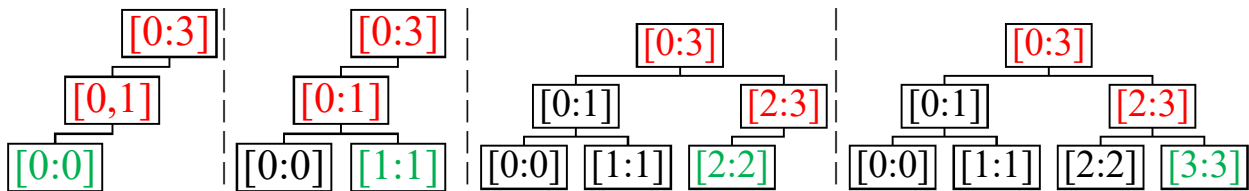


Figure 2.15: The progression of Merkle² through 4 appends. The maximum height is set to $H = 2$. The green nodes represent the most recent appends to Merkle². Red indicates the internal nodes whose prefix trees the ID-value pair was appended to.

Fig. 2.15 illustrates the progression of Merkle² through 4 appends. The first ID-value pair is appended to the green node [0:0]. To maintain the pre-build strategy, the ID-value pair is also appended to the prefix trees of nodes [0:1] and [0:3]. Note that the internal nodes at [0:1] and [0:3] have yet to be created but that the pre-build strategy requires that we maintain their corresponding prefix trees. The next ID-value pair is appended to the node [1:1]. It is also added to the prefix trees of internal nodes [0:1] and [0:3] since they are in its ancestor chain. After the second append, the subtree rooted at [0:1] will no longer change because no future ID-value pairs will be added to [0:1]. The third append will be added to the node [2:2] and also to the prefix trees of nodes [2:3] and [0:3]. Finally, the last ID-value pair is appended to the node [3:3] and the prefix trees of nodes [2:3] and [0:3], completing this instance of Merkle² with height 2.

2.11.2 Compression algorithm

In this section, we explain the compression algorithm for prefix trees. We show that a prefix tree with $O(n)$ ID-value pairs can be compressed so that it requires only $O(n)$ nodes.

We have already introduced prefix trees in Section 2.10.1 . We observe that many of the internal nodes in the prefix tree only have one child, and a chain of such nodes can be represented by a single compressed node instead; thus, in a compressed prefix tree, each node will either have 2 children or 0 children (leaf nodes). Each node will also store a partial prefix representing the compressed path. The internal node hash additionally includes the partial prefix. For example, in Fig. 2.16, nodes on the path from the root node to the leaf node associated with Alice’s value only have one child; thus, they can be compressed as a single node in the compressed prefix tree. The partial prefix “001” represents the compressed path.

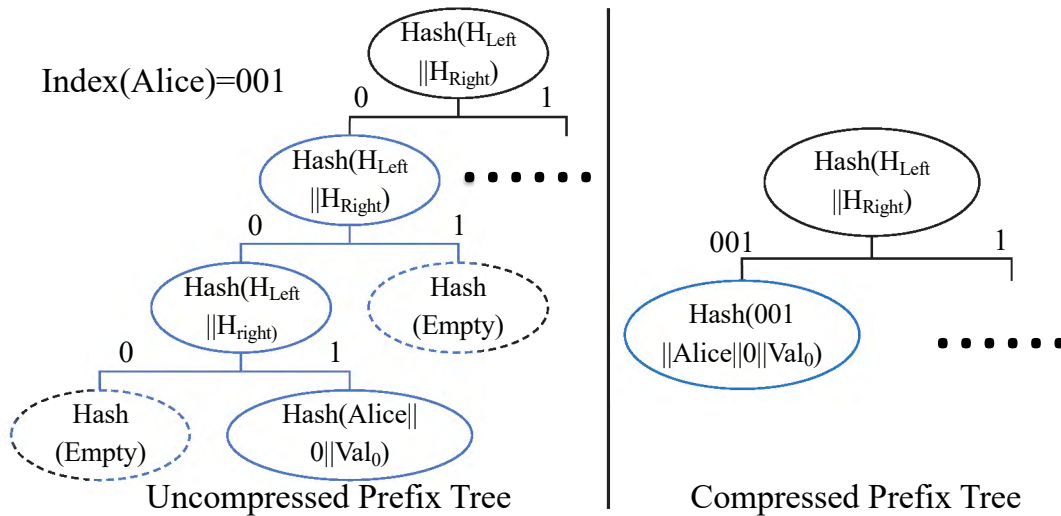


Figure 2.16: Blue nodes in the left tree will be compressed into one node in the right tree because they are all nodes (internal or leaf) with at most one child.

Now we explain how to add a new ID-value pair to the compressed prefix tree. If there exists no value for the ID in the prefix tree, we will find a compressed node v whose index only partially matches the ID. We denote by p the shared prefix between the index of v and the ID. Then, we split the node v and add three new compressed nodes: (1) a new node v' that replaces the old node v with the index p ; (2) a child node u_0 of v' with the index matching that of v ; (3) a child node u_1 of v' with the index matching the ID. The partial prefix of new nodes can be computed accordingly. The node u_1 will store the hash of the new ID-value pair.

In Fig. 2.17, if a new ID-value pair with index “0110” is added, we will find the node X with index “010”. Because “010” is not the prefix of “0110”, we must decompress the node X into three nodes (node A, B, C). The index of node A is “01”, which is the prefix of “0110”, and node B inherits the index and children of node X. We also add node C as the child of node A to store the new ID-value pair.

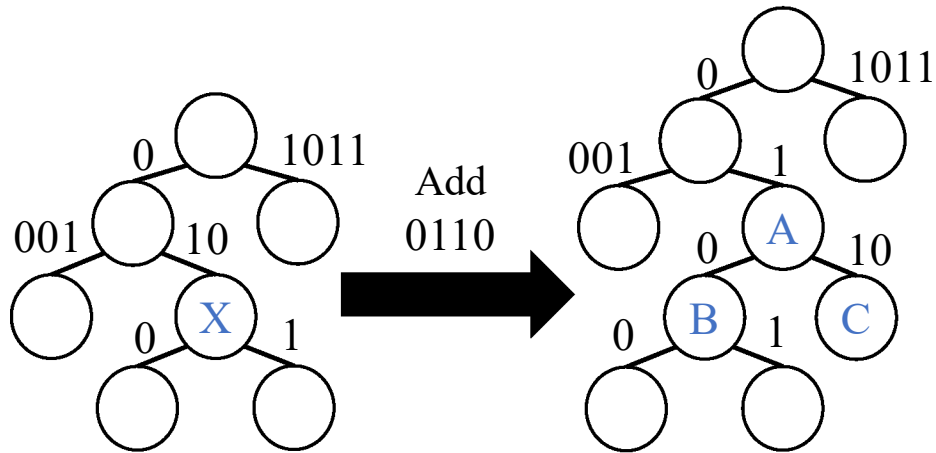


Figure 2.17: Node X will be split into three nodes (A, B, C). The new ID-value pair will be stored in node C.

Next, we analyze the storage cost of a compressed prefix tree. When adding a new ID-value pair, we only add a constant number (three) of new nodes to the prefix tree; thus, a compressed prefix tree with $O(n)$ appends only requires $O(n)$ nodes. Note that although adding a new ID-value pair creates only three new nodes, it still costs $O(\log(n))$ time per append because we must update the hash of $O(\log(n))$ ancestor nodes.

Chapter 3

MARLIN: Preprocessing zkSNARKs with Universal and Updatable SRS

We present a methodology to construct preprocessing zkSNARKs where the structured reference string (SRS) is universal and updatable. This exploits a novel use of *holography* [Babai et al., STOC 1991], where fast verification is achieved provided the statement being checked is given in encoded form.

We use our methodology to obtain a preprocessing zkSNARK where the SRS has linear size and arguments have constant size. Our construction improves on Sonic [Maller et al., CCS 2019], the prior state of the art in this setting, in all efficiency parameters: proving is an order of magnitude faster and verification is thrice as fast, even with smaller SRS size and argument size. Our construction is most efficient when instantiated in the algebraic group model (also used by Sonic), but we also demonstrate how to realize it under concrete knowledge assumptions. We implement and evaluate our construction.

The core of our preprocessing zkSNARK is an efficient *algebraic holographic proof* (AHP) for rank-1 constraint satisfiability (R1CS) that achieves linear proof length and constant query complexity.

This work was previously published in [CHMMVW20].

3.1 Introduction

Succinct non-interactive arguments (SNARGs) are efficient certificates of membership in non-deterministic languages. Recent years have seen a surge of interest in zero-knowledge SNARGs of knowledge (zkSNARKs), with researchers studying constructions under different cryptographic assumptions, improvements in asymptotic efficiency, concrete performance of implementations, and numerous applications. The focus of this paper is *SNARGs in the preprocessing setting*, a notion that we motivate next.

When is fast verification possible? The size of a SNARG must be, as a minimum condition, sublinear in the size of the non-deterministic witness, and often is required to be even smaller (e.g., logarithmic in the size of the non-deterministic computation). The time to verify a SNARG would be, ideally, as fast as reading the SNARG. *This is in general too much to hope for, however.* The verification procedure must also read the *description* of the computation, in order know what statement is being verified. While there are natural computations that have succinct descriptions (e.g., machine computations), in general the description of a computation could be as large as the computation itself, which means that the time to verify the SNARG could be asymptotically comparable to the size of the computation. This is unfortunate because there is a very useful class of computations for which we cannot expect fast verification: general circuit computations.

The preprocessing setting. An approach to avoid the above limitation is to design a verification procedure that has two phases: an offline phase that produces a short summary for a given circuit; and an online phase that uses this short summary to verify SNARGs that attest to the satisfiability of the circuit with different partial assignments to its input wires. Crucially, now the online phase could in principle be as fast as reading the SNARG (and the partial assignment), and thus sublinear in the circuit size. This goal was captured by *preprocessing SNARGs* [Gro10; Lip12; GGPR13; BCIOP13], which have been studied in an influential line of works that has led to highly-efficient constructions that fulfill this goal (e.g., [Gro16]) and large-scale deployments in the real world that benefit from the online fast verification (e.g., [Zcab]).

The problem: circuit-specific SRS. The offline phase in efficient constructions of preprocessing SNARGS consists of sampling a structured reference string (SRS) that depends on the circuit that is being preprocessed. This implies that producing/validating proofs with respect to different circuits requires different SRSs. In many applications of interest, there is no single party that can be entrusted with sampling the SRS, and so real-world deployments have had to rely on cryptographic “ceremonies” [ZcashMPC] that use secure multi-party sampling protocols [BCGTV15; BGG17; BGM17; ABLSZ19]. However, any modification in the circuit used in an application requires another cryptographic ceremony, which is unsustainable for many applications.

A solution: universal SRS. The above motivates preprocessing SNARGs where the SRS is *universal*, which means that the SRS supports any circuit up to a given size bound by enabling anyone, in an offline phase *after* the SRS is sampled, to publicly derive a circuit-specific SRS.¹ Known techniques to obtain a universal SRS from circuit-specific SRS introduce expensive overheads due

¹Even better than a universal SRS would be a URS (uniform reference string). However, achieving preprocessing SNARGs in the URS model with small argument size remains an open problem; see Section 3.1.2.

to universal simulation [BCTV14a; BCTV14b]. Also, these techniques lead to universal SRSs that are not *updatable*, a property introduced in [GKMMM18] that significantly simplifies cryptographic ceremonies. The recent work of Maller et al. [MBKM19] overcomes these shortcomings, obtaining the first efficient construction of a preprocessing SNARG with universal (and updatable) SRS. Even so, the construction in [MBKM19] is considerably more expensive than the state of the art for circuit-specific SRS [Gro16]. In this paper we ask: *can the efficiency gap between universal SRS and circuit-specific SRS be closed, or at least significantly reduced?*

Concurrent work. A concurrent work [GWC19] studies the same question as this paper. See Section 3.1.2 for a brief discussion that compares the two works.

construction		argument size over BN-256 (bytes)			argument size over BLS12-381 (bytes)			
Sonic [MBKM19]		1152			1472			
MARLIN [this work]		704			880			
Groth16 [Gro16]		128			192			

zkSNARK construction	sizes				time complexity			
		$ \text{ipk} $	$ \text{ivk} $	$ \pi $	generator	indexer	prover	verifier
Sonic [MBKM19]	\mathbb{G}_1	$8m$	—	20	8 f-MSM(M)	4 v-MSM($3m$)	273 v-MSM(m)	7 pairings
	\mathbb{G}_2	$8m$	3	—	8 f-MSM(M)	—	—	
	\mathbb{F}_q	—	—	16	—	$O(m \log m)$	$O(m \log m)$	
MARLIN [this work]	\mathbb{G}_1	$4m$	2	13	1 f-MSM($3M$)	12 v-MSM(m)	22 v-MSM(m)	2 pairings
	\mathbb{G}_2	—	2	—	—	—	—	
	\mathbb{F}_q	—	—	8	—	$O(m \log m)$	$O(m \log m)$	
Groth16 [Gro16]	\mathbb{G}_1	$4n$	$O(\mathbb{x})$	2	4 f-MSM(n)	—	4 v-MSM(n)	1 v-MSM($ \mathbb{x})$
	\mathbb{G}_2	n	$O(1)$	1	1 f-MSM(n)	N/A	1 v-MSM(n)	3 pairings
	\mathbb{F}_q	—	—	—	$O(m + n \log n)$	—	$O(m + n \log n)$	—

n : number of multiplication gates in the circuit

m : total number of (addition or multiplication) gates in the circuit

M : maximum supported circuit size (maximum number of addition and multiplication gates)

Figure 3.1: Comparison of two preprocessing zkSNARKs with universal (and updatable) SRS: the prior state of the art and our construction. We include the current state of the art for circuit-specific SRS (in gray), for reference. Here $\mathbb{G}_1/\mathbb{G}_2/\mathbb{F}_q$ denote the number of elements or operations over the respective group/field; also, f-MSM(m) and v-MSM(m) denote fixed-base and variable-base multi-scalar multiplications (MSM) each of size m , respectively. The number of pairings that we report for Sonic’s verifier is lower than that reported in [MBKM19] because we account for standard batching techniques for pairing equations.

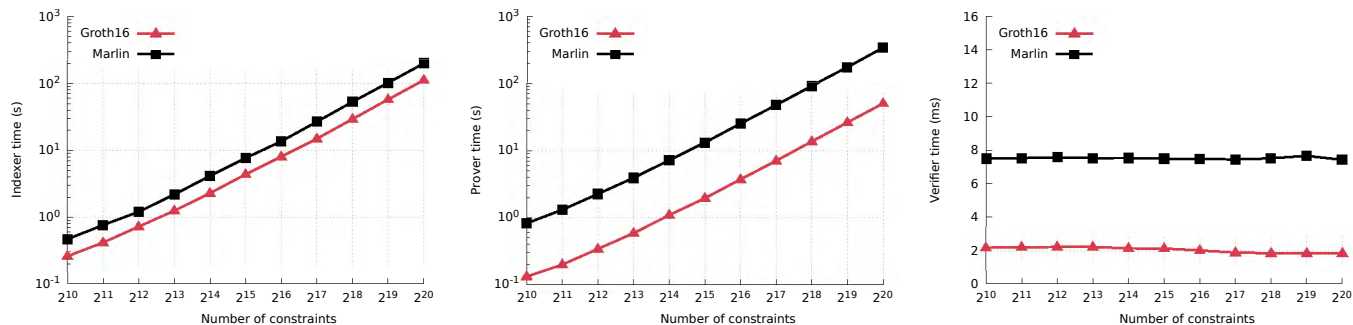


Figure 3.2: Measured performance of MARLIN and [Gro16] over the BLS12-381 curve. We could not include measurements for [MBKM19, Sonic] because at the time of writing there is no working implementation of its unhelped variant.

3.1.1 Our results

In this paper we present MARLIN, a new preprocessing zkSNARK with universal (and updatable) SRS that improves on the prior state of the art [MBKM19, Sonic] in essentially all relevant efficiency parameters.² In addition to reducing argument size by several group and field elements and reducing time complexity of the verifier by over $3\times$, our construction overcomes the main efficiency drawback of [MBKM19, Sonic]: the cost of producing proofs. Indeed, our construction improves time complexity of the prover by over $10\times$, achieving prover efficiency comparable to the case of preprocessing zkSNARKs with *circuit-specific* SRS. In Fig. 3.1 we provide a comparison of our construction and [MBKM19, Sonic], including argument sizes for two popular elliptic curves; the table also includes the state of the art for circuit-specific SRS. We have implemented MARLIN in a Rust library,³ and report evaluation results in Fig. 3.2.

Our zkSNARK is the result of several contributions that we deem of independent interest, summarized below.

(1) A new methodology. We present a general methodology to construct preprocessing SNARGs (and also zkSNARKs) where the SRS is universal (and updatable). The methodology in fact produces succinct *interactive* arguments that can be made non-interactive via the Fiat–Shamir transformation [FS86]. Hence below we focus on *preprocessing arguments with universal and updatable SRS* (see Section 3.7 for the definition).

Our key observation is that the ability to preprocess a circuit in an offline phase is closely related to constructing “holographic proofs” [BFLS91], which means that the verifier does not receive the circuit description as an input but, rather, makes a small number of queries to an encoding of it. These queries are in addition to queries that the verifier makes to proofs sent by the prover.

²Maller et al. [MBKM19] discuss two variants of their protocol, a cheaper one for the “helped setting” and a costlier one for the “unhelped setting”. The variant that is relevant to this paper is the latter one, because it is a preprocessing zkSNARK. (The former variant does not achieve succinct verification, and instead achieves a weaker guarantee that applies to proof batches.)

³<https://github.com/scipr-lab/marlin>

Moreover, in this paper we focus on the setting where the encoding of the circuit description consists of low-degree polynomials and also where proofs are themselves low-degree polynomials — this can be viewed as a requirement that honest and malicious provers are “algebraic”. We call these *algebraic holographic proofs* (AHPs); see Section 3.4 for definitions.

We present a transformation that “compiles” any public-coin AHP into a corresponding preprocessing argument with universal (and updatable) SRS by using suitable polynomial commitments.

Theorem 1 (informal version of Theorem 3.8.1). *There is an efficient transformation that combines any public-coin AHP for a relation \mathcal{R} and an extractable polynomial commitment scheme to obtain a public-coin preprocessing argument with universal SRS for the relation \mathcal{R} . The transformation preserves zero knowledge and proof of knowledge of the underlying AHP. The SRS is updatable provided the SRS of the polynomial commitment scheme is.*

The above transformation provides us with a *methodology* to construct preprocessing zkSNARKs with universal SRS (see Fig. 3.3). Namely, to improve the efficiency of preprocessing zkSNARKs with universal SRS it suffices to improve the efficiency of *simpler building blocks*: AHPs (an information-theoretic primitive) and polynomial commitments (a cryptographic primitive).⁴

The improvements achieved by our preprocessing zkSNARK (see Fig. 3.1) were obtained by following this methodology: we designed efficient constructions for each of these two building blocks (which we discuss shortly), combined them via Theorem 1, and then applied the Fiat–Shamir transformation [FS86].

Methodologies that combine information-theoretic probabilistic proofs and cryptographic tools have played a fundamental role in the construction of efficient argument systems. In the particular setting of preprocessing SNARGs, for example, the compiler introduced in [BCIOP13] for *circuit-specific* SRS has paved the way towards current state-of-the-art constructions [Gro16], and also led to constructions that are plausibly post-quantum [BISW17; BISW18]. We believe that our methodology for universal SRS will also be useful in future work, and may lead to further efficiency improvements.

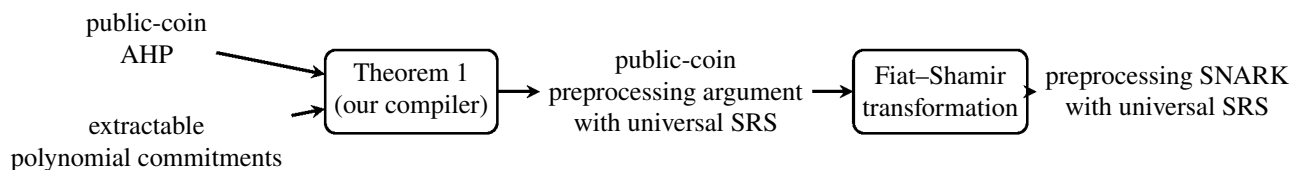


Figure 3.3: Diagram of our methodology to construct preprocessing SNARGs with universal SRS.

(2) An efficient AHP for RICS. We design an algebraic holographic proof (AHP) that achieves linear proof length and constant query complexity, among other useful efficiency features. The

⁴The methodology also captures as a special case various folklore approaches used in prior works to construct *non*-preprocessing zkSNARKs via polynomial commitment schemes (see Section 3.1.2), thereby providing the first formal statement that clarifies what properties of algebraic proofs and polynomial commitment schemes are essential for these folklore approaches.

protocol is for rank-1 constraint satisfiability (R1CS), a well-known generalization of arithmetic circuits where the “circuit description” is given by coefficient matrices (see definition below). Note that the relations that we consider consist of *triples* rather than *pairs*, because we need to split the verifier’s input into a part for the offline phase and a part for the online phase. The offline input is called the *index*, and it consists of the coefficient matrices; the online input is called the *instance*, and it consists of a partial assignment to the variables. The algorithm that encodes the index (coefficient matrices) in the offline phase is called the *indexer*.

Definition 1 (informal). *The indexed relation $\mathcal{R}_{\text{R1CS}}$ is the set of triples*

$$(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) = \left((\mathbb{F}, n, m, A, B, C), x, w \right)$$

where \mathbb{F} is a finite field, A, B, C are $n \times n$ matrices over \mathbb{F} , each containing at most m non-zero entries, and $z := (x, w)$ is a vector in \mathbb{F}^n such that $Az \circ Bz = Cz$. (Here “ \circ ” denotes the entry-wise product.)

Theorem 2 (informal). *There exists a constant-round AHP for the indexed relation $\mathcal{R}_{\text{R1CS}}$ with linear proof length and constant query complexity. The soundness error is $O(m/|\mathbb{F}|)$, and the construction is a zero knowledge proof of knowledge. The arithmetic complexity of the indexer is $O(m \log m)$, of the prover is $O(m \log m)$, and of the verifier is $O(|x| + \log m)$.*

The literature on probabilistic proofs contains algebraic protocols that are holographic (e.g., [BFLS91] and [GKR15]) but *none* achieve constant query complexity, and so applying our methodology (Theorem 1) to these would lead to large argument sizes (many tens of kilobytes). These prior algebraic protocols rely on the multivariate sumcheck protocol applied to certain multivariate polynomials, which means that they incur sizable communication costs due to (a) the many rounds of the sumcheck protocol, and (b) the fact that applying the methodology would involve using multivariate polynomial commitment schemes that (for known constructions) lead to communication costs that are linear in the number of variables.

In contrast, our algebraic protocol relies on univariate polynomials and achieves constant query complexity, incurring small communication costs. Our algebraic protocol can be viewed as a “holographic variant” of the algebraic protocol for R1CS used in Aurora [BCRSVW19], because it achieves an *exponential* improvement in verification time when the verifier is given a suitable encoding of the coefficient matrices; see Table 3.1.

construction	holographic?	indexer	prover	verifier	messages	proof length	queries
[BCRSVW19]	NO	N/A	$O(m + n \log n)$	$O(\mathfrak{x} + n)$	3	$O(n)$	$O(1)$
this work	YES	$O(m \log m)$	$O(m \log m)$	$O(\mathfrak{x} + \log m)$	7	$O(m)$	$O(1)$

Table 3.1: Comparison of the *non*-holographic protocol for R1CS in [BCRSVW19], and the AHP for R1CS that we construct. Here n denotes the number of variables and m the number of non-zero coefficients in the matrices.

(3) Extractable polynomial commitments. Polynomial commitment schemes, introduced in [KZG10], are commitment schemes specialized to work with univariate polynomials. The security properties in [KZG10], while sufficient for the applications therein, do not appear sufficient for standalone use, or even just for the transformation in Theorem 1. We propose a definition for polynomial commitment schemes that incorporates the functionality and security that we believe to suffice for standalone use (and in particular suffices for Theorem 1). Moreover, we show how to extend the construction of [KZG10] to fulfill this definition in the plain model under non-falsifiable knowledge assumptions, or via a more efficient construction in the algebraic group model [FKL18] under falsifiable assumptions. These constructions are of independent interest, and when combined with our transformation, lead to the first efficient preprocessing arguments with universal SRS under concrete knowledge assumptions, and also to the efficiency reported in Fig. 3.1.

We have implemented in a Rust library⁵ the polynomial commitment schemes, and our implementation of MARLIN relies on this library. We deem this library of independent interest for other projects.

3.1.2 Related work

In this paper we study the goal of constructing preprocessing SNARGs with universal SRS, which achieve succinct verification regardless of the structure of the non-deterministic computation being checked. The most relevant prior work is Sonic [MBKM19], on which we improve as already discussed (see Fig. 3.1). The notion of updatable SRS was defined and achieved in [GKMMM18], but with a less efficient construction.

Concurrent work. A concurrent work [GWC19] studies the same question as this paper, and also obtains efficiency improvements over Sonic [MBKM19]. Below is a brief comparison.

- Similarly to our work, [GWC19] extends the polynomial commitment in [KZG10] to support batching, and proves the extension secure in the algebraic group model. We additionally show how to prove security in the plain model under non-falsifiable knowledge assumptions, and consider the problem of enforcing different degrees for different polynomials (a feature that is not needed in [GWC19]).
- We show how to compile any algebraic holographic proof into a preprocessing argument with universal SRS, while [GWC19] focus on compiling a more restricted notion that they call “polynomial protocols”.
- Our protocol natively supports R1CS, and can be viewed as a holographic variant of the algebraic protocol in [BCRSVW19]. The protocol in [GWC19] natively supports a different constraint system, and involves a protocol that, similar to [Gro10], uses a permutation argument to attest that all variables in the same cycle of a permutation are equal (e.g., $(1)(2, 3)(4)$ would require that the second and third entries are equal).

⁵<https://github.com/scipr-lab/poly-commit>

Preprocessing SNARGs with a URS. Setty [Set19] studies preprocessing SNARGs with a URS (uniform reference string), and describes a protocol that for n -gate arithmetic circuits and a chosen constant $c \geq 2$ achieves proving time $O_\lambda(n)$, argument size $O_\lambda(n^{1/c})$, and verification time $O_\lambda(n^{1-1/c})$. The protocol in [Set19] offers a tradeoff compared to our work: preprocessing with a URS instead of a SRS, at the cost of asymptotically larger argument size and verification time. The question of achieving processing with a URS while also achieving asymptotically small argument size and verification time remains open.

The protocol in [Set19] is obtained by combining the multivariate polynomial commitments of [WTSTW18] and a modern rendition of the PCP in [BFLS91] (which itself can be viewed as the “bare bones” protocol of [GKR15] for circuits of depth 1). [Set19] lacks an analysis of concrete costs, and also does not discuss how to achieve zero knowledge beyond stating that techniques in other papers [ZGKPP17a; WTSTW18; XZZPS19] can be applied. Nevertheless, argument sizes would at best be similar to these other papers (tens of kilobytes), which is much larger than our argument sizes (in the SRS model).

We conclude by noting that the informal security proof in [Set19] appears insufficient to show soundness of the argument system, because the polynomial commitment scheme is only assumed to be binding but not also extractable (there is no explanation of where the witness encoded in the committed polynomial comes from). Our definitions and security proofs, if ported over to the multivariate setting, would fill this gap.

Remark 3.1.1. Setty [Set19] also suggests using multivariate polynomial commitments with an SRS [PST13], which could lead to asymptotically smaller argument size and faster verification time. Perhaps because this is not the focus of Spartan (which advocates the benefits of a URS) there are no analyses of security or concrete efficiency for this case. By analogy to arguments with an SRS that use such commitments [XZZPS19], one may guess that Setty’s suggestion would lead to arguments with faster prover time and larger argument sizes (tens of kilobytes) in comparison to our work. Working out the details of this suggestion is left to future work.

Non-preprocessing SNARGs for arbitrary computations. Checking arbitrary circuits without preprocessing them requires the verifier to read the circuit, so the main goal is to obtain small argument size. In this setting of non-preprocessing SNARGs for arbitrary circuits, constructions with a URS (uniform reference string) are based on discrete logarithms [BCCGP16; BBBPWM18] or hash functions [AHIV17; BCRSVW19], while constructions with a universal SRS (structured reference string) combine polynomial commitments and non-holographic algebraic proofs [Gab19]; all use random oracles to obtain non-interactive arguments.⁶

We find it interesting to remark that our methodology from Theorem 1 generalizes protocols such as [Gab19] in two ways. First, it formalizes the folklore approach of combining polynomial commitments and algebraic proofs to obtain arguments, identifying the security properties required

⁶The linear verification time in most of the cited constructions can typically be partially mitigated via techniques that enable an untrusted party to help the verifier to check a batch of proofs for the same circuit faster than checking each proof individually (the linear cost in the circuit is paid only once per batch rather than once for each proof in the batch).

to make this approach work. Second, it demonstrates how for algebraic *holographic* proofs the resulting argument enables preprocessing.

Non-preprocessing SNARGs for structured computations. Several works study SNARGs for structured computations. This structure enables fast verification *without* preprocessing. A line of works [Ben+17; BBHR19; BCGGRS19] combines hash functions and various interactive oracle proofs. Another line of works [ZGKPP17b; ZGKPP18; ZGKPP17a; WTSTW18; XZZPS19] combines multivariate polynomial commitments [PST13] and doubly-efficient interactive proofs [GKR15].

While in this paper we study a different setting (*preprocessing* SNARGs for *arbitrary* computations), there are similarities, and notable differences, in the polynomial commitments used in our work and prior works. We begin by noting that the notion of “multivariate polynomial commitments” varies considerably across prior works, despite the fact that most of those commitments are based on the protocol introduced in [PST13].

- The commitments used in [ZGKPP17b; ZGKPP18] are required to satisfy extractability (a stronger notion than binding) because the security proof of the argument system involves extracting a polynomial encoding a witness. The commitment is a modification of [PST13] that uses knowledge commitments, a standard ingredient to achieve extractability under non-falsifiable assumptions in the plain model. Neither of these works consider hiding commitments as zero knowledge is not a goal for them.
- The commitments used in [ZGKPP17a; WTSTW18] must be compatible with the Cramer–Damgård transform [CD98] used in constructing the argument system. They consider a *modified setting* where the sender does not reveal the value of the commitment polynomial at a desired point but, instead, reveals a commitment to this value, along with a proof attesting that the committed value is correct. For this modified setting, they consider commitments that satisfy natural notions of extractability *and hiding* (achieving zero knowledge arguments is a goal in both papers). The commitments constructed in the two papers offer different tradeoffs. The commitment in [ZGKPP17a] is based on [PST13]: it relies on a SRS (structured reference string); it uses pairings; and for ℓ -variate polynomials achieves $O_\lambda(\ell)$ -size arguments that can be checked in $O_\lambda(\ell)$ time. The commitment in [WTSTW18] is inspired from [BG12] and [BBBPWM18]: it relies on a URS (uniform reference string); it does not use pairings; and for ℓ -variate *multilinear* polynomials and a given constant $c \geq 2$ achieves $O_\lambda(2^{\ell/c})$ -size arguments that can be checked in $O_\lambda(2^{\ell-\ell/c})$ time.
- The commitments used in [XZZPS19] are intended for the regular (unmodified) setting of commitment schemes where the sender reveals the value of the polynomial, because zero knowledge is later achieved by building on the algebraic techniques described in [CFS17]. The commitment definition in [XZZPS19] considers binding and hiding, but not extractability. However, the given security analysis for the argument system does not seem to go through for this definition (there is no explanation of where the witness encoded in the committed polynomial comes from). Also, no commitment construction is provided in [XZZPS19], and

instead the reader is referred to [ZGKPP17a], which considers the modified setting described above.

In sum there are multiple notions of commitment and one must be precise about the functionality and security needed to construct an argument system. We now compare prior notions of commitments to the one that we use.

First, since in this paper we do not use the Cramer–Damgård transform for zero knowledge, commitments in the modified setting are not relevant. Instead, we achieve zero knowledge via *bounded independence* [BCGV16], and in particular we consider the familiar setting where the sender reveals evaluations to the committed polynomial. Second, prior works consider protocols where the sender commits to a polynomial in a single round, while we consider protocols where the sender commits to multiple polynomials of different degrees in each of several rounds. This multi-polynomial multi-round setting requires suitable extensions in terms of functionality (to enable batching techniques to save on argument size) and security (extractability and hiding need to be strengthened), which means that prior definitions do not suffice for us.

The above discrepancies have led us to formulate new definitions of functionality and security for polynomial commitments (as summarized in Section 3.2.2). We conclude by noting that, since in this paper we construct arguments that use *univariate* polynomials, our definitions are specialized to commitments for univariate polynomials. Corresponding definitions for multivariate polynomials can be obtained with straightforward modifications, and would strengthen definitions appearing in some prior works. Similarly, we fulfill the required definitions via natural adaptations of the univariate scheme of [KZG10], and analogous adaptations of the multivariate scheme of [PST13] would fulfill the multivariate analogues of these definitions.

3.2 Techniques

We discuss the main ideas behind our results. First we describe the two building blocks used in Theorem 1: AHPs and polynomial commitment schemes (described in Sections 3.2.1 and 3.2.2 respectively). We describe how to combine these to obtain preprocessing arguments with universal SRS in Section 3.2.3. Next, we discuss constructions for these building blocks: in Section 3.2.4 we describe our AHP (underlying Theorem 2), and in Section 3.2.5 we describe our construction of polynomial commitments.

Throughout, instead of considering the usual notion of relations that consist of instance-witness pairs, we consider *indexed relations*, which consist of triples $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ where \mathfrak{i} is the index, \mathfrak{x} is the instance, and \mathfrak{w} is the witness. This is because \mathfrak{i} represents the part of the verifier input that is preprocessed in the offline phase (e.g., the circuit description) and \mathfrak{x} represents the part of the verifier input that comes in the online phase (e.g., a partial assignment to the circuit’s input wires). The *indexed language* corresponding to an indexed relation \mathcal{R} , denoted $\mathcal{L}(\mathcal{R})$, is the set of pairs $(\mathfrak{i}, \mathfrak{x})$ for which there exists a witness \mathfrak{w} such that $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \in \mathcal{R}$.

3.2.1 Building block: algebraic holographic proofs

Interactive oracle proofs (IOPs) [BCS16; RRR16] are multi-round protocols where in each round the verifier sends a challenge and the prover sends an oracle (which the verifier can query). IOPs combine features of interactive proofs [Bab85; GMR89] and probabilistically checkable proofs [BFLS91; AS98; ALMSS98]. *Algebraic holographic proofs* (AHPs) modify the notion of an IOP in two ways.

- *Holographic*: the verifier does not receive its input explicitly but, rather, has oracle access to a prescribed *encoding* of it. This potentially enables the verifier to run in time that is much faster than the time to read its input in full. (Our constructions will achieve this fast verification.)
- *Algebraic*: the honest prover must produce oracles that are low-degree polynomials (this restricts the completeness property), and all malicious provers must produce oracles that are low-degree polynomials (this relaxes the soundness property). The encoded input to the verifier must also be a low-degree polynomial.

Since in this paper we only work with *univariate* polynomials, our definitions focus on this case, but they can be modified in a straightforward way to be more general.

Informally, a (public-coin) AHP over a field \mathbb{F} for an indexed relation \mathcal{R} is specified by an indexer \mathbf{I} , prover \mathbf{P} , and verifier \mathbf{V} that work as follows.

- *Offline phase*. The indexer \mathbf{I} receives as input the index \mathfrak{i} to be preprocessed, and outputs one or more univariate polynomials over \mathbb{F} encoding \mathfrak{i} .
- *Online phase*. For some instance \mathfrak{x} and witness \mathfrak{w} , the prover \mathbf{P} receives $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ and the verifier \mathbf{V} receives \mathfrak{x} ; \mathbf{P} and \mathbf{V} interact over some (in this paper, constant) number of rounds, where in each round \mathbf{V} sends a challenge and \mathbf{P} sends one or more polynomials; after the

interaction, $V(\mathbb{x})$ probabilistically queries the polynomials output by the indexer and the polynomials output by the prover, and then accepts or rejects. Crucially, V does *not* receive i as input, but instead queries the polynomials output by I that encode i . This enables the construction of verifiers V that run in time that is sublinear in $|i|$.

The completeness property states that for every $(i, \mathbb{x}, w) \in \mathcal{R}$ the probability that $P(i, \mathbb{x}, w)$ convinces $V^{I(i)}(\mathbb{x})$ to accept is 1. The soundness property states that for every $(i, \mathbb{x}) \notin \mathcal{L}(\mathcal{R})$ and *admissible* prover \tilde{P} the probability that \tilde{P} convinces $V^{I(i)}(\mathbb{x})$ to accept is at most a given soundness error ϵ . A prover is “admissible” if the degrees of the polynomials it outputs fit within prescribed degree bounds of the protocol. See Section 3.4 for details on AHPs, including definitions of proof of knowledge and zero knowledge.

Remark 3.2.1 (prior holographic proofs). Various definitions of “holographic proofs” have been studied in the literature on probabilistic proofs, starting with the seminal work of Babai, Fortnow, Levin, and Szegedy [BFLS91]. Recent examples include the IPs in [GKR15], whose verifier runs in sublinear time when given (multivariate low-degree) encodings of the circuit’s wiring predicates and of the circuit’s input; and also the IOPs in [RRR16], where encoded provers and encoded inputs play a role in amortizing interactive proofs.

3.2.2 Building block: polynomial commitments

Informally, a *polynomial commitment scheme* [KZG10] allows a prover to produce a commitment c to a univariate polynomial $p \in \mathbb{F}[X]$, and later “open” $p(X)$ at any point $z \in \mathbb{F}$, producing an *evaluation proof* π showing that the opened value is consistent with the polynomial “inside” c at z . Turning this informal goal into a useful definition requires some care, however, as we explain below. In this paper we propose a set of definitions for polynomial commitment schemes that we believe are useful for standalone use, and in particular suffice as a building block for our compiler described in Sections 3.2.3 and 3.8.

First, we consider constructions with strong efficiency requirements: the commitment c is much smaller than the polynomial p (e.g., c consists of a constant number of group elements), and the proof π can be validated very fast (e.g., in a constant number of cryptographic operations). These requirements not only rule out natural constructions,⁷ but also imply that the usual binding property, which states that an efficient adversary cannot open the same commitment to two different values, does not capture the desired security. Indeed, even if the adversary were to be bound to opening values of some function $f: \mathbb{F} \rightarrow \mathbb{F}$, it may be that the function f is consistent with a polynomial

⁷A natural construction would be to use a standard commitment scheme to commit to each coefficient of p , and then open to a value by revealing the committed coefficients. However, this construction is inefficient, because the commitment c and evaluation proof π are “long” (linear in the degree of p). An alternative construction would be to use a Merkle tree on the coefficients of p . While c now becomes short, the evaluation proof π remains long because the receiver would need to see all coefficients to validate a claimed evaluation. Crucially, both constructions enable the receiver to check the degree of the committed polynomial.

whose degree is *higher* than what was claimed. This means that a security definition needs to incorporate guarantees about the degree of the committed function.⁸

Second, in many applications of polynomial commitments, an adversary produces multiple commitments to polynomials within a round of interaction and across rounds of interaction. After this interaction, the adversary reveals values of all of these polynomials at one or more locations. This setting motivates a number of considerations. First, it is desirable to rely on a single set of public parameters for committing to multiple polynomials, even if the polynomials differ in degree. A construction such as that of [KZG10] can be modified in a natural way to achieve this by committing both to the polynomial and its shift to the maximum degree, similarly to techniques used to bundle multiple low-degree tests into a single one [BCRSVW19]. This modification needs to be addressed in any proof of security. Second, it would be desirable to batch evaluation proofs across different polynomials for the same location. Again the construction in [KZG10] can support this, but one must argue that security still holds in this more general case.

The preceding considerations require an extension of previous definitions and motivate our re-formulation of the primitive. Informally, a polynomial commitment scheme PC is a tuple of algorithms $PC = (\text{Setup}, \text{Trim}, \text{Commit}, \text{Open}, \text{Check})$. The setup algorithm $PC.\text{Setup}$ takes as input a security parameter and maximum supported degree bound D , and outputs public parameters pp that contain the description of a finite field \mathbb{F} . The “trimming” algorithm $PC.\text{Trim}$ then deterministically specializes these parameters for a given set of degree bounds and outputs a committer key ck and a receiver key rk . The sender can then invoke $PC.\text{Commit}$ with input ck and a list of polynomials p with respective degree bounds d to generate a set of commitments c . Subsequently, the sender can use $PC.\text{Open}$ to produce a proof π that convinces the receiver that the polynomials “inside” c respect the degree bounds d and, moreover, evaluate to the claimed set of values v at a given query set Q that specifies any number of evaluation points for each polynomial. The receiver can invoke $PC.\text{Check}$ to check this proof.

The scheme PC is required to satisfy *extractability* and *efficiency* properties, and also, optionally, a *hiding* property. We outline these properties below (see Section 3.6.1 for the details).

Extractability. Consider an efficient sender adversary \mathcal{A} that can produce a commitment c and degree bound $d \leq D$ such that, when asked for an evaluation at some point $z \in \mathbb{F}$, can produce a supposed evaluation v and proof π such that $PC.\text{Check}$ accepts. Then PC is *extractable* if for every maximum degree bound D and every sender adversary \mathcal{A} who can produce such commitments, there exists a corresponding efficient extractor $\mathcal{E}_{\mathcal{A}}$ that outputs a polynomial p of degree at most d that “explains” c so that $p(z) = v$. While for simplicity we have described the most basic case here, our definition considers adversaries and extractors who interact over multiple rounds, wherein the adversary may produce multiple commitments in each round and the extractor is required to output corresponding polynomials on a per-round basis (before seeing the query set, proof, or supposed evaluations).

⁸This consideration motivates the *strong correctness* property in [KZG10], which states that *if* the adversary knows a polynomial that leads to the claimed commitment c then this polynomial has bounded degree. This notion, while sufficient for the application in [KZG10], does not seem to suffice for standalone use because there is no a priori guarantee that an adversary that can open values to a commitment knows a polynomial inside the commitment. In some sense, a knowledge assumption is hidden in this hypothesis.

In this work we rely on extractability to prove the security of our compiler (see Section 3.2.3); we do not know if weaker security notions studied in prior works, such as evaluation binding, suffice. More generally, we believe that extractability is a useful property that may be required across a range of other applications.

Efficiency. We require two notions of efficiency for PC. First, the time required to commit to a polynomial p and then to create an evaluation proof must be proportional to the degree of p , and not to the maximum degree D . (This ensures that the argument prover runs in time proportional to the size of the index.)

On the receiver’s side, the commitment size, proof size, and time to verify an opening must be independent of the claimed degrees for the polynomials. (This ensures that the argument produced by our compiler is succinct.)

Hiding. The hiding property of PC states that commitments and proofs of evaluation reveal no information about the committed polynomial beyond the publicly stated degree bound and the evaluation itself. Namely, PC is *hiding* if there exists an efficient simulator that outputs simulated commitments and simulated evaluation proofs that cannot be distinguished from their real counterparts by any malicious distinguisher that only knows the degree bound and the evaluation.

Analogously to the case of extractability, we actually consider a more general definition that considers commitments to multiple polynomials within and across multiple rounds; moreover, the definition considers the case where some polynomials are designated as not hidden (and thus given to the simulator) because in our application we sometimes prefer to commit to a polynomial in a non-hiding way (for efficiency reasons).

3.2.3 Compiler: from AHPs to preprocessing arguments with universal SRS

We describe the main ideas behind Theorem 1, which uses polynomial commitment schemes to compile any (public-coin) AHP into a corresponding (public-coin) preprocessing argument with universal SRS. In a subsequent step, the argument can be made non-interactive via the Fiat–Shamir transformation, and thereby obtain a preprocessing SNARG with universal SRS.

The basic intuition of the compiler follows the well-known framework of “commit to oracles and then open query answers” pioneered by Kilian [Kil92]. However, the commitment scheme used in our compiler leverages and enforces the algebraic structure of these oracles. While several works in the literature already take advantage of algebraic commitment schemes applied to algebraic oracles, our contribution is to observe that if we apply this framework to a holographic proof then we obtain a preprocessing argument.

Informally, first the argument indexer invokes the AHP indexer to generate polynomials, and then deterministically commits to these using the polynomial commitment scheme. Subsequently, the argument prover and argument verifier interact, each respectively simulating the AHP prover and AHP verifier. In each round, the argument prover sends succinct commitments to the polynomials output by the AHP prover in that round. After the interaction, the argument verifier declares its queries to the polynomials (of the prover and of the indexer). The argument prover replies with the desired evaluations along with an evaluation proof attesting to their correctness relative to the commitments.

This approach, while intuitive, must be proven secure. In particular, in the proof of soundness, we need to show that if the argument prover convinces the argument verifier with a certain probability, then we can find an AHP prover that convinces the AHP verifier with similar probability. This step is non-trivial: the AHP prover outputs polynomials, while the argument prover merely outputs succinct commitments and a few evaluations, which is much less information. In order to deduce the former from the latter requires *extraction*. This motivates considering polynomial commitment schemes that are extractable, in the sense described in Section 3.2.2. We do not know whether weaker security properties, such as the evaluation binding property studied in some prior works, suffice for proving the compiler secure.

The compiler outlined above is compatible with the properties of argument of knowledge and zero knowledge. Specifically, we prove that if the AHP is a proof of knowledge, then the compiler produces an argument of knowledge; also, if the AHP is (bounded-query) zero knowledge and the polynomial commitment scheme is hiding, then the compiler produces a zero knowledge argument.

See Section 3.8 for more details on the compiler.

3.2.4 Construction: an AHP for constraint systems

In prior sections we have described how we can use polynomial commitment schemes to compile AHPs into corresponding preprocessing SNARGs. In this section we discuss the main ideas behind Theorem 2, which provides an efficient AHP for the indexed relation corresponding to R1CS (see Definition 1). The preprocessing zkSNARK that we achieve in this paper (see Fig. 3.1) is based on this AHP.

Our protocol can be viewed as a “holographic variant” of the *non*-holographic algebraic proof for R1CS constructed in [BCRSVW19]. Achieving holography involves designing a new sub-protocol that enables the verifier to evaluate low-degree extensions of the coefficient matrices at a random location. While in [BCRSVW19] the verifier performed this computation in time $\text{poly}(|\mathfrak{i}|)$ on its own, in our protocol the verifier performs it *exponentially faster*, in time $O(\log |\mathfrak{i}|)$, by receiving help from the prover and having oracle access to the polynomials produced by the indexer. We introduce notation and then discuss the protocol.

Some notation. Consider an index $\mathfrak{i} = (\mathbb{F}, n, m, A, B, C)$ specifying coefficient matrices, an instance $\mathfrak{x} = x \in \mathbb{F}^*$ specifying a partial assignment to the variables, and a witness $\mathfrak{w} = w \in \mathbb{F}^*$ specifying an assignment to the other variables such that the R1CS equation holds. The R1CS equation holds if and only if $Az \circ Bz = Cz$ for $z := (x, w) \in \mathbb{F}^n$. Below, we let H and K be prescribed subsets of \mathbb{F} of sizes n and m respectively; we also let $v_H(X)$ and $v_K(X)$ be the vanishing polynomials of these two sets. (The vanishing polynomial of a set S is the monic polynomial of degree $|S|$ that vanishes on S , i.e., $\prod_{\gamma \in S} (X - \gamma)$.) We assume that both H and K are smooth multiplicative subgroups. This allows interpolation/evaluation over H in $O(n \log n)$ operations and also makes $v_H(X)$ computable in $O(\log n)$ operations (and similarly for K). Given an $n \times n$ matrix M with rows/columns indexed by elements of H , we denote by $\hat{M}(X, Y)$ the low-degree extension of M , i.e., the polynomial of individual degree less than n such that $\hat{M}(\kappa, \iota)$ is the (κ, ι) -th entry of M for every $\kappa, \iota \in H$.

A non-holographic starting point. We sketch a *non-holographic* protocol for R1CS with linear proof length and constant query complexity, inspired from [BCRSVW19], that forms the starting point of our work. In this case the prover receives as input $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ and the verifier receives as input $(\mathfrak{i}, \mathfrak{x})$. (The verifier reads the non-encoded index \mathfrak{i} because we are describing a non-holographic protocol.)

In the first message the prover \mathbf{P} sends the univariate polynomial $\hat{z}(X)$ of degree less than n that agrees with the variable assignment z on H , and also sends the univariate polynomials $\hat{z}_A(X), \hat{z}_B(X), \hat{z}_C(X)$ of degree less than n that agree with the linear combinations $z_A := Az, z_B := Bz$, and $z_C := Cz$ on H . The prover is left to convince the verifier that the following two conditions hold:

- (1) Entry-wise product: $\forall \kappa \in H, \hat{z}_A(\kappa)\hat{z}_B(\kappa) - \hat{z}_C(\kappa) = 0$.
- (2) Linear relation: $\forall M \in \{A, B, C\}, \forall \kappa \in H, \hat{z}_M(\kappa) = \sum_{\iota \in H} M[\kappa, \iota]\hat{z}(\iota)$.

(The prover also needs to convince the verifier that $\hat{z}(X)$ encodes a full assignment z that is consistent with the partial assignment x , but we for simplicity we ignore this in this informal discussion.)

In order to convince the verifier of the first (entry-wise product) condition, the prover sends the polynomial $h_0(X)$ such that $\hat{z}_A(X)\hat{z}_B(X) - \hat{z}_C(X) = h_0(X)v_H(X)$. This polynomial equation is equivalent to the first condition (the left-hand side equals zero everywhere on H if and only if it is a multiple of H 's vanishing polynomial). The verifier will check the equation at a random point $\beta \in \mathbb{F}$: it queries $\hat{z}_A(X), \hat{z}_B(X), \hat{z}_C(X), h_0(X)$ at β , evaluates $v_H(X)$ at β on its own, and checks that $\hat{z}_A(\beta)\hat{z}_B(\beta) - \hat{z}_C(\beta) = h_0(\beta)v_H(\beta)$. The soundness error is the maximum degree over the field size, which is at most $2n/|\mathbb{F}|$.

In order to convince the verifier of the second (linear relation) condition, the prover expects a random challenge $\alpha \in \mathbb{F}$ from the verifier, and then replies in a second message. For each $M \in \{A, B, C\}$, the prover sends polynomials $h_M(X)$ and $g_M(X)$ such that

$$r(\alpha, X)\hat{z}_M(X) - r_M(\alpha, X)\hat{z}(X) = h_M(X)v_H(X) + Xg_M(X)$$

for

$$r_M(Z, X) := \sum_{\kappa \in H} r(Z, \kappa)\hat{M}(\kappa, X)$$

where $r(Z, X)$ is a prescribed polynomial of individual degree less than n such that $(r(Z, \kappa))_{\kappa \in H}$ are n linearly independent polynomials. Prior work [BCRSVW19] on checking linear relations via univariate sumchecks shows that this polynomial equation is equivalent, up to a soundness error of $n/|\mathbb{F}|$ over α , to the second condition.⁹ The verifier will check this polynomial equation at the random point $\beta \in \mathbb{F}$: it queries $\hat{z}(X), \hat{z}_A(X), \hat{z}_B(X), \hat{z}_C(X), h_M(X), g_M(X)$ at β , evaluates $v_H(X)$ at β on its own, evaluates $r(Z, X)$ and $r_M(Z, X)$ at (α, β) on its own, and checks that

⁹In particular, we are using the fact from [BCRSVW19] that, given a multiplicative subgroup S of \mathbb{F} , a polynomial $f(X)$ sums to σ over S if and only if $f(X)$ can be written as $h(X)v_S(X) + Xg(X) + \sigma/|S|$ for some $h(X)$ and $g(X)$ with $\deg(g) < |S| - 1$.

$r(\alpha, \beta)\hat{z}_M(\beta) - r_M(\alpha, \beta)\hat{z}(\beta) = h_M(\beta)v_H(\beta) + \beta g_M(\beta)$. The additional soundness error is $2n/|\mathbb{F}|$.

The above is a simple 3-message protocol for RICS with soundness error $\max\{2n/|\mathbb{F}|, 3n/|\mathbb{F}|\} = 3n/|\mathbb{F}|$ in the setting where the honest prover and malicious provers send polynomials of prescribed degrees, which the verifier can query at any location. The proof length (sum of all degrees) is linear in n and the query complexity is constant.

Barrier to holography. The verifier in the above protocol runs in time that is $\Omega(|\mathfrak{i}|) = \Omega(n + m)$. While this is inherent in the non-holographic setting (because the verifier must read \mathfrak{i}), we now discuss how exactly the verifier's computation depends on \mathfrak{i} . We shall later use this understanding to achieve an exponential improvement in the verifier's time when given a suitable encoding of \mathfrak{i} .

The verifier's check for the entry-wise product is $\hat{z}_A(\beta)\hat{z}_B(\beta) - \hat{z}_C(\beta) = h_0(\beta)v_H(\beta)$, and can be carried out in $O(\log n)$ operations *regardless* of the coefficient matrices contained in the index \mathfrak{i} . In other words, this check is efficient even in the non-holographic setting. However, the verifier's check for the linear relation is $r(\alpha, \beta)\hat{z}_M(\beta) - r_M(\alpha, \beta)\hat{z}(\beta) = h_M(\beta)v_H(\beta) + \beta g_M(\beta)$, which has a linear cost. Concretely, evaluating the polynomial $r_M(Z, X)$ at (α, β) requires $\Omega(n + m)$ operations.

In the holographic setting, a natural idea to reduce this cost would be to grant the verifier oracle access to the low-degree extension \hat{M} for $M \in \{A, B, C\}$. This idea has two problems: the verifier *still* needs $\Omega(n)$ operations to evaluate $r_M(Z, X)$ at (α, β) and, moreover, the size of \hat{M} is *quadratic* in n , which means that the encoding of the index \mathfrak{i} is $\Omega(n^2)$. We cannot afford such an expensive encoding in the offline preprocessing phase. We now describe how we overcome both of these problems, and obtain a holographic protocol.

Achieving holography. To overcome the above problems and obtain a holographic protocol, we rely yet again on the univariate sumcheck protocol. We introduce two additional rounds of interaction, and in each round the verifier learns that their verification equation holds provided the sumcheck from the next round holds. The last sumcheck will rely on polynomials output by the indexer, which the verifier knows are correct.

We address the first problem by letting the prover and verifier interact in an additional round, where we rely on an additional univariate sumcheck to reduce the problem of evaluating $r_M(Z, X)$ at (α, β) to the problem of evaluating \hat{M} at (β_2, β) for a random $\beta_2 \in \mathbb{F}$. Namely, the verifier sends β to the prover, who computes

$$\sigma_2 := r_M(\alpha, \beta) = \sum_{\kappa \in H} r(\alpha, \kappa)\hat{M}(\kappa, \beta).$$

Then the prover replies with σ_2 and the polynomials $h_2(X)$ and $g_2(X)$ such that

$$r(\alpha, X)\hat{M}(X, \beta) = h_2(X)v_H(X) + Xg_2(X) + \sigma_2/n .$$

Prior techniques on univariate sumcheck [BCRSVW19] tell us that this equation is equivalent to the polynomial $r(\alpha, X)\hat{M}(X, \beta)$ summing to σ_2 on H . Thus the verifier needs to check this equation at a random $\beta_2 \in \mathbb{F}$: $r(\alpha, \beta_2)\hat{M}(\beta_2, \beta) = h_2(\beta_2)v_H(\beta_2) + \beta_2 g_2(\beta_2) + \sigma_2/n$. The only expensive part of this equation for the verifier is computing the value $\hat{M}(\beta_2, \beta)$, which is problematic. Indeed, we

have already noted that we cannot afford to simply let the verifier have oracle access to \hat{M} , because this polynomial has quadratic size (it contains a quadratic number of terms).

We address this second problem as follows. Let $u_H(X, Y) := \frac{v_H(X) - v_H(Y)}{X - Y}$ be the formal derivative of the vanishing polynomial $v_H(X)$, and note that $u_H(X, Y)$ vanishes on the square $H \times H$ except for on the diagonal, where it takes on the (non-zero) values $(u_H(a, a))_{a \in H}$. Moreover, $u_H(X, Y)$ can be evaluated at any point in $\mathbb{F} \times \mathbb{F}$ in $O(\log n)$ operations. Using this polynomial, we can write \hat{M} as a sum of $m = |K|$ terms instead of $n^2 = |H|^2$ terms:

$$\hat{M}(X, Y) := \sum_{\kappa \in K} u_H(X, \text{r\hat{o}w}_M(\kappa)) \cdot u_H(Y, \text{c\hat{o}l}_M(\kappa)) \cdot \text{v\hat{a}l}_M(\kappa) ,$$

where $\text{r\hat{o}w}_M, \text{c\hat{o}l}_M, \text{v\hat{a}l}_M$ are the low-degree extensions of the row, column, and value of the non-zero entries in M according to some canonical order over K .¹⁰

This method of representing the low-degree extension of M suggests an idea: let the verifier have oracle access to the polynomials $\text{r\hat{o}w}_M, \text{c\hat{o}l}_M, \text{v\hat{a}l}_M$ and do *yet another* univariate sumcheck, but this time over the set K . The verifier sends β_2 to the prover, who computes

$$\sigma_3 := \hat{M}(\beta_2, \beta) = \sum_{\kappa \in K} u_H(\beta_2, \text{r\hat{o}w}_M(\kappa)) \cdot u_H(\beta, \text{c\hat{o}l}_M(\kappa)) \cdot \text{v\hat{a}l}_M(\kappa) .$$

Then the prover replies with σ_3 and the polynomials $h_3(X)$ and $g_3(X)$ such that

$$u_H(\beta_2, \text{r\hat{o}w}_M(X))u_H(\beta, \text{c\hat{o}l}_M(X))\text{v\hat{a}l}_M(X) = h_3(X)v_K(X) + Xg_3(X) + \sigma_3/m .$$

The verifier can then check this equation at a random $\beta_3 \in \mathbb{F}$, which only requires $O(\log m)$ operations.

The above idea *almost* works; the one remaining problem is that $h_3(X)$ has degree $\Omega(nm)$ (because the left-hand side of the equation has quadratic degree), which is too expensive for our target of a quasilinear-time prover. We overcome this problem by letting the prover run the univariate sumcheck protocol on the unique low-degree extension $\hat{f}(X)$ of the function $f: K \rightarrow \mathbb{F}$ defined as $f(\kappa) := u_H(\beta_2, \text{r\hat{o}w}_M(\kappa))u_H(\beta, \text{c\hat{o}l}_M(\kappa))\text{v\hat{a}l}_M(\kappa)$. Observe that $\hat{f}(X)$ has degree less than m . The verifier checks that $\hat{f}(X)$ and $u_H(\beta_2, \text{r\hat{o}w}_M(X))u_H(\beta, \text{c\hat{o}l}_M(X))\text{v\hat{a}l}_M(X)$ agree on K .

From sketch to protocol. In the above discussion we have ignored a number of technical aspects, such as proof of knowledge and zero knowledge (which are ultimately needed in the compiler if we want to construct a preprocessing zkSNARK). We have also not discussed time complexities of many algebraic steps, and we omitted discussion of how to batch multiple sumchecks into fewer ones, which brings important savings in argument size. For details, see our detailed construction in Section 3.5.

3.2.5 Construction: extractable polynomial commitments

We now sketch how to construct a polynomial commitment scheme that achieves the strong functionality and security requirements of our definition in Section 3.2.2. Our starting point is

¹⁰Technicality: $\text{v\hat{a}l}(\kappa)$ actually equals the value divided by $u_H(\text{r\hat{o}w}_M(\kappa), \text{r\hat{o}w}_M(\kappa))u_H(\text{c\hat{o}l}_M(\kappa), \text{c\hat{o}l}_M(\kappa))$.

the $\text{PolyCommit}_{\text{DL}}$ construction of Kate et al. [KZG10], and then describe a sequence of natural and generic transformations that extend this construction to enable extractability, commitments to multiple polynomials, and the enforcement of per-polynomial degree bounds. In fact, once we arrive at a scheme that supports extractability for committed polynomials at a single point (the full version[CHMMVW20]), our transformations build on this construction in a black box way to first support per-polynomial degree bounds (the full version[CHMMVW20]), and then query sets that may request multiple evaluation points per polynomial (the full version[CHMMVW20]). Indeed, it is sufficient to produce a polynomial commitment scheme that satisfies the much more simple interface and definitions in the full version[CHMMVW20], and apply these black box transformations to obtain a polynomial commitment scheme that satisfies the interface of and provides the properties described in Section 3.6.1 ultimately needed by our compiler.

Starting point: $\text{PolyCommit}_{\text{DL}}$. The setup phase samples a cryptographically secure bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, G, H, e)$ and then samples a committer key ck and receiver key rk for a given degree bound D . The committer key consists of group elements encoding powers of a random field element β , namely, $\text{ck} := \{G, \beta G, \dots, \beta^D G\} \in \mathbb{G}_1^{D+1}$. The receiver key consists of the group elements $\text{rk} := (G, H, \beta H) \in \mathbb{G}_1 \times \mathbb{G}_2^2$. Note that the SRS, which consists of the keys ck and rk , is updatable because the coefficients of group elements in the SRS are all monomials (see Remark 3.7.1).

To commit to a polynomial $p \in \mathbb{F}_q[X]$, the sender computes $c := p(\beta)G$. To subsequently prove that the committed polynomial evaluates to v at a point z , the sender computes a witness polynomial $w(X) := (p(X) - p(z))/(X - z)$, and provides as proof a commitment to w : $\pi := w(\beta)G$. The idea is that the witness function w is a polynomial *if and only if* $p(z) = v$; otherwise, it is a rational function, and cannot be committed to using ck .

Finally, to verify a proof of evaluation, the receiver checks that the commitment and proof of evaluation are consistent. That is, it checks that the proof commits to a polynomial of the form $(p(X) - p(z))/(X - z)$ by checking the equality $e(c - vG, H) = e(\pi, \beta H - zH)$.

Achieving extractability. While the foregoing construction guarantees correctness of evaluations, it does not by itself guarantee that a commitment actually “contains” a suitable polynomial of degree at most D . We study two methods to address this issue, and thereby achieve extractability. One method is to modify the construction to use knowledge commitments [Gro10], and rely on a concrete knowledge assumption. The main disadvantage of this approach is that each commitment doubles in size. The other method is to move away from the plain model, and instead conduct the security analysis in the algebraic group model (AGM) [FKL18]. This latter method is more efficient because each commitment remains a single group element.

Committing to multiple polynomials at once. We enable the sender to simultaneously open multiple polynomials $[p_i]_{i=1}^n$ at the same point z as follows. Before generating a proof of evaluation for $[p_i]_{i=1}^n$, the sender requests from the receiver a random field element ξ , which he uses to take a random linear combination of the polynomials: $p := \sum_{i=1}^n \xi^i p_i$, and generates a proof of evaluation π for this polynomial p .

The receiver verifies π by using the fact that the commitments are additively homomorphic. The receiver takes a linear combination of the commitments and claimed evaluations, obtaining the combined commitment $c = \sum_{i=1}^n \xi^i c_i$ and evaluation $v = \sum_{i=1}^n \xi^i v_i$. Finally, it checks the pairing equations for c , π , and v .

Completeness of this check is straightforward, while soundness follows from the fact that if any polynomial does not match its evaluation, then the combined polynomial will not match its evaluation with high probability.

Enforcing multiple degree bounds. The construction so far enforces a single bound D on the degrees of all the polynomials p_i . To enforce a different degree bound d_i for each p_i , we require the sender to commit not only to each p_i , but also to “shifted polynomials” $p'_i(X) := X^{D-d_i}p_i(X)$. The proof of evaluation proves that, if p_i evaluates to v_i at z , then p'_i evaluates to $z^{D-d_i}v_i$.

The receiver checks that the commitment for each p'_i corresponds to an evaluation $z^{D-d_i}v_i$ so that, if z is sampled from a super-polynomial subset of \mathbb{F}_q , the probability that $\deg(p_i) \neq d_i$ is negligible. This trick is similar to the one used in [BS08; BCRSVW19] to derive low-degree tests for specific degree bounds.

However, while sound, this approach is inefficient in our setting: the witness polynomial for p'_i has $\Omega(D)$ non-zero coefficients (instead of $O(d_i)$), and so constructing an evaluation proof for it requires $\Omega(D)$ scalar multiplications (instead of $O(d_i)$). To work around this, we instead produce a proof that the related polynomial $p_i^*(X) := p'_i(X) - p_i(z)X^{D-d_i}$ evaluates to 0 at z . As we show in the full version [CHMMVW20], the witness polynomial for this claim has $O(d_i)$ non-zero coefficients, and so constructing the evaluation proof can be done in $O(d_i)$ scalar multiplications. Completeness is preserved because the receiver can check the correct evaluation of p_i^* by subtracting $p_i(z)(\beta^{D-d_i}\mathbb{G})$ from the commitment to the shifted polynomial p'_i , thereby obtaining a commitment to p_i^* , while security is preserved because $p'_i(z) = z^{D-d_i}v_i \iff p_i^*(z) = 0$.

Evaluating at a query set instead of a single point. To support the case where the polynomials $[p_i]_{i=1}^n$ are evaluated at a set of points Q , the sender proceeds as follows. Say that there are k different points $[z_i]_{i=1}^k$ in Q . The sender partitions the polynomials $[p_i]_{i=1}^n$ into different groups such that every polynomial in a group is to be evaluated at the same point z_i . The sender runs PC.Open on each group, and outputs the resulting list of evaluation proofs.

Achieving hiding. To additionally achieve hiding, we follow the above blueprint, replacing $\text{PolyCommit}_{\text{DL}}$ with the hiding scheme $\text{PolyCommit}_{\text{ped}}$ described in [KZG10].

3.3 Preliminaries

We denote by $[n]$ the set $\{1, \dots, n\} \subseteq \mathbb{N}$. We use $\mathbf{a} = [a_i]_{i=1}^n$ as a short-hand for the tuple (a_1, \dots, a_n) , and $[\mathbf{a}_i]_{i=1}^n = [[a_{i,j}]_{j=1}^m]_{i=1}^n$ as a short-hand for the tuple

$$(a_{1,1}, \dots, a_{1,m}, \dots, a_{n,1}, \dots, a_{n,m});$$

$|\mathbf{a}|$ denotes the number of entries in \mathbf{a} . If x is a binary string then $|x|$ denotes its bit length. If M is a matrix then $\|M\|$ denotes the number of nonzero entries in M . If S is a finite set then $|S|$ denotes its cardinality and $x \leftarrow S$ denotes that x is an element sampled at random from S . We denote by \mathbb{F} a finite field, and whenever \mathbb{F} is an input to an algorithm we implicitly assume that \mathbb{F} is represented in a way that allows efficient field arithmetic. Given a finite set S , we denote by \mathbb{F}^S the set of vectors indexed by elements in S . We denote by $\mathbb{F}[X]$ the ring of univariate polynomials over \mathbb{F} in X , and by $\mathbb{F}^{<d}[X]$ the set of polynomials in $\mathbb{F}[X]$ with degree less than d .

We denote by $\lambda \in \mathbb{N}$ a security parameter. When we state that $n \in \mathbb{N}$ for some variable n , we implicitly assume that $n = \text{poly}(\lambda)$. We denote by $\text{negl}(\lambda)$ an unspecified function that is *negligible* in λ (namely, a function that vanishes faster than the inverse of any polynomial in λ). When a function can be expressed in the form $1 - \text{negl}(\lambda)$, we say that it is *overwhelming* in λ . When we say that \mathcal{A} is an *efficient adversary* we mean that \mathcal{A} is a family $\{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$ of non-uniform polynomial-size circuits. If the adversary consists of multiple circuit families $\mathcal{A}_1, \mathcal{A}_2, \dots$ then we write $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \dots)$.

Given two interactive algorithms A and B , we denote by $\langle A(x), B(y) \rangle(z)$ the output of $B(y, z)$ when interacting with $A(x, z)$. Note that this output could be a random variable. If we use this notation when A or B is a circuit, we mean that we are considering a circuit that implements a suitable next-message function to interact with the other party of the interaction.

3.3.1 Indexed relations

An *indexed relation* \mathcal{R} is a set of triples $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ where \mathfrak{i} is the index, \mathfrak{x} is the instance, and \mathfrak{w} is the witness; the corresponding *indexed language* $\mathcal{L}(\mathcal{R})$ is the set of pairs $(\mathfrak{i}, \mathfrak{x})$ for which there exists a witness \mathfrak{w} such that $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \in \mathcal{R}$. For example, the indexed relation of satisfiable boolean circuits consists of triples where \mathfrak{i} is the description of a boolean circuit, \mathfrak{x} is a partial assignment to its input wires, and \mathfrak{w} is an assignment to the remaining wires that makes the circuit to output 0. Given a size bound $N \in \mathbb{N}$, we denote by \mathcal{R}_N the restriction of \mathcal{R} to triples $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ with $|\mathfrak{i}| \leq N$.

3.4 Algebraic holographic proofs

We define *algebraic holographic proofs* (AHPs), the notion of proofs that we use. For simplicity, the formal definition below is tailored to univariate polynomials, because our AHP construction is in this setting. The definition can be modified in a straightforward way to consider the general case of multivariate polynomials.

We represent polynomials through the coefficients that define them, as opposed to through their evaluation over a sufficiently large domain (as is typically the case in probabilistic proofs). This definitional choice is due to the fact that we will consider verifiers that may query the polynomials at any location in the field of definition. Moreover, the field of definition itself can be chosen from a given field family, and so we make the field an additional input to all algorithms; this degree of freedom is necessary when combining this component with polynomial commitment schemes (see Section 3.8). Finally, we consider the setting of *indexed relations* (see Section 3.3.1), where the verifier's input has two parts, the index and the instance; in the definition below, the verifier receives the index encoded and the instance explicitly.

Formally, an **algebraic holographic proof** (AHP) over a field family \mathcal{F} for an indexed relation \mathcal{R} is specified by a tuple

$$\text{AHP} = (k, s, d, \mathbf{I}, \mathbf{P}, \mathbf{V})$$

where $k, s, d: \{0, 1\}^* \rightarrow \mathbb{N}$ are polynomial-time computable functions and $\mathbf{I}, \mathbf{P}, \mathbf{V}$ are three algorithms known as the *indexer*, *prover*, and *verifier*. The parameter k specifies the number of interaction rounds, s specifies the number of polynomials in each round, and d specifies degree bounds on these polynomials.

In the offline phase (“0-th round”), the indexer \mathbf{I} receives as input a field $\mathbb{F} \in \mathcal{F}$ and an index \mathfrak{i} for \mathcal{R} , and outputs $s(0)$ polynomials $p_{0,1}, \dots, p_{0,s(0)} \in \mathbb{F}[X]$ of degrees at most $d(|\mathfrak{i}|, 0, 1), \dots, d(|\mathfrak{i}|, 0, s(0))$ respectively. Note that the offline phase does not depend on any particular instance or witness, and merely considers the task of encoding the given index \mathfrak{i} .

In the online phase, given an instance \mathfrak{x} and witness \mathfrak{w} such that $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \in \mathcal{R}$, the prover \mathbf{P} receives $(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ and the verifier \mathbf{V} receives $(\mathbb{F}, \mathfrak{x})$ and oracle access to the polynomials output by $\mathbf{I}(\mathbb{F}, \mathfrak{i})$. The prover \mathbf{P} and the verifier \mathbf{V} interact over $k = k(|\mathfrak{i}|)$ rounds.

For $i \in [k]$, in the i -th round of interaction, the verifier \mathbf{V} sends a message $\rho_i \in \mathbb{F}^*$ to the prover \mathbf{P} ; then the prover \mathbf{P} replies with $s(i)$ oracle polynomials $p_{i,1}, \dots, p_{i,s(i)} \in \mathbb{F}[X]$. The verifier may query any of the polynomials it has received any number of times. A query consists of a location $z \in \mathbb{F}$ for an oracle $p_{i,j}$, and its corresponding answer is $p_{i,j}(z) \in \mathbb{F}$. After the interaction, the verifier accepts or rejects.

The function d determines which provers to consider for the completeness and soundness properties of the proof system. In more detail, we say that a (possibly malicious) prover $\tilde{\mathbf{P}}$ is **admissible** for AHP if, on every interaction with the verifier \mathbf{V} , it holds that for every round $i \in [k]$ and oracle index $j \in [s(i)]$ we have $\deg(p_{i,j}) \leq d(|\mathfrak{i}|, i, j)$. The honest prover \mathbf{P} is required to be admissible under this definition.

We say that AHP has perfect completeness and soundness error ϵ if the following holds.

- **Completeness.** For every field $\mathbb{F} \in \mathcal{F}$ and index-instance-witness tuple $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \in \mathcal{R}$, the probability that $\mathbf{P}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ convinces $\mathbf{V}^{\mathbf{I}(\mathbb{F}, \mathfrak{i})}(\mathbb{F}, \mathfrak{x})$ to accept in the interactive oracle protocol is 1.
- **Soundness.** For every field $\mathbb{F} \in \mathcal{F}$, index-instance pair $(\mathfrak{i}, \mathfrak{x}) \notin \mathcal{L}(\mathcal{R})$, and admissible prover $\tilde{\mathbf{P}}$, the probability that $\tilde{\mathbf{P}}$ convinces $\mathbf{V}^{\mathbf{I}(\mathbb{F}, \mathfrak{i})}(\mathbb{F}, \mathfrak{x})$ to accept in the interactive oracle protocol is at most ϵ .

The *proof length* l is the sum of all degree bounds in the offline and online phases, $l(|\mathfrak{i}|) := \sum_{i=0}^{k(|\mathfrak{i}|)} \sum_{j=1}^{s(i)} d(|\mathfrak{i}|, i, j)$. The intuition for this definition is that in a probabilistic proof each oracle would consist of the evaluation of a polynomial over a domain whose size (in field elements) is linearly related to its degree bound, so that the resulting proof length would be linearly related to the sum of all degree bounds.

The *query complexity* q is the total number of queries made by the verifier to the polynomials. This includes queries to the polynomials output by the indexer and those sent by the prover.

All AHPs that we construct achieve the stronger property of *knowledge soundness* (against admissible provers), and optionally also *zero knowledge*. We define both of these properties below. **Knowledge soundness.** We say that AHP has knowledge error ϵ if there exists a probabilistic polynomial-time extractor \mathbf{E} for which the following holds. For every field $\mathbb{F} \in \mathcal{F}$, index \mathfrak{i} , instance \mathfrak{x} , and admissible prover $\tilde{\mathbf{P}}$, the probability that $\mathbf{E}^{\tilde{\mathbf{P}}}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, 1^{l(|\mathfrak{i}|)})$ outputs \mathfrak{w} such that $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \in \mathcal{R}$ is at least the probability that $\tilde{\mathbf{P}}$ convinces $\mathbf{V}^{\mathbf{I}(\mathbb{F}, \mathfrak{i})}(\mathbb{F}, \mathfrak{x})$ to accept minus ϵ . Here the notation $\mathbf{E}^{\tilde{\mathbf{P}}}$ means that the extractor \mathbf{E} has black-box access to each of the next-message functions that define the interactive algorithm $\tilde{\mathbf{P}}$. (In particular, the extractor \mathbf{E} can “rewind” the prover $\tilde{\mathbf{P}}$.) Note that since \mathbf{E} receives the proof length $l(|\mathfrak{i}|)$ in unary, \mathbf{E} has enough time to receive, and perform efficient computations on, polynomials output by $\tilde{\mathbf{P}}$.

Zero knowledge. We say that AHP has (perfect) zero knowledge with query bound b and query checker \mathbf{C} if there exists a probabilistic polynomial-time simulator \mathbf{S} such that for every field $\mathbb{F} \in \mathcal{F}$, index-instance-witness tuple $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \in \mathcal{R}$, and (b, \mathbf{C}) -query algorithm $\tilde{\mathbf{V}}$ the random variables $\text{View}(\mathbf{P}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, \mathfrak{w}), \tilde{\mathbf{V}})$ and $\mathbf{S}^{\tilde{\mathbf{V}}}(\mathbb{F}, \mathfrak{i}, \mathfrak{x})$, defined below, are identical. Here, we say that an algorithm is (b, \mathbf{C}) -query if it makes at most b queries to oracles it has access to, and each query individually leads the checker \mathbf{C} to output “ok”.

- $\text{View}(\mathbf{P}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, \mathfrak{w}), \tilde{\mathbf{V}})$ is the *view* of $\tilde{\mathbf{V}}$, namely, is the random variable (r, a_1, \dots, a_q) where r is $\tilde{\mathbf{V}}$'s randomness and a_1, \dots, a_q are the responses to $\tilde{\mathbf{V}}$'s queries determined by the oracles sent by $\mathbf{P}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, \mathfrak{w})$.
- $\mathbf{S}^{\tilde{\mathbf{V}}}(\mathbb{F}, \mathfrak{i}, \mathfrak{x})$ is the output of $\mathbf{S}(\mathbb{F}, \mathfrak{i}, \mathfrak{x})$ when given straightline access to $\tilde{\mathbf{V}}$ (\mathbf{S} may interact with $\tilde{\mathbf{V}}$, *without rewinding*, by exchanging messages with $\tilde{\mathbf{V}}$ and answering any oracle queries along the way), *prepended* with $\tilde{\mathbf{V}}$'s randomness r . Note that r could be of super-polynomial size, so \mathbf{S} cannot sample r on $\tilde{\mathbf{V}}$'s behalf and then output it; instead, as in prior work, we restrict \mathbf{S} to not see r , and prepend r to \mathbf{S} 's output.

A special case of interest. We only consider AHPs that satisfy the following properties.

- *Public coins:* AHP is *public-coin* if each verifier message to the prover is a uniformly random string of some prescribed length (or an empty string). Hence the verifier’s randomness is its messages $\rho_1, \dots, \rho_k \in \mathbb{F}^*$ and possibly additional randomness $\rho_{k+1} \in \mathbb{F}^*$ used after the interaction. All verifier queries can be postponed, without loss of generality, to a query phase that occurs after the interactive phase with the prover.
- *Non-adaptive queries:* AHP is *non-adaptive* if all of the verifier’s query locations are solely determined by the verifier’s randomness and inputs (the field \mathbb{F} and the instance \mathbb{x}).

Given these properties, we can view the verifier as two subroutines that execute in the query phase: a query algorithm \mathbf{Q}_V that produces query locations based on the verifier’s randomness, and a decision algorithm \mathbf{D}_V that accepts or rejects based on the answers to the queries (and the verifier’s randomness). In more detail, \mathbf{Q}_V receives as input the field \mathbb{F} , the instance \mathbb{x} , and randomness $\rho_1, \dots, \rho_k, \rho_{k+1}$, and outputs a query set Q consisting of tuples $((i, j), z)$ to be interpreted as “query $p_{i,j}$ at $z \in \mathbb{F}$ ”; and \mathbf{D}_V receives as input the field \mathbb{F} , the instance \mathbb{x} , answers $(v_{((i,j),z)})_{((i,j),z) \in Q}$, and randomness $\rho_1, \dots, \rho_k, \rho_{k+1}$, and outputs the decision bit.

While the above properties are not strictly necessary for the compiler that we describe in Section 3.8, all “natural” protocols that we are aware of (including those that we construct in this paper) satisfy these properties, and so we restrict our attention to public-coin non-adaptive protocols for simplicity.

3.5 AHP for constraint systems

We construct an AHP for *rank-1 constraint satisfiability* (R1CS) that has linear proof length and constant query complexity. Below we define the indexed relation that represents this problem, and then state our result.

Definition 3.5.1 (R1CS indexed relation). *The indexed relation $\mathcal{R}_{\text{R1CS}}$ is the set of all triples*

$$(i, \mathbb{x}, \mathbb{w}) = \left((\mathbb{F}, H, K, A, B, C), x, w \right)$$

where \mathbb{F} is a finite field, H and K are subsets of \mathbb{F} , A, B, C are $H \times H$ matrices over \mathbb{F} with $|K| \geq \max\{\|A\|, \|B\|, \|C\|\}$, and $z := (x, w)$ is a vector in \mathbb{F}^H such that $Az \circ Bz = Cz$.

Theorem 3.5.2. *There exists an AHP for the indexed relation $\mathcal{R}_{\text{R1CS}}$ that is a zero knowledge proof of knowledge with the following features. The indexer uses $O(|K| \log |K|)$ field operations and outputs $O(|K|)$ field elements. The prover and verifier exchange 7 messages. To achieve zero knowledge against b queries (with a query checker \mathcal{C} that rejects queries in H), the prover uses $O((|K| + b) \log(|K| + b))$ field operations and outputs a total of $O(|H| + b)$ field elements. The verifier makes $O(1)$ queries to the encoded index and to the prover's messages, has soundness error $O((|K| + b)/|\mathbb{F}|)$, and uses $O(|x| + \log |K|)$ field operations.*

Remark 3.5.3 (restrictions on domains). Our protocol uses the univariate sumcheck of [BCRSVW19] as a subroutine, and in particular inherits the requirement that the domains H and K must be additive or multiplicative subgroups of the field \mathbb{F} . For simplicity, in our descriptions we use multiplicative subgroups because we use this case in our implementation; the case of additive subgroups involves only minor modifications. Moreover, the arithmetic complexities for the indexer and prover stated in Theorem 3.5.2 assume that the domains H and K are “FFT-friendly” (e.g., they have smooth sizes); this is not a requirement, since in general the arithmetic complexities will be that of an FFT over the domains H and K . Note that we can assume without loss of generality that $|H| = O(|K|)$, for otherwise (if $|K| < |H|/3$) then are empty rows or columns across the matrices that we can drop and reduce their size. Finally, we assume that $|H| \leq |\mathbb{F}|/2$.

This section is organized as follows: in Section 3.5.1 we introduce algebraic notations and facts used in this section; in Section 3.5.2 we describe an AHP for checking linear relations; and in Section 3.5.3 we build on this latter to obtain an AHP for R1CS.

Throughout we assume that H and K come equipped with bijections $\phi_H : H \rightarrow [|H|]$ and $\phi_K : K \rightarrow [|K|]$ that are computable in linear time. Moreover, we define the two sets $H[\leq k] := \{\kappa \in H : 1 \leq \phi_H(\kappa) \leq k\}$ and $H[> k] := \{\kappa \in H : \phi_H(\kappa) > k\}$ to denote the first k elements in H and the remaining elements, respectively. We can then write that $x \in \mathbb{F}^{H[\leq |x|]}$ and $w \in \mathbb{F}^{H[> |x|]}$.

3.5.1 Algebraic preliminaries

Polynomial encodings. For a finite field \mathbb{F} , subset $S \subseteq \mathbb{F}$, and function $f : S \rightarrow \mathbb{F}$ we denote by \hat{f} the (unique) univariate polynomial over \mathbb{F} with degree less than $|S|$ such that $\hat{f}(a) = f(a)$ for every

$a \in S$. We sometimes abuse notation and write \hat{f} to denote *some* polynomial that agrees with f on S , which need not equal the (unique) such polynomial of smallest degree.

Vanishing polynomials. For a finite field \mathbb{F} and subset $S \subseteq \mathbb{F}$, we denote by v_S the unique non-zero monic polynomial of degree at most $|S|$ that is zero everywhere on S ; v_S is called the *vanishing polynomial* of S . If S is an additive or multiplicative coset in \mathbb{F} then v_S can be evaluated in $\text{polylog}(|S|)$ field operations. For example, if S is a multiplicative subgroup of \mathbb{F} then $v_S(X) = X^{|S|} - 1$ and, more generally, if S is a ξ -coset of a multiplicative subgroup S_0 (namely, $S = \xi S_0$) then $v_S(X) = \xi^{|S|} v_{S_0}(X/\xi) = X^{|S|} - \xi^{|S|}$; in either case, v_S can be evaluated in $O(\log |S|)$ field operations.

Derivative of vanishing polynomials. We rely on various properties of a bivariate polynomial u_S introduced in [BCGGRS19]. For a finite field \mathbb{F} and subset $S \subseteq \mathbb{F}$, we define

$$u_S(X, Y) := \frac{v_S(X) - v_S(Y)}{X - Y},$$

which is a polynomial of individual degree $|S| - 1$ because $X - Y$ divides $X^i - Y^i$ for any positive integer i . Note that $u_S(X, X)$ is the formal derivative of the vanishing polynomial $v_S(X)$. The bivariate polynomial $u_S(X, Y)$ satisfies two useful algebraic properties. First, the univariate polynomials $(u_S(X, a))_{a \in S}$ are linearly independent, and $u_S(X, Y)$ is their (unique) low-degree extension. Second, $u_S(X, Y)$ vanishes on the square $S \times S$ except for on the diagonal, where it takes on the (non-zero) values $(u_S(a, a))_{a \in S}$.

If S is an additive or multiplicative coset in \mathbb{F} , $u_S(X, Y)$ can be evaluated at any $(\alpha, \beta) \in \mathbb{F}^2$ in $\text{polylog}(|S|)$ field operations because in this case both v_S (and its derivative) can be evaluated in $\text{polylog}(|S|)$ field operations. For example, if S is a multiplicative subgroup then $u_S(X, Y) = (X^{|S|} - Y^{|S|})/(X - Y)$ and $u_S(X, X) = |S|X^{|S|-1}$, so both can be evaluated in $O(\log |S|)$ field operations.

Univariate sumcheck for subgroups. Prior work [BCRSVW19] shows that, given a multiplicative subgroup S of \mathbb{F} , a polynomial $f(X)$ sums to σ over S if and only if $f(X)$ can be written as $h(X)v_S(X) + Xg(X) + \sigma/|S|$ for some $h(X)$ and $g(X)$ with $\deg(g) < |S| - 1$. This can be viewed as a univariate sumcheck protocol, and we shall rely on it throughout this section.

3.5.2 AHP for the lincheck problem

The *lincheck problem* for univariate polynomials considers the task of deciding whether two polynomials encode vectors that are linearly related in a prescribed way. In more detail, the problem is parametrized by a field \mathbb{F} , two subsets H and K of \mathbb{F} , and a matrix $M \in \mathbb{F}^{H \times H}$ with $|K| \geq \|M\| > 0$. Given oracle access to two low-degree polynomials $f_1, f_2 \in \mathbb{F}^{<d}[X]$, the problem asks to decide whether for every $a \in H$ it holds that $f_1(a) = \sum_{b \in H} M_{a,b} \cdot f_2(b)$, by asking a small number of queries to f_1 and f_2 . The matrix M thus prescribes the linear relations that relate the values of f_1 and f_2 on H .

Ben-Sasson et al. [BCRSVW19] solve this problem by reducing the lincheck problem to a sumcheck problem, and then reducing the sumcheck problem to low-degree testing (of univariate polynomials). In particular, this prior work achieves a 2-message algebraic *non-holographic* protocol

that solves the lincheck problem with linear proof length and constant query complexity. In this section we show how to achieve a 6-message algebraic *holographic* protocol, again with linear proof length and constant query complexity. In Section 3.5.2.1 we describe the indexer algorithm, in Section 3.5.2.2 we describe the prover and verifier algorithms, and in Section 3.5.2.3 we analyze the protocol. Fig. 3.4 summarizes the protocol.

3.5.2.1 Offline phase: encoding the linear relation

The indexer \mathbf{I} for the lincheck problem receives as input a field \mathbb{F} , two subsets H and K of \mathbb{F} , and a matrix $M \in \mathbb{F}^{H \times H}$ with $|K| \geq \|M\|$. The non-zero entries of M are assumed to be presented in some canonical order (e.g., row-wise or column-wise). The output of \mathbf{I} is three univariate polynomials $\hat{\text{row}}, \hat{\text{col}}, \hat{\text{val}}$ over \mathbb{F} of degree less than $|K|$ such that the following polynomial is a low-degree extension of M :

$$\hat{M}(X, Y) := \sum_{\kappa \in K} u_H(X, \hat{\text{row}}(\kappa)) u_H(Y, \hat{\text{col}}(\kappa)) \hat{\text{val}}(\kappa) . \quad (3.1)$$

The three three aforementioned polynomials are the (unique) low-degree extensions of the three functions $\text{row}, \text{col}, \text{val}: K \rightarrow \mathbb{F}$ that respectively represent the row index, column index, and value of the non-zero entries of the matrix M . In more detail, for every $\kappa \in K$ with $1 \leq \phi_K(\kappa) \leq \|M\|$:

- $\text{row}(\kappa) := \phi_H^{-1}(t_\kappa)$ where t_κ is the row index of the $\phi_K(\kappa)$ -th nonzero entry in M ;
- $\text{col}(\kappa) := \phi_H^{-1}(t_\kappa)$ where t_κ is the column index of the $\phi_K(\kappa)$ -th nonzero entry in M ;
- $\text{val}(\kappa)$ is the value of the $\phi_K(\kappa)$ -th nonzero entry in M , divided by $u_H(\text{row}(\kappa), \text{row}(\kappa)) u_H(\text{col}(\kappa), \text{col}(\kappa))$.

Also, $\text{val}(\kappa)$ returns the element 0 for every $\kappa \in K$ with $\phi_K(\kappa) > \|M\|$, while $\text{row}(\kappa)$ and $\text{col}(\kappa)$ return an arbitrary element in H for such κ . The evaluation tables of these functions can be found in $O(|K| \log |H|)$ operations, from which interpolation yields the desired polynomials in $O(|K| \log |K|)$ operations.

Recall from Section 3.5.1 that the bivariate polynomial $u_H(X, Y)$ vanishes on the square $H \times H$ except for on the diagonal, where it takes on the (non-zero) values $(u_H(a, a))_{a \in H}$. By construction of the polynomials $\hat{\text{row}}, \hat{\text{col}}, \hat{\text{val}}$, the polynomial $\hat{M}(X, Y)$ agrees with the matrix M everywhere on the domain $H \times H$. The individual degree of $\hat{M}(X, Y)$ is less than $|H|$. Thus, \hat{M} is the unique low-degree extension of M .

We rewrite the polynomial $\hat{M}(X, Y)$ in a form that will be useful later:

Claim 3.5.4.

$$\hat{M}(X, Y) = \sum_{\kappa \in K} \frac{v_H(X)}{(X - \hat{\text{row}}(\kappa))} \cdot \frac{v_H(Y)}{(Y - \hat{\text{col}}(\kappa))} \cdot \hat{\text{val}}(\kappa) . \quad (3.2)$$

Proof. Note that $v_H(\text{r}\hat{\text{ow}}(\kappa)) = v_H(\text{c}\hat{\text{ol}}(\kappa)) = 0$ for every $\kappa \in K$ because $\text{r}\hat{\text{ow}}(X)$ and $\text{c}\hat{\text{ol}}(X)$ map K to H and v_H vanishes on H . Therefore:

$$\begin{aligned} \hat{M}(X, Y) &= \sum_{\kappa \in K} u_H(X, \text{r}\hat{\text{ow}}(\kappa)) \cdot u_H(Y, \text{c}\hat{\text{ol}}(\kappa)) \cdot \hat{\text{val}}(\kappa) \\ &= \sum_{\kappa \in K} \frac{v_H(X) - v_H(\text{r}\hat{\text{ow}}(\kappa))}{X - \text{r}\hat{\text{ow}}(\kappa)} \cdot \frac{v_H(Y) - v_H(\text{c}\hat{\text{ol}}(\kappa))}{Y - \text{c}\hat{\text{ol}}(\kappa)} \cdot \hat{\text{val}}(\kappa) \\ &= \sum_{\kappa \in K} \frac{v_H(X)}{X - \text{r}\hat{\text{ow}}(\kappa)} \cdot \frac{v_H(Y)}{Y - \text{c}\hat{\text{ol}}(\kappa)} \cdot \hat{\text{val}}(\kappa) . \end{aligned}$$

□

3.5.2.2 Online phase: proving and verifying the linear relation

The prover \mathbf{P} for the lincheck problem receives as input a field \mathbb{F} , two subsets H and K of \mathbb{F} , a matrix $M \in \mathbb{F}^{H \times H}$ with $|K| \geq \|M\|$, and two polynomials $f_1, f_2 \in \mathbb{F}^{\leq d}[X]$. The verifier \mathbf{V} for the lincheck problem receives as input the field \mathbb{F} and two subsets H and K of \mathbb{F} ; \mathbf{V} also has oracle access to the polynomials $\text{r}\hat{\text{ow}}, \text{c}\hat{\text{ol}}, \hat{\text{val}}$ output by the indexer \mathbf{I} invoked on appropriate inputs.

The protocol begins with a reduction from a lincheck problem to a sumcheck problem: \mathbf{V} samples a random element $\alpha \in \mathbb{F}$ and sends it to \mathbf{P} . Indeed, letting $r(X, Y)$ denote the polynomial $u_H(X, Y)$, \mathbf{P} is left to convince \mathbf{V} that the following univariate polynomial sums to 0 on H :

$$q_1(X) := r(\alpha, X)f_1(X) - r_M(\alpha, X)f_2(X) \quad \text{where} \quad r_M(X, Y) := \sum_{\kappa \in H} r(X, \kappa)\hat{M}(\kappa, Y) . \quad (3.3)$$

We rely on the univariate sumcheck protocol for this step: \mathbf{P} sends to \mathbf{V} the polynomials $g_1(X)$ and $h_1(X)$ such that $q_1(X) = h_1(X)v_H(X) + Xg_1(X)$. In order to check this polynomial identity, \mathbf{V} samples a random element $\beta_1 \in \mathbb{F}$ with the intention of checking the identity at $X := \beta_1$. For the right-hand side, \mathbf{V} queries g_1 and h_1 at β_1 , and then evaluates $h_1(\beta_1)v_H(\beta_1) + \beta_1g_1(\beta_1)$ in $O(\log |H|)$ operations. For the left-hand side, \mathbf{V} queries f_1 and f_2 at β_1 and then needs to ask help from \mathbf{P} to evaluate $r(\alpha, \beta_1)f_1(\beta_1) - r_M(\alpha, \beta_1)f_2(\beta_1)$. The reason is that while $r(\alpha, \beta_1)$ is easy to evaluate (it requires $O(\log |H|)$ operations), $r_M(\alpha, \beta_1) = \sum_{\kappa \in H} r(\alpha, \kappa)\hat{M}(\kappa, \beta_1)$ in general requires $\Omega(|H||K|)$ operations.

We thus rely on the univariate sumcheck protocol again. We define

$$q_2(X) := r(\alpha, X)\hat{M}(X, \beta_1) \quad (3.4)$$

\mathbf{V} sends β_1 to \mathbf{P} , and then \mathbf{P} replies with the sum $\sigma_2 := \sum_{\kappa \in H} r(\alpha, \kappa)\hat{M}(\kappa, \beta_1)$ and the polynomials $g_2(X)$ and $h_2(X)$ such that $q_2(X) = h_2(X)v_H(X) + Xg_2(X) + \sigma_2/|H|$. In order to check this polynomial identity, \mathbf{V} samples a random element $\beta_2 \in \mathbb{F}$ with the intention of checking the identity at $X := \beta_2$. For the right-hand side, \mathbf{V} queries g_2 and h_2 at β_2 , and then evaluates $h_2(\beta_2)v_H(\beta_2) + \beta_2g_2(\beta_2) + \sigma_2/|H|$ in $O(\log |H|)$ operations. To evaluate the left-hand side,

however, \mathbf{V} needs to ask help from \mathbf{P} . The reason is that while $r(\alpha, \beta_2)$ is easy to evaluate (it requires $O(\log |H|)$ operations), $\hat{M}(\beta_2, \beta_1)$ in general requires $\Omega(|K|)$ operations.

We thus rely on the univariate sumcheck protocol (yet) again: \mathbf{V} sends β_2 to \mathbf{P} , and then \mathbf{P} replies with the value $\sigma_3 := \hat{M}(\beta_2, \beta_1)$, which the verifier must check. Note though that we *cannot* use the sumcheck protocol directly to compute the sum obtained from Eq. (3.1):

$$\hat{M}(\beta_2, \beta_1) = \sum_{\kappa \in K} u_H(\beta_2, \text{r\hat{o}w}(\kappa)) u_H(\beta_1, \text{c\hat{o}l}(\kappa)) \hat{\text{v}a}l(\kappa) .$$

The reason is because the degree of the above addend, if we replace κ with an indeterminate, is $\Omega(|H||K|)$, which means that the degree of the polynomial h_3 sent as part of a sumcheck protocol also has degree $\Omega(|H||K|)$, which is not within our budget of an AHP with proof length $O(|H| + |K|)$. Instead, we make the minor modification that in the earlier rounds β_1 and β_2 are sampled from $\mathbb{F} \setminus H$ instead of \mathbb{F} , and we will leverage the sumcheck protocol to verify the equivalent (well defined) expression from Eq. (3.2):

$$\hat{M}(\beta_2, \beta_1) = \sum_{\kappa \in K} \frac{v_H(\beta_2) v_H(\beta_1) \hat{\text{v}a}l(\kappa)}{(\beta_2 - \text{r\hat{o}w}(\kappa)) (\beta_1 - \text{c\hat{o}l}(\kappa))} .$$

This may appear to be an odd choice, because if we replace κ with an indeterminate in the sum above, we obtain a rational function that is (in general) *not a polynomial*, and so does not immediately fit the sumcheck protocol. Nevertheless, we are still able to use the sumcheck protocol with it, as we now explain.

Define $f_3(X)$ to be the (unique) polynomial of degree less than $|K|$ such that

$$\forall \kappa \in K, f_3(\kappa) = \frac{v_H(\beta_2) v_H(\beta_1) \hat{\text{v}a}l(\kappa)}{(\beta_2 - \text{r\hat{o}w}(\kappa)) (\beta_1 - \text{c\hat{o}l}(\kappa))} . \quad (3.5)$$

The prover computes the polynomials $g_3(X)$ and $h_3(X)$ such that

$$\begin{aligned} f_3(X) &= X g_3(X) + \sigma_3 / |K| , \\ v_H(\beta_2) v_H(\beta_1) \hat{\text{v}a}l(X) - (\beta_2 - \text{r\hat{o}w}(X)) (\beta_1 - \text{c\hat{o}l}(X)) f_3(X) &= h_3(X) v_K(X) . \end{aligned}$$

The first equation demonstrates that f_3 sums to σ_3 over K , and the second equation demonstrates that f_3 agrees with the correct addends over K . These two equations can be combined in a single equation that involves only $g_3(X)$ and $h_3(X)$:

$$v_H(\beta_2) v_H(\beta_1) \hat{\text{v}a}l(X) - (\beta_2 - \text{r\hat{o}w}(X)) (\beta_1 - \text{c\hat{o}l}(X)) (X g_3(X) + \sigma_3 / |K|) = h_3(X) v_K(X) .$$

The prover thus only sends the two polynomials $g_3(X)$ and $h_3(X)$. In order to check this polynomial identity, \mathbf{V} samples a random element $\beta_3 \in \mathbb{F}$ with the intention of checking the identity at $X := \beta_3$. Then \mathbf{V} queries $g_3, h_3, \text{r\hat{o}w}, \text{c\hat{o}l}, \hat{\text{v}a}l$ at β_3 , and then evaluates $v_H(\beta_2) v_H(\beta_1) \hat{\text{v}a}l(\beta_3) - (\beta_2 - \text{r\hat{o}w}(\beta_3)) (\beta_1 - \text{c\hat{o}l}(\beta_3)) (\beta_3 g_3(\beta_3) + \sigma_3 / |K|) = h_3(\beta_3) v_K(\beta_3)$ in $O(\log |K|)$ operations.

If this third test passes then \mathbf{V} can use the value σ_3 in place of $\hat{M}(\beta_2, \beta_1)$ to finish the second test. If this latter passes, \mathbf{V} can in turn use the value σ_2 in place of $r_M(\alpha, \beta_1)$ to finish the first test.

3.5.2.3 Analysis

Soundness. We argue that the soundness error is at most

$$\frac{|H| + 3|K|}{|\mathbb{F}|} + \frac{d + 3|H|}{|\mathbb{F} \setminus H|}.$$

There are four ways in which the verifier could still accept if the lincheck statement is false: if the randomized reduction to the first sumcheck produces a polynomial that sums to zero; or if any one of the three sumchecks accepts despite the claimed sum being incorrect. The probability that the randomized reduction to sumcheck fails is at most the individual degree in X of $r(X, Y)$ divided by $|\mathbb{F}|$, which is less than $|H|/|\mathbb{F}|$. The probability that any one of the sumchecks fail to detect an incorrectly declared sum is at most the maximum degree of the polynomial equation tested in the respective sumcheck divided by the size from which the test element is sampled. The innermost sumcheck has maximum degree less than $3|K|$, the intermediate sumcheck has maximum degree less than $2|H|$, and the outermost sumcheck has maximum degree less than $|H| + d$. These errors add up to the soundness error claimed above.

Efficiency. The protocol consists of 6 messages, with the verifier moving first. The verifier makes a constant number of queries, evaluates v_H and v_K at a constant number of locations, and then performs a constant number of field operations. In particular, the arithmetic complexity of the verifier is $O(\log |H| + \log |K|)$. The prover sends a constant number of polynomials with degrees linearly related to d (the bound on the degrees of f_1 and f_2), $|H|$, and $|K|$. We now argue that prover time is $O((|H| + d) \log(|H| + d) + |K| \log |K|)$. In the first round, the prover sends the coefficients of the polynomials $g_1(X)$ and $h_1(X)$, which can be found in time $O(|K| + (|H| + d) \log(|H| + d))$, as we argue in Lemma 3.5.5. In the second round, the prover sends the field element σ_2 and the polynomials $g_2(X)$ and $h_2(X)$, which can be found in time $O(|K| + |H| \log |H|)$, as we argue in Lemma 3.5.6. In the third round, the prover sends the field element σ_3 and the polynomials $g_3(X)$ and $h_3(X)$, which can be found in time $O(|K| \log |K|)$, as we argue in Lemma 3.5.7.

Lemma 3.5.5 (first round). *The coefficients of the polynomials $g_1(X)$ and $h_1(X)$ can be found in $O(|K| + (|H| + d) \log(|H| + d))$ field operations, when given coefficients of the polynomials $f_1(X)$ and $f_2(X)$, the subsets H and K , and the matrix M (in sparse form).*

Proof. It suffices to find the coefficients of the polynomial $q_1(X)$ from Eq. (3.3), which has degree at most $|H| + d - 2$, because the polynomials $g_1(X)$ and $h_1(X)$ can be found via polynomial long division of $q_1(X)$ by v_H in time $O((|H| + d) \log |H|)$. In turn, $q_1(X)$ can be computed from the coefficients of $f_1(X)$, $f_2(X)$, $r(\alpha, X)$, and $r_M(\alpha, X)$ in time $O((|H| + d) \log(|H| + d))$ via fast polynomial multiplication and polynomial addition. The first two are given to us in coefficient form; to find the coefficients of the latter two polynomials, we can evaluate each of them over H and then interpolate.

The values of $r(\alpha, X)$ on H can be obtained in $O(|H| \log |H|)$ operations via direct computation of formulas described in Section 3.5.1. The problem is now reduced to finding the values of $r_M(\alpha, X)$ on H — this is the “hard part” that motivates the present proof.

Observe that, by definition of r_M (see Eq. (3.3)) and \hat{M} (see Eq. (3.1)), the following holds:

$$\begin{aligned} r_M(\alpha, X) &= \sum_{\kappa_1 \in H} r(\alpha, \kappa_1) \sum_{\kappa_2 \in K} u_H(\kappa_1, \text{row}(\kappa_2)) u_H(X, \hat{\text{col}}(\kappa_2)) \hat{\text{val}}(\kappa_2) \\ &= \sum_{\kappa_2 \in K} u_H(X, \hat{\text{col}}(\kappa_2)) \hat{\text{val}}(\kappa_2) \sum_{\kappa_1 \in H} r(\alpha, \kappa_1) u_H(\kappa_1, \text{row}(\kappa_2)) \\ &= \sum_{\kappa_2 \in K} u_H(X, \hat{\text{col}}(\kappa_2)) \hat{\text{val}}(\kappa_2) r(\alpha, \text{row}(\kappa_2)) u_H(\text{row}(\kappa_2), \text{row}(\kappa_2)) . \end{aligned}$$

The last equality uses the fact that, for every $\kappa_2 \in K$, the summation

$$\sum_{\kappa_1 \in H} r(\alpha, \kappa_1) u_H(\kappa_1, \text{row}(\kappa_2))$$

collapses to a single term corresponding to $\kappa_1 = \text{row}(\kappa_2)$; the other terms, which correspond to $\kappa_1 \neq \text{row}(\kappa_2)$, are zero due to the fact that the polynomial u_H vanishes on the square $H \times H$ except for on its diagonal.

Next, again using the fact that u_H vanishes on the square $H \times H$ except for on its diagonal, we note that for every $\kappa_1 \in H$

$$r_M(\alpha, \kappa_1) = \sum_{\kappa_2 \in K \text{ s.t. } \hat{\text{col}}(\kappa_2) = \kappa_1} u_H(\kappa_1, \hat{\text{col}}(\kappa_2)) \hat{\text{val}}(\kappa_2) \cdot r(\alpha, \text{row}(\kappa_2)) u_H(\text{row}(\kappa_2), \text{row}(\kappa_2)) .$$

In other words, as κ_1 ranges over H , each element of the sum in $r_M(\alpha, \kappa_1)$ contributes a nonzero value precisely when κ_1 equals a particular element of H , namely, when $\kappa_1 = \hat{\text{col}}(\kappa_2)$. Also, since κ_2 ranges only in K , $\text{row}(\kappa_2) = \text{row}(\kappa_2)$, $\hat{\text{col}}(\kappa_2) = \text{col}(\kappa_2)$, and $\hat{\text{val}}(\kappa_2) = \text{val}(\kappa_2)$ are just the row index, column index, and value of the κ_2 -th entry of M (or zero).

This immediately leads to the following strategy to finding the values of $r_M(\alpha, X)$ on H . Initialize for each $\kappa_1 \in H$ a variable for $r_M(\alpha, \kappa_1)$ that is initially set to 0. Then, for each $\kappa_2 \in K$, compute the term $u_H(\text{col}(\kappa_2), \text{col}(\kappa_2)) \text{val}(\kappa_2) r(\alpha, \text{row}(\kappa_2)) u_H(\text{row}(\kappa_2), \text{row}(\kappa_2))$ and add it to the variable for $r_M(\alpha, \text{col}(\kappa_2))$. Since the values $(u_H(\kappa_1, \kappa_1))_{\kappa_1 \in H}$ and $(r(\alpha, \kappa_1))_{\kappa_1 \in H}$ can be precomputed in $O(|H| \log |H|)$ operations, the foregoing strategy can be carried out in $O(|K| + |H| \log |H|)$ operations. \square

Lemma 3.5.6 (second round). *The field element σ_2 and the coefficients of the polynomials $g_2(X)$ and $h_2(X)$ can be found in $O(|K| + |H| \log |H|)$ field operations, when given the subsets H and K and the matrix M (in sparse form).*

Proof. It suffices to find the coefficients of the polynomial $q_2(X)$ from Eq. (3.4), which has degree at most $2|H| - 2$, because the polynomials $g_2(X)$ and $h_2(X)$ can be found via polynomial long division of $q_2(X)$ by v_H in time $O(|H| \log |H|)$, and the sum σ_2 can be found by evaluating $q_2(X)$ over H in time $O(|H| \log |H|)$ and summing in time $O(|H|)$. In turn, $q_2(X)$ can be computed from the coefficients of $r(\alpha, X)$ and of $\hat{M}(X, \beta_1)$ in time $O(|H| \log |H|)$ using fast polynomial multiplication. To find the coefficients of these two polynomials, we can evaluate each of them over

H and then interpolate. The values of $r(\alpha, X)$ on H can be obtained in $O(|H| \log |H|)$ operations. We now need to find the values of $\hat{M}(X, \beta_1)$ on H .

Recall that

$$\hat{M}(X, \beta_1) = \sum_{\kappa \in K} u_H(X, \text{r}\hat{\text{ow}}(\kappa)) u_H(\beta_1, \hat{\text{c}}\hat{\text{ol}}(\kappa)) \hat{\text{v}}\hat{\text{al}}(\kappa) .$$

Using the fact that u_H vanishes on the square $H \times H$ except for the diagonal, we note that for every $\kappa_1 \in H$

$$\hat{M}(\kappa_1, \beta_1) = \sum_{\kappa_2 \in K \text{ s.t. } \text{r}\hat{\text{ow}}(\kappa_2) = \kappa_1} u_H(\kappa_1, \text{r}\hat{\text{ow}}(\kappa_2)) u_H(\beta_1, \hat{\text{c}}\hat{\text{ol}}(\kappa_2)) \hat{\text{v}}\hat{\text{al}}(\kappa_2) .$$

Thus, to find the values of $\hat{M}(X, \beta_1)$ on H , we initialize for each $\kappa_1 \in H$ a variable for $\hat{M}(\kappa_1, \beta_1)$ that is initially set to 0. Then, for each $\kappa_2 \in K$, we compute the term $u_H(\text{r}\hat{\text{ow}}(\kappa_2), \text{r}\hat{\text{ow}}(\kappa_2)) u_H(\beta_1, \hat{\text{c}}\hat{\text{ol}}(\kappa_2)) \hat{\text{v}}\hat{\text{al}}(\kappa_2)$ and add it to the variable for $\hat{M}(\text{r}\hat{\text{ow}}(\kappa_2), \beta_1)$. Since the values $(u_H(\kappa, \kappa))_{\kappa \in H}$ and $(u_H(\beta_1, \kappa))_{\kappa \in H}$ can be precomputed in $O(|H| \log |H|)$ operations, the foregoing strategy can be carried out in $O(|K| + |H| \log |H|)$ operations. \square

Lemma 3.5.7 (third round). *The field element σ_3 and the coefficients of the polynomials $g_3(X)$ and $h_3(X)$ can be found in $O(|K| \log |K|)$ field operations, when given the subsets H and K and the matrix M (in sparse form).*

Proof. First, we find the coefficients of the polynomial $f_3(X)$ from Eq. (3.5), which has degree at most $|K| - 1$. We traverse the matrix M to find the values of $\text{r}\hat{\text{ow}}(\kappa) = \text{row}(\kappa)$, $\hat{\text{c}}\hat{\text{ol}}(\kappa) = \text{col}(\kappa)$, and $\hat{\text{v}}\hat{\text{al}}(\kappa) = \text{val}(\kappa)$, for every $\kappa \in K$. Then, for each $\kappa \in K$, we calculate $f_3(\kappa) = \frac{v_H(\beta_2) v_H(\beta_1) \hat{\text{v}}\hat{\text{al}}(\kappa)}{(\beta_2 - \text{r}\hat{\text{ow}}(\kappa))(\beta_1 - \hat{\text{c}}\hat{\text{ol}}(\kappa))}$, and interpolate those $|K|$ values, in time $O(|K| \log |K|)$. Those values can also be summed, in time $O(|K|)$, to obtain σ_3 . Then $g_3(X)$ can be found easily, by subtracting $\sigma_3/|K|$ from $f_3(X)$ and dividing by X .

Next, the prover interpolates the values from M to find the three polynomials $\text{r}\hat{\text{ow}}$, $\hat{\text{c}}\hat{\text{ol}}$, and $\hat{\text{v}}\hat{\text{al}}$. Using fast polynomial multiplication, the prover calculates $v_H(\beta_2) v_H(\beta_1) \hat{\text{v}}\hat{\text{al}}(X) - (\beta_2 - \text{r}\hat{\text{ow}}(X))(\beta_1 - \hat{\text{c}}\hat{\text{ol}}(X)) f_3(X)$, and divides this polynomial by $v_K(X)$ to find $h_3(X)$. This too can be done in time $O(|K| \log |K|)$. \square

3.5.3 AHP for R1CS

We prove Theorem 3.5.2. In Section 3.5.3.1 we describe the indexer algorithm, in Section 3.5.3.2 we describe the prover and verifier algorithms, and in Section 3.5.3.3 we analyze the protocol. Fig. 3.5 summarizes the protocol.

The AHP for R1CS directly builds on the AHP for the lincheck problem, analogously to how in [BCRSVW19] the non-holographic protocol for R1CS builds on the non-holographic lincheck protocol. The three lincheck problems associated to the three matrices in the index are bundled together via random coefficients, while the entry-wise product is checked with a polynomial identity. Zero knowledge is achieved via bounded independence and random masks [BCGV16; BCRSVW19].

Consistency with the instance is achieved by having the verifier combine a low-degree extension of the instance and the low-degree extension of the (alleged) witness sent by the prover, in order to create a low-degree extension of the full assignment.

3.5.3.1 Offline phase: encoding the constraint system

The indexer **I** for RICS receives as input a field \mathbb{F} , two subsets H and K of \mathbb{F} , and three matrices $A, B, C \in \mathbb{F}^{H \times H}$ with $|K| \geq \max\{\|A\|, \|B\|, \|C\|\}$. The non-zero entries of A, B, C are assumed to be presented in some common canonical order. The output of **I** consists of the output of the lincheck indexer separately invoked on A, B, C . This produces nine univariate polynomials $\{\hat{r}ow_M, \hat{c}ol_M, \hat{v}al_M\}_{M \in \{A, B, C\}}$ over \mathbb{F} of degree less than $|K|$ that can be used to compute the low-degree extensions of A, B, C .

3.5.3.2 Online phase: proving and verifying satisfiability

The prover **P** for RICS receives as input a field \mathbb{F} , two subsets H and K of \mathbb{F} , three matrices $A, B, C \in \mathbb{F}^{H \times H}$ with $|K| \geq \max\{\|A\|, \|B\|, \|C\|\}$, input $x \in \mathbb{F}^{H[\leq |x|]}$, and witness $w \in \mathbb{F}^{H[> |x|]}$. The verifier **V** for RICS receives as input the field \mathbb{F} , two subsets H and K of \mathbb{F} , and input $x \in \mathbb{F}^{H[\leq |x|]}$; **V** also has oracle access to the polynomials $\{\hat{r}ow_M, \hat{c}ol_M, \hat{v}al_M\}_{M \in \{A, B, C\}}$ output by the indexer **I** invoked on appropriate inputs.

The protocol begins with the prover sending randomized encodings for (a certain shift of) the assignment and its linear combinations. Define $\hat{x}(X)$ to be the polynomial of degree less than $|x|$ that agrees with the instance x in $H[\leq |x|]$. Define the shifted witness $\bar{w}: H[> |x|] \rightarrow \mathbb{F}$ according to the equation

$$\forall \gamma, \bar{w}(\gamma) := \frac{w(\gamma) - \hat{x}(\gamma)}{v_{H[\leq |x|]}(\gamma)}.$$

The prover **P** sends to **V** a random $\hat{w}(X) \in \mathbb{F}^{< |w| + b}[X]$ that agrees with \bar{w} on $H[> |x|]$; **P** also sets $z := (x, w) \in \mathbb{F}^H$ to be the full assignment, computes the three linear combinations $z_A := Az$, $z_B := Bz$, and $z_C := Cz$, and sends to **V** random $\hat{z}_A(X), \hat{z}_B(X), \hat{z}_C(X) \in \mathbb{F}^{< |H| + b}[X]$ that agree with z_A, z_B, z_C on H . Note that the values of up to b locations in each of $\hat{w}(X), \hat{z}_A(X), \hat{z}_B(X), \hat{z}_C(X)$ reveal no information about the witness w , provided the locations are in $\mathbb{F} \setminus H$. Note also that $\hat{z}(X) := \hat{w}(X)v_{H[\leq |x|]}(X) + \hat{x}(X)$ agrees with z on H ; moreover, **V** can evaluate $\hat{z}(X)$ at any location γ with $O(|x|)$ operations by querying \hat{w} at γ and computing the expression $\hat{w}(\gamma)v_{H[\leq |x|]}(\gamma) + \hat{x}(\gamma)$ by using x .

The rest of the protocol is for **P** to convince **V** that $z_A \circ z_B = z_C$ and also that z_A, z_B, z_C are obtained as linear combinations from z .

In the same message as above, **P** also sends to **V** the polynomial $h_0(X)$ such that $\hat{z}_A(X)\hat{z}_B(X) - \hat{z}_C(X) = h_0(X)v_H(X)$. In addition, **P** sends to **V** a (fully) random $s(X) \in \mathbb{F}^{< 2|H| + b - 1}[X]$ and its sum $\sigma_1 := \sum_{\kappa \in H} s(\kappa)$ over H . This random polynomial will be used as a ‘‘mask’’ to make the univariate sumcheck zero knowledge.

Next, **V** samples random elements $\alpha, \eta_A, \eta_B, \eta_C \in \mathbb{F}$ and sends them to **P**. The element α is used to reduce lincheck problems to sumcheck, while the elements η_A, η_B, η_C are used to bundle the

three sumcheck problems into one. Indeed, \mathbf{P} is left to convince \mathbf{V} that the following univariate polynomial sums to σ_1 on H :

$$q_1(X) := s(X) + r(\alpha, X) \left(\sum_{M \in \{A, B, C\}} \eta_M \hat{z}_M(X) \right) - \left(\sum_{M \in \{A, B, C\}} \eta_M r_M(\alpha, X) \right) \hat{z}(X) \quad (3.6)$$

where $r_M(X, Y) := \sum_{\kappa \in H} r(X, \kappa) \hat{M}(\kappa, Y)$.

We now rely on the univariate sumcheck protocol: \mathbf{P} sends to \mathbf{V} the polynomials $g_1(X)$ and $h_1(X)$ such that $q_1(X) = h_1(X)v_H(X) + Xg_1(X)$. In order to check this polynomial identity, \mathbf{V} samples a random element $\beta_1 \in \mathbb{F} \setminus H$ with the intention of checking the identity at $X := \beta_1$. For the right-hand side, \mathbf{V} queries g_1 and h_1 at β_1 and then evaluates $h_1(\beta_1)v_H(\beta_1) + \beta_1g_1(\beta_1)$ in $O(\log |H|)$ operations. For the left-hand side, \mathbf{V} queries $s, \hat{z}_A, \hat{z}_B, \hat{z}_C, \hat{w}$ at β_1 and then needs to ask help from \mathbf{P} to evaluate $q_1(\beta_1)$. The reason is that the term $\eta_A r_A(\alpha, \beta_1) + \eta_B r_B(\alpha, \beta_1) + \eta_C r_C(\alpha, \beta_1)$ in general requires $\Omega(|H||K|)$ operations to compute.

Observe that

$$\begin{aligned} & \eta_A r_A(\alpha, \beta_1) + \eta_B r_B(\alpha, \beta_1) + \eta_C r_C(\alpha, \beta_1) \\ &= \eta_A \sum_{\kappa \in H} r(\alpha, \kappa) \hat{A}(\kappa, \beta_1) + \eta_B \sum_{\kappa \in H} r(\alpha, \kappa) \hat{B}(\kappa, \beta_1) + \eta_C \sum_{\kappa \in H} r(\alpha, \kappa) \hat{C}(\kappa, \beta_1) \\ &= \sum_{\kappa \in H} r(\alpha, \kappa) (\eta_A \hat{A}(\kappa, \beta_1) + \eta_B \hat{B}(\kappa, \beta_1) + \eta_C \hat{C}(\kappa, \beta_1)) . \end{aligned}$$

We define the polynomial

$$q_2(X) := r(\alpha, X) (\eta_A \hat{A}(X, \beta_1) + \eta_B \hat{B}(X, \beta_1) + \eta_C \hat{C}(X, \beta_1)) \quad (3.7)$$

and rely on the univariate sumcheck protocol again: \mathbf{V} sends β_1 to \mathbf{P} , and then \mathbf{P} replies with the sum $\sigma_2 := \sum_{\kappa \in H} q_2(\kappa)$ and the polynomials $g_2(X)$ and $h_2(X)$ such that $q_2(X) = h_2(X)v_H(X) + Xg_2(X) + \sigma_2/|H|$. In order to check this polynomial identity, \mathbf{V} samples a random element $\beta_2 \in \mathbb{F} \setminus H$ with the intention of checking the identity at $X := \beta_2$. (Excluding H is needed later in the protocol, as discussed below.) For the right-hand side, \mathbf{V} queries g_2 and h_2 at β_2 , and then evaluates $h_2(\beta_2)v_H(\beta_2) + \beta_2g_2(\beta_2) + \sigma_2/|H|$ in $O(\log |H|)$ operations. To evaluate the left-hand side, however, \mathbf{V} needs to ask help from \mathbf{P} . The reason is that while $r(\alpha, \beta_2)$ is easy to evaluate (it requires $O(\log |H|)$ operations), each term $\hat{M}(\beta_2, \beta_1)$ in general requires $\Omega(|K|)$ operations.

We thus rely on the univariate sumcheck protocol (yet) again: \mathbf{V} sends β_2 to \mathbf{P} , and then \mathbf{P} replies with the value $\sigma_3 := \eta_A \hat{A}(\beta_2, \beta_1) + \eta_B \hat{B}(\beta_2, \beta_1) + \eta_C \hat{C}(\beta_2, \beta_1)$, which the verifier much check. Observe that

$$\eta_A \hat{A}(\beta_2, \beta_1) + \eta_B \hat{B}(\beta_2, \beta_1) + \eta_C \hat{C}(\beta_2, \beta_1) = \sum_{\kappa \in K} \sum_{M \in \{A, B, C\}} \eta_M \frac{v_H(\beta_2)v_H(\beta_1)\hat{\text{val}}_M(\kappa)}{(\beta_2 - \hat{\text{row}}_M(\kappa))(\beta_1 - \hat{\text{col}}_M(\kappa))} .$$

Define $f_3(X)$ to be the (unique) polynomial of degree less than $|K|$ such that

$$\forall \kappa \in K, f_3(\kappa) = \sum_{M \in \{A, B, C\}} \eta_M \frac{v_H(\beta_2)v_H(\beta_1)\hat{\text{val}}_M(\kappa)}{(\beta_2 - \hat{\text{row}}_M(\kappa))(\beta_1 - \hat{\text{col}}_M(\kappa))} . \quad (3.8)$$

The prover computes the polynomials $g_3(X)$ and $h_3(X)$ such that

$$f_3(X) = Xg_3(X) + \sigma_3/|K| \quad \text{and} \quad a(X) - b(X)f_3(X) = h_3(X)v_K(X)$$

where

$$\begin{aligned} a(X) &:= \sum_{M \in \{A, B, C\}} \eta_M v_H(\beta_2) v_H(\beta_1) \hat{\text{val}}_M(X) \prod_{N \in \{A, B, C\} \setminus \{M\}} (\beta_2 - \text{row}_N(X)) (\beta_1 - \hat{\text{col}}_N(X)) , \\ b(X) &:= \prod_{M \in \{A, B, C\}} (\beta_2 - \text{row}_M(X)) (\beta_1 - \hat{\text{col}}_M(X)) . \end{aligned}$$

The first equation demonstrates that f_3 sums to σ_3 over K , and the second equation demonstrates that f_3 agrees with the correct addends over K . These two equations can be combined in a single equation that involves only $g_3(X)$ and $h_3(X)$:

$$a(X) - b(X)(Xg_3(X) + \sigma_3/|K|) = h_3(X)v_K(X) .$$

The prover thus only sends the two polynomials $g_3(X)$ and $h_3(X)$. In order to check this polynomial identity, \mathbf{V} samples a random element $\beta_3 \in \mathbb{F}$ with the intention of checking the identity at $X := \beta_3$. Then \mathbf{V} queries $g_3, h_3, \{\text{row}_M, \hat{\text{col}}_M, \hat{\text{val}}_M\}_{M \in \{A, B, C\}}$ at β_3 , and checks the identity in $O(\log |H|)$ operations.

If this third test passes then \mathbf{V} can use the value σ_3 in place of $\sum_{M \in \{A, B, C\}} \eta_M \hat{M}(\beta_2, \beta_1)$ to finish the second test. If this latter passes, \mathbf{V} can in turn use the value σ_2 in place of $\sum_{M \in \{A, B, C\}} \eta_M r_M(\alpha, \beta_1)$ to finish the first test.

3.5.3.3 Analysis

Soundness. We argue that the soundness error is at most

$$\max \left\{ \frac{2|H| + 2b}{|\mathbb{F}|}, \frac{3|K| + |H| + 1}{|\mathbb{F}|} + \frac{4|H| + b}{|\mathbb{F} \setminus H|} \right\} .$$

Suppose that for the given index $\mathfrak{i} = (\mathbb{F}, H, K, A, B, C)$ and instance $\mathfrak{x} = x$ there is no witness $\mathfrak{w} = w$ such that $Az \circ Bz = Cz$ for $z := (x, w)$ is a vector in \mathbb{F}^H . In particular, this holds for the witness w that is encoded in the polynomial $\hat{w}(X)$ sent by the prover. Let z_A, z_B, z_C be the vectors encoded in the polynomials $\hat{z}_A(X), \hat{z}_B(X), \hat{z}_C(X)$ sent by the prover, respectively. We know that either $z_A \circ z_B \neq z_C$ or one of z_A, z_B, z_C is not the correct linear combination of z . In the first case, the polynomial identity $\hat{z}_A \hat{z}_B - \hat{z}_C = h_0 v_H$ does not hold, so the probability that the verifier still accepts is at most $(2|H| + 2b)/|\mathbb{F}|$. In the second case, we rely on the randomized reduction to sumcheck, which fails with probability at most $(|H| + 1)/|\mathbb{F}|$. Next we have to account for the soundness errors of the three sequential sumchecks, which are bounded by the maximum degree in the respective polynomial equation divided by the size of the set from which the test point is chosen. Thus, the innermost sumcheck has soundness error at most $3|K|/|\mathbb{F}|$; the intermediate sumcheck has soundness error at most $2|H|/(|\mathbb{F} \setminus H|)$; the outermost sumcheck has soundness error at most $(2|H| + b)/(|\mathbb{F} \setminus H|)$.

Proof of knowledge. If the verifier accepts with probability greater than the soundness error argued above, then the prover's polynomial \hat{w} must encode a valid witness w .

Zero knowledge. We only sketch the intuition because a full proof (which includes constructing a simulator) is similar to the non-holographic setting described in [BCRSVW19]. The first message of the prover includes an encoding of the witness and encodings of its linear combinations. These encodings are protected against up to b queries outside of H because the encodings are b -wise independent over $\mathbb{F} \setminus H$. The first message also includes the polynomial $h_0(X)$, which in fact is b -wise independent everywhere on \mathbb{F} . Subsequent messages from the prover do not reveal any further information because they are produced for a sumcheck instance that is shifted by a random polynomial (the polynomial $s(X)$). This leads to (perfect) zero knowledge with query bound b and a query checker C that rejects any query to any of $\hat{w}(X), \hat{z}_A(X), \hat{z}_B(X), \hat{z}_C(X)$ that lies in H .

Efficiency. The indexer computes and outputs a constant number of polynomials of degree less than $|K|$, using time $O(|K| \log |K|)$. The subsequent protocol between the prover and verifier consists of 7 messages, with the prover moving first. The verifier makes a constant number of queries, evaluates \hat{x}, v_H, v_K at a constant number of locations, and then performs a constant number of field operations. Thus, verifier time is $O(|x| + \log |H| + \log |K|)$. The prover sends a constant number of polynomials whose degree is linearly related to $|H| + b$ or $|K|$. In the first round, the prover computes the linear combinations Az, Bz, Cz and interpolates them, which can be done in time $O(|K| + (|H| + b) \log(|H| + b))$; in the second round, the prover finds the coefficients of the polynomials $g_1(X)$ and $h_1(X)$ in time $O(|K| + (|H| + b) \log(|H| + b))$, similarly to the proof of Lemma 3.5.5; in the third round, the prover finds the sum σ_2 and the coefficients of $g_2(X)$ and $h_2(X)$ in time $O(|K| + |H| \log |H|)$, similarly to the proof of Lemma 3.5.6; and in the final round, the prover finds the sum σ_3 and the coefficients of $g_3(X)$ and $h_3(X)$ in time $O(|K| \log |K|)$, similarly to the proof of Lemma 3.5.7. Thus, prover time is $O((|H| + b) \log(|H| + b) + |K| \log |K|)$, which is $O((|K| + b) \log(|K| + b))$ since $|H| = O(|K|)$ (see Remark 3.5.3).

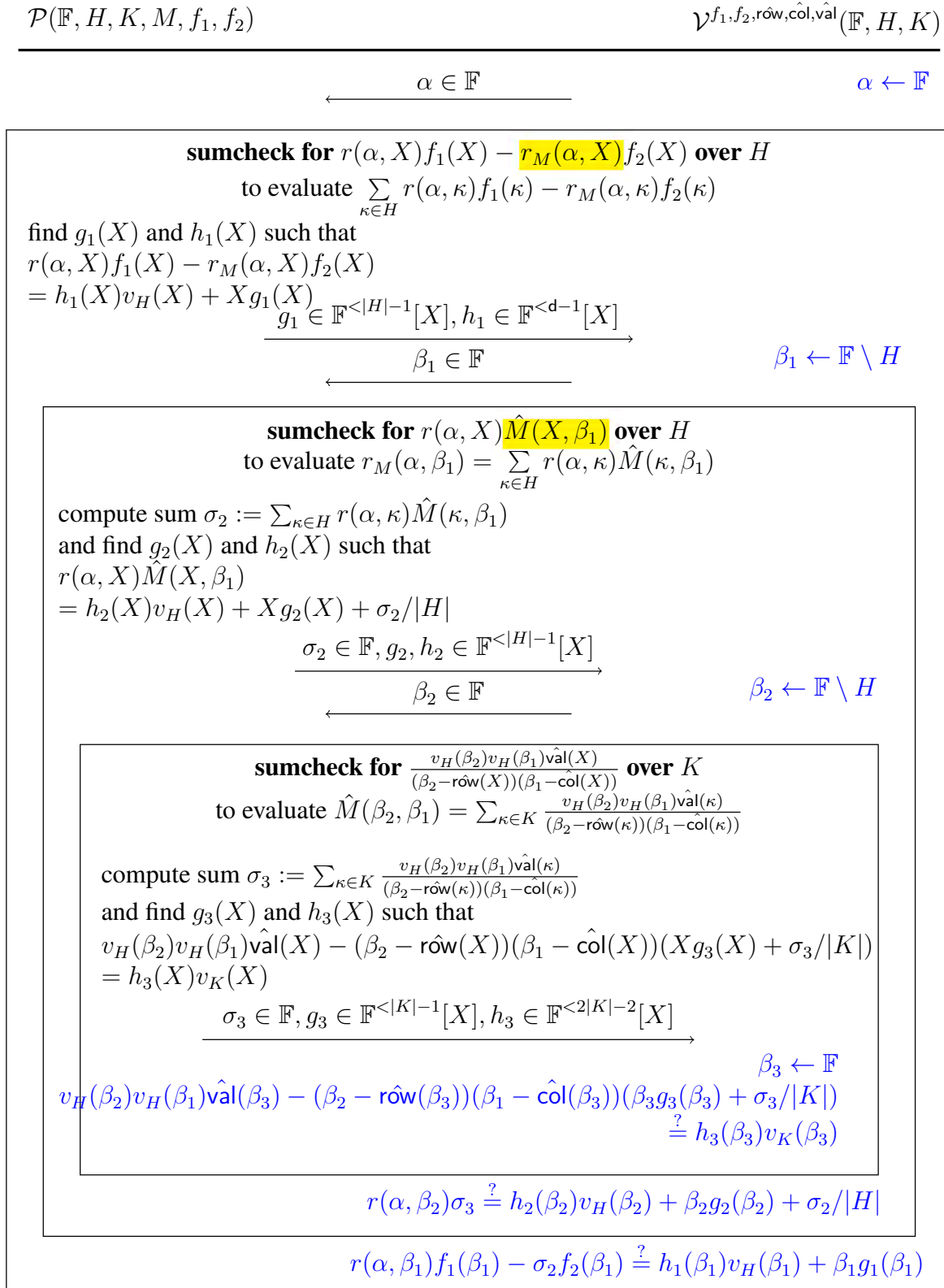


Figure 3.4: AHP for the lincheck problem.

$$\mathcal{P}(\mathbb{F}, H, K, A, B, C, x, w) \quad \mathcal{V}^{\text{row}\{A,B,C\}, \text{col}\{A,B,C\}, \text{val}\{A,B,C\}}(\mathbb{F}, H, K, x)$$

$z := (x, w)$ $z_A := Az$ $z_B := Bz$ $z_C := Cz$
 sample $\hat{w}(X) \in \mathbb{F}^{\langle |w|+b \rangle}[X]$ and $\hat{z}_A(X), \hat{z}_B(X), \hat{z}_C(X) \in \mathbb{F}^{\langle |H|+b \rangle}[X]$
 find $h_0(X)$ s.t. $\hat{z}_A(X)\hat{z}_B(X) - \hat{z}_C(X) = h_0(X)v_H(X)$
 sample $s(X) \in \mathbb{F}^{\langle 2|H|+b-1 \rangle}[X]$ and compute sum $\sigma_1 := \sum_{\kappa \in H} s(\kappa)$

$$\begin{array}{c} \xrightarrow{\quad} \sigma_1 \in \mathbb{F}, \hat{w} \in \mathbb{F}^{\langle |w|+b \rangle}[X], \hat{z}_A, \hat{z}_B, \hat{z}_C \in \mathbb{F}^{\langle |H|+b \rangle}[X], \\ \xrightarrow{\quad} h_0 \in \mathbb{F}^{\langle |H|+2b-1 \rangle}[X], s \in \mathbb{F}^{\langle 2|H|+b-1 \rangle}[X] \xrightarrow{\quad} \alpha, \eta_A, \eta_B, \eta_C \leftarrow \mathbb{F} \\ \xleftarrow{\quad} \alpha, \eta_A, \eta_B, \eta_C \in \mathbb{F} \xleftarrow{\quad} \end{array}$$

sumcheck for $s(X) + r(\alpha, X)(\sum_M \eta_M z_M(X)) - (\sum_M \eta_M r_M(\alpha, X))\hat{z}(X)$ **over** H
 find $g_1(X)$ and $h_1(X)$ such that
 $s(X) + r(\alpha, X)(\sum_M \eta_M z_M(X)) - (\sum_M \eta_M r_M(\alpha, X))\hat{z}(X)$
 $= h_1(X)v_H(X) + Xg_1(X) + \sigma_1/|H|$

$$\begin{array}{c} \xrightarrow{\quad} g_1 \in \mathbb{F}^{\langle |H|-1 \rangle}[X], h_1 \in \mathbb{F}^{\langle |H|+b-1 \rangle}[X] \xrightarrow{\quad} \beta_1 \leftarrow \mathbb{F} \setminus H \\ \xleftarrow{\quad} \beta_1 \in \mathbb{F} \xleftarrow{\quad} \end{array}$$

sumcheck for $r(\alpha, X)(\eta_A \hat{A}(X, \beta_1) + \eta_B \hat{B}(X, \beta_1) + \eta_C \hat{C}(X, \beta_1))$ **over** H

$\sigma_2 := \sum_{\kappa \in H} r(\alpha, \kappa) \sum_{M \in \{A,B,C\}} \eta_M \hat{M}(\kappa, \beta_1)$
 and find $g_2(X)$ and $h_2(X)$ such that
 $r(\alpha, X) \sum_{M \in \{A,B,C\}} \eta_M \hat{M}(X, \beta_1)$
 $= h_2(X)v_H(X) + Xg_2(X) + \sigma_2/|H|$

$$\begin{array}{c} \xrightarrow{\quad} \sigma_2 \in \mathbb{F}, g_2, h_2 \in \mathbb{F}^{\langle |H|-1 \rangle}[X] \xrightarrow{\quad} \beta_2 \leftarrow \mathbb{F} \setminus H \\ \xleftarrow{\quad} \beta_2 \in \mathbb{F} \xleftarrow{\quad} \end{array}$$

sumcheck for $\sum_{M \in \{A,B,C\}} \eta_M \frac{v_H(\beta_2)v_H(\beta_1)\hat{\text{val}}_M(X)}{(\beta_2 - \text{row}_M(X))(\beta_1 - \text{col}_M(X))}$ **over** K

to evaluate $\eta_A \hat{A}(\beta_2, \beta_1) + \eta_B \hat{B}(\beta_2, \beta_1) + \eta_C \hat{C}(\beta_2, \beta_1)$

$\sigma_3 := \sum_{\kappa \in K} \sum_{M \in \{A,B,C\}} \eta_M \frac{v_H(\beta_2)v_H(\beta_1)\hat{\text{val}}_M(\kappa)}{(\beta_2 - \text{row}_M(\kappa))(\beta_1 - \text{col}_M(\kappa))}$

and find $g_3(X)$ and $h_3(X)$ such that
 $h_3(X)v_K(X) = a(X) - b(X)(Xg_3(X) + \sigma_3/|K|)$

$$\begin{array}{c} \xrightarrow{\quad} \sigma_3 \in \mathbb{F}, g_3 \in \mathbb{F}^{\langle |K|-1 \rangle}[X], h_3 \in \mathbb{F}^{\langle 6|K|-6 \rangle}[X] \xrightarrow{\quad} \beta_3 \leftarrow \mathbb{F} \\ \xrightarrow{\quad} h_3(\beta_3)v_K(\beta_3) \stackrel{?}{=} a(\beta_3) - b(\beta_3)(\beta_3 g_3(\beta_3) + \sigma_3/|K|) \end{array}$$

The polynomials $a(X), b(X)$ are defined as follows:

$$\begin{aligned} a(X) &:= \sum_{M \in \{A,B,C\}} \eta_M v_H(\beta_2)v_H(\beta_1)\hat{\text{val}}_M(X) \prod_{N \in \{A,B,C\} \setminus \{M\}} (\beta_2 - \text{row}_N(X))(\beta_1 - \hat{\text{col}}_N(X)) \\ b(X) &:= \prod_{M \in \{A,B,C\}} (\beta_2 - \text{row}_M(X))(\beta_1 - \hat{\text{col}}_M(X)) \end{aligned}$$

$$r(\alpha, \beta_2)\sigma_3 \stackrel{?}{=} h_2(\beta_2)v_H(\beta_2) + \beta_2 g_2(\beta_2) + \sigma_2/|H|$$

$$\begin{aligned} &s(\beta_1) + r(\alpha, \beta_1)(\sum_M \eta_M z_M(\beta_1)) - \sigma_2 \hat{z}(\beta_1) \\ &\stackrel{?}{=} h_1(\beta_1)v_H(\beta_1) + \beta_1 g_1(\beta_1) + \sigma_1/|H| \end{aligned}$$

$$\hat{z}_A(\beta_1)\hat{z}_B(\beta_1) - \hat{z}_C(\beta_1) \stackrel{?}{=} h_0(\beta_1)v_H(\beta_1)$$

Figure 3.5: AHP for R1CS.

3.6 Polynomial commitment schemes with extractability

We use *polynomial commitment schemes*, a class of commitment schemes specialized to work with univariate polynomials. This notion was introduced by Kate, Zaverucha, and Goldberg [KZG10], who gave an elegant construction using bilinear groups. The security properties in [KZG10], however, do not appear sufficient for standalone use (nor for use in this paper). This limitation was recently noted in [MBKM19], which relies on a different construction for which certain properties are proved in the algebraic group model [FKL18]. However, [MBKM19] stops short of formulating a cryptographic primitive that captures the features of the construction.

In this section we propose definitions for polynomial commitment schemes that incorporate the functionality and security that we believe to be a bare minimum for standalone use. (In particular, in Section 3.8 we generically rely on these definitions to build preprocessing arguments with universal SRS.) We also describe a “knowledge” variant of the construction in [KZG10], which we prove secure under knowledge of exponent assumptions. To learn more about the insights motivating our definitions, we refer the reader back to Section 3.2.2.

The rest of this section is organized as follows. In Section 3.6.1 we present the definitions that we propose. In Section 3.6.2 we provide a theorem statement for constructions that realize the definitions, and then sketch these constructions. We formal descriptions of the constructions are in the full version[CHMMVW20].

3.6.1 Definition

A polynomial commitment scheme over a field family \mathcal{F} is a tuple of algorithms $\text{PC} = (\text{Setup}, \text{Trim}, \text{Commit}, \text{Open}, \text{Check})$ with the following syntax.

- $\text{PC.Setup}(1^\lambda, D) \rightarrow \text{pp}$. On input a security parameter λ (in unary), and a maximum degree bound $D \in \mathbb{N}$, PC.Setup samples public parameters pp . The parameters contain the description of a finite field $\mathbb{F} \in \mathcal{F}$.
- $\text{PC.Trim}^{\text{pp}}(1^\lambda, \mathbf{d}) \rightarrow (\text{ck}, \text{rk})$. Given oracle access^{pp} to public parameters pp , and on input a security parameter λ (in unary), and degree bounds \mathbf{d} , PC.Trim deterministically computes a key pair (ck, rk) that is specialized to \mathbf{d} .
- $\text{PC.Commit}(\text{ck}, \mathbf{p}, \mathbf{d}; \omega) \rightarrow \mathbf{c}$. On input ck , univariate polynomials $\mathbf{p} = [p_i]_{i=1}^n$ over the field \mathbb{F} , and degree bounds $\mathbf{d} = [d_i]_{i=1}^n$ with $\deg(p_i) \leq d_i \leq D$, PC.Commit outputs commitments $\mathbf{c} = [c_i]_{i=1}^n$ to the polynomials $\mathbf{p} = [p_i]_{i=1}^n$. The randomness $\omega = [\omega_i]_{i=1}^n$ is used if the commitments $\mathbf{c} = [c_i]_{i=1}^n$ are hiding.
- $\text{PC.Open}(\text{ck}, \mathbf{p}, \mathbf{d}, Q, \xi; \omega) \rightarrow \pi$. On input ck , univariate polynomials $\mathbf{p} = [p_i]_{i=1}^n$, degree bounds $\mathbf{d} = [d_i]_{i=1}^n$, a query set Q consisting of tuples $(i, z) \in [n] \times \mathbb{F}$, and opening challenge ξ , PC.Open outputs an evaluation proof π . The randomness ω must equal the one previously used in PC.Commit .

- $\text{PC.Check}(\text{rk}, \mathbf{c}, \mathbf{d}, Q, \mathbf{v}, \pi, \xi) \in \{0, 1\}$. On input rk , commitments $\mathbf{c} = [c_i]_{i=1}^n$, degree bounds $\mathbf{d} = [d_i]_{i=1}^n$, query set Q consisting of tuples $(i, z) \in [n] \times \mathbb{F}$, alleged evaluations $\mathbf{v} = (v_{(i,z)})_{(i,z) \in Q}$, evaluation proof π , and opening challenge ξ , PC.Check outputs 1 if π attests that, for every $(i, z) \in Q$, the polynomial p_i committed in c_i has degree at most d_i and evaluates to $v_{(i,z)}$ at z .

A polynomial commitment scheme PC must satisfy the completeness and extractability properties defined below. We also consider two additional properties, efficiency and hiding, also defined below. To simplify notation, we denote by $\deg(\mathbf{p})$ the degrees $[\deg(p_i)]_{i=1}^n$ of polynomials $\mathbf{p} = [p_i]_{i=1}^n$, and denote by $\mathbf{p}(Q)$ the evaluations $(p_i(z))_{(i,z) \in Q}$ of the polynomials $\mathbf{p} = [p_i]_{i=1}^n$ at a query set $Q \subseteq [n] \times \mathbb{F}$.

Definition 3.6.1 (Completeness). For every maximum degree bound $D \in \mathbb{N}$ and efficient adversary \mathcal{A} ,

$$\Pr \left[\begin{array}{c} \deg(\mathbf{p}) \leq \mathbf{d} \leq D \\ \Downarrow \\ \text{PC.Check}(\text{rk}, \mathbf{c}, \mathbf{d}, Q, \mathbf{v}, \pi, \xi) = 1 \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D) \\ (\mathbf{p}, \mathbf{d}, Q, \xi, \omega) \leftarrow \mathcal{A}(\text{pp}) \\ (\text{ck}, \text{rk}) \leftarrow \text{PC.Trim}^{\text{PP}}(1^\lambda, \mathbf{d}) \\ \mathbf{c} \leftarrow \text{PC.Commit}(\text{ck}, \mathbf{p}, \mathbf{d}; \omega) \\ \mathbf{v} \leftarrow \mathbf{p}(Q) \\ \pi \leftarrow \text{PC.Open}(\text{ck}, \mathbf{p}, \mathbf{d}, Q, \xi; \omega) \end{array} \right] = 1 .$$

Definition 3.6.2 (Extractability). For every maximum degree bound $D \in \mathbb{N}$ and efficient adversary \mathcal{A} there exists an efficient extractor \mathcal{E} such that for every round bound $r \in \mathbb{N}$, efficient public-coin challenger \mathcal{C} (each of its messages is a uniformly random string of prescribed length, or an empty string), efficient query sampler \mathcal{Q} , and efficient adversary $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ the probability below is negligibly close to 1 (as a function of λ):

$$\Pr \left[\begin{array}{c} \text{PC.Check}(\text{rk}, \mathbf{c}, \mathbf{d}, Q, \mathbf{v}, \pi, \xi) = 1 \\ \Downarrow \\ \deg(\mathbf{p}) \leq \mathbf{d} \leq D \text{ and } \mathbf{v} = \mathbf{p}(Q) \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D) \\ \hline \text{For } i = 1, \dots, r: \\ \rho_i \leftarrow \mathcal{C}(\text{pp}, i) \\ (\mathbf{c}_i, \mathbf{d}_i) \leftarrow \mathcal{A}(\text{pp}, [\rho_j]_{j=1}^i) \\ \mathbf{p}_i \leftarrow \mathcal{E}(\text{pp}, [\rho_j]_{j=1}^i) \\ \hline Q \leftarrow \mathcal{Q}(\text{pp}, [\rho_j]_{j=1}^r) \\ (\mathbf{v}, \text{st}) \leftarrow \mathcal{B}_1(\text{pp}, [\rho_j]_{j=1}^r, Q) \\ \text{Sample opening challenge } \xi \\ \pi \leftarrow \mathcal{B}_2(\text{st}, \xi) \\ \text{Set } [c_i]_{i=1}^n := [c_i]_{i=1}^r, [p_i]_{i=1}^n := [p_i]_{i=1}^r, [d_i]_{i=1}^n := [d_i]_{i=1}^r \\ (\text{ck}, \text{rk}) \leftarrow \text{PC.Trim}^{\text{PP}}(1^\lambda, [d_i]_{i=1}^n) \\ \text{Define the set of queried polynomials } T := \{(i, z) \in Q\} \\ \text{Set } \mathbf{c} := [c_i]_{i \in T}, \mathbf{p} := [p_i]_{i \in T}, \mathbf{d} := [d_i]_{i \in T} \end{array} \right]$$

(The above definition captures the case where $\mathcal{A}, \mathcal{Q}, \mathcal{B}$ share the same random string to win the game.)

Definition 3.6.3 (Efficiency). We say that a polynomial commitment scheme PC is:

- **degree-efficient** if the time to run PC.Commit and PC.Open is proportional to the maximum degree $\max(\mathbf{d})$ (as opposed to the maximum supported degree D). In particular this implies that $|\text{ck}| = O_\lambda(\max(\mathbf{d}))$.
- **succinct** if the size of commitments, the size of evaluation proofs, and the time to check an opening are all independent of the degree of the committed polynomials. That is, $|\mathbf{c}| = n \cdot \text{poly}(\lambda)$, $|\pi| = |Q| \cdot \text{poly}(\lambda)$, $|\text{rk}| = O_\lambda(n)$, and $\text{time}(\text{Check}) = (n + |Q|) \cdot \text{poly}(\lambda)$.

Definition 3.6.4 (Hiding). There exists a polynomial-time simulator $\mathcal{S} = (\text{Setup}, \text{Commit}, \text{Open})$ such that, for every maximum degree bound $D \in \mathbb{N}$, and efficient adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$, the probability that $b = 1$ in the following two experiments is identical:

Real($1^\lambda, D, \mathcal{A}$):

1. $\text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D)$.
2. Letting $\mathbf{c}_0 := \perp$, for $i = 1, \dots, r$:
 - a) $(\mathbf{p}_i, \mathbf{d}_i, h_i) \leftarrow \mathcal{A}_1(\text{pp}, \mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_{i-1})$.
 - b) $(\text{ck}_i, \text{rk}_i) \leftarrow \text{PC.Trim}^{\text{PP}}(1^\lambda, \mathbf{d}_i)$.
 - c) If $h_i = 0$: sample commitment randomness ω_i .
 - d) If $h_i = 1$: set randomness ω_i to \perp .
 - e) $\mathbf{c}_i \leftarrow \text{PC.Commit}(\text{ck}_i, \mathbf{p}_i, \mathbf{d}_i; \omega_i)$.
3. $\mathbf{c} := [\mathbf{c}_i]_{i=1}^r$, $\mathbf{p} := [\mathbf{p}_i]_{i=1}^r$, $\mathbf{d} := [\mathbf{d}_i]_{i=1}^r$, $\omega := [\omega_i]_{i=1}^r$.
4. $(\text{ck}, \text{rk}) \leftarrow \text{PC.Trim}^{\text{PP}}(1^\lambda, \mathbf{d})$.
5. $([Q_j]_{j=1}^\tau, [\xi_j]_{j=1}^\tau, \text{st}) \leftarrow \mathcal{A}_2(\text{pp}, \mathbf{c})$.
6. For $j \in [\tau]$:
$$\pi_j \leftarrow \text{PC.Open}(\text{ck}, \mathbf{p}, \mathbf{d}, Q_j, \xi_j; \omega).$$
7. $b \leftarrow \mathcal{A}_3(\text{st}, [\pi]_{j=1}^\tau)$.

Ideal($1^\lambda, D, \mathcal{A}$):

1. $(\text{pp}, \text{trap}) \leftarrow \mathcal{S.Setup}(1^\lambda, D)$.
2. Letting $\mathbf{c}_0 := \perp$, for $i = 1, \dots, r$:
 - a) $(\mathbf{p}_i, \mathbf{d}_i, h_i) \leftarrow \mathcal{A}_1(\text{pp}, \mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_{i-1})$.
 - b) $(\text{ck}_i, \text{rk}_i) \leftarrow \text{PC.Trim}^{\text{PP}}(1^\lambda, \mathbf{d}_i)$.
 - c) If $h_i = 0$: sample randomness ω_i and compute simulated commitments $\mathbf{c}_i \leftarrow \mathcal{S.Commit}(\text{trap}, \mathbf{d}_i; \omega_i)$.
 - d) If $h_i = 1$: set $\omega_i := \perp$ and compute (real) commitments $\mathbf{c}_i \leftarrow \text{PC.Commit}(\text{ck}_i, \mathbf{p}_i, \mathbf{d}_i; \omega_i)$.
3. $\mathbf{c} := [\mathbf{c}_i]_{i=1}^r$, $\mathbf{p} := [\mathbf{p}_i]_{i=1}^r$, $\mathbf{d} := [\mathbf{d}_i]_{i=1}^r$, $\omega := [\omega_i]_{i=1}^r$.
4. $(\text{ck}, \text{rk}) \leftarrow \text{PC.Trim}^{\text{PP}}(1^\lambda, \mathbf{d})$.
5. $([Q_j]_{j=1}^\tau, [\xi_j]_{j=1}^\tau, \text{st}) \leftarrow \mathcal{A}_2(\text{pp}, \mathbf{c})$.
6. Zero out hidden polynomials: $\mathbf{p}' := [\mathbf{h}_i \mathbf{p}_i]_{i=1}^r$.
7. For $j \in [\tau]$:
$$\pi_j \leftarrow \mathcal{S.Open}(\text{trap}, \mathbf{p}', \mathbf{p}(Q_j), \mathbf{d}, Q_j, \xi_j; \omega).$$
8. $b \leftarrow \mathcal{A}_3(\text{st}, [\pi]_{j=1}^\tau)$.

(We implicitly assume that \mathcal{A}_1 outputs $\text{poly}(\lambda)$ polynomials overall and that \mathcal{A}_2 outputs $\text{poly}(\lambda)$ query sets each consisting of $\text{poly}(\lambda)$ points, ensuring that $\text{PC}_s.\text{Commit}$, $\text{PC}_s.\text{Open}$, $\mathcal{S}.\text{Commit}$, $\mathcal{S}.\text{Open}$ are efficient.)

3.6.2 Construction

The theorem below states the properties of our constructions. For simplicity, our construction are restricted to work with respect to “admissible” query samplers.

Definition 3.6.5. A query sampler \mathcal{Q} is **admissible** if it outputs query sets such that each polynomial to be evaluated is evaluated at a point sampled uniformly at random from a super-polynomially large subset of the field, and possibly also at other points that can be arbitrarily chosen.

Theorem 3.6.6. *There exist succinct polynomial commitment schemes that: (a) achieve extractability against admissible query samplers under knowledge assumptions, or in the algebraic group model; (b) achieve hiding; and (c) have an updatable SRS. See Table 3.2 for the efficiency of these schemes under these assumptions.*

We note that the restriction to admissible query samplers is minor because one can transform an arbitrary query sampler \mathcal{Q} into an admissible query sampler \mathcal{Q}' as follows: \mathcal{Q}' invokes \mathcal{Q} to obtain a query set Q , and then outputs $Q' := Q \cup \{(i, t)\}_{i \in [n]}$, where $t \in \mathbb{F}$ is a random field element and n is the number of polynomials. This transformation yields evaluation proofs that are twice as large, a minor cost. That said, this transformation is often not even needed because “natural” query samplers are often already admissible, as is the case for those that we consider in this paper.

assumption	hiding	communication complexity				time complexity			
		ck	rk	$ \{c_i\}_{i=1}^n $	$ \pi $	Setup	Commit	Open	Check
PKE	no	$2d \mathbb{G}_1$	$2 \mathbb{G}_2$	$4n \mathbb{G}_1$	$1 \mathbb{G}_1$	$2 \text{f-MSM}(D)$	$4n \text{v-MSM}(d)$	$1 \text{v-MSM}(d)$	2 $\text{v-MSM}(2n)$ $+ 4 \text{ pairings}$
dPKE	yes	$4d \mathbb{G}_1$	$2 \mathbb{G}_2$	$4n \mathbb{G}_1$	$\frac{1}{1} \mathbb{G}_1 + \frac{1}{1} \mathbb{F}_q$	$4 \text{f-MSM}(D)$	$8n \text{v-MSM}(d)$	$2 \text{v-MSM}(d)$	2 $\text{v-MSM}(2n)$ $+ 4 \text{ pairings}$
AGM	no	$d \mathbb{G}_1$	$1 \mathbb{G}_2$	$2n \mathbb{G}_1$	$1 \mathbb{G}_1$	$1 \text{f-MSM}(D)$	$2n \text{v-MSM}(d)$	$1 \text{v-MSM}(d)$	1 $\text{v-MSM}(2n)$ $+ 2 \text{ pairings}$
AGM	yes	$2d \mathbb{G}_1$	$1 \mathbb{G}_2$	$2n \mathbb{G}_1$	$\frac{1}{1} \mathbb{G}_1 + \frac{1}{1} \mathbb{F}_q$	$2 \text{f-MSM}(D)$	$4n \text{v-MSM}(d)$	$2 \text{v-MSM}(d)$	1 $\text{v-MSM}(2n)$ $+ 2 \text{ pairings}$

Table 3.2: Efficiency of our polynomial commitment schemes. Here $\text{f-MSM}(m)$ and $\text{v-MSM}(m)$ denote fixed-base and variable-base multi-scalar multiplications (MSM) each of size m , respectively. All MSMs are carried out over \mathbb{G}_1 . For simplicity we assume above that the query set evaluates each polynomial at the same point. If there are multiple points in the set, then proof size and time for checking proofs scales linearly with the number of points. Furthermore, we assume above that the n committed polynomials all have degree d .

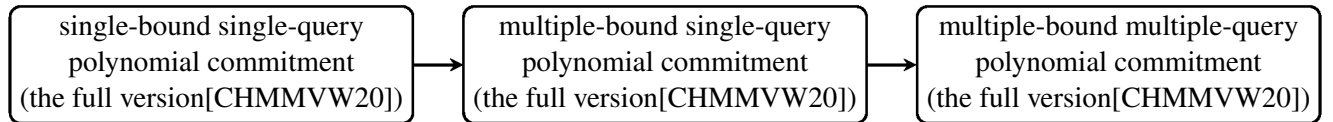


Figure 3.6: Our approach to construct polynomial commitment schemes.

The constructions behind Theorem 3.6.6 are achieved in three steps, as summarized in Fig. 3.6. The rest of this section is organized in three parts sketching these three steps respectively: (1) opening multiple polynomials with the same degree bound at a single point; (2) opening multiple polynomials

with multiple degree bounds at a single point; (3) opening multiple polynomials with multiple degree bounds at multiple points. Detailed descriptions, along with security proofs, are provided in the corresponding appendices.

3.6.2.1 Single-bound single-query (see the full version[CHMMVW20] for details)

We begin by discussing the case of opening multiple polynomials with the same degree bound at a single point. We describe a non-hiding construction based on $\text{PolyCommit}_{\text{DL}}$ from [KZG10] (see Section 3.2.5) and a hiding construction based on $\text{PolyCommit}_{\text{ped}}$ from [KZG10], using “knowledge commitments” [Gro10] or the algebraic group model [FKL18] to achieve extractability for a single degree bound D chosen at setup.

Extractability with knowledge commitments. While $\text{PolyCommit}_{\text{DL}}$ guarantees correctness of evaluations, it does not ensure extractability: there is no guarantee that a commitment actually “contains” a polynomial. To achieve extraction, we modify the construction in such a way that the PKE assumption [Gro10] forces the sender to demonstrate knowledge of the committed polynomial. In more detail, we extend ck to encode powers of β with respect to a different generator αG : $\text{ck} := \{(G, \beta G, \dots, \beta^D G), (G, \alpha\beta G, \dots, \alpha\beta^D G)\} \in \mathbb{G}_1^{2(D+1)}$. (Note that this modification does not affect the updatability of the SRS.) To commit to a polynomial p of degree at most D , the sender now provides a “knowledge commitment”: $c := (U, V) := (p(\beta)G, \alpha p(\beta)G)$. Proving correctness of evaluations proceeds unchanged, while verification additionally requires checking extractability of the commitment by checking the pairing equation $e(U, \alpha H) = e(V, H)$.

Extractability in the AGM. Knowledge commitments require, unfortunately, two group elements instead of one. Alternatively, we could keep each commitment as one group element, by relying on the algebraic group model (AGM) [FKL18]. Informally, whenever an adversary in the AGM outputs a group element G_n , it is required to additionally output scalar coefficients a_1, \dots, a_{n-1} which “explain” G_n as a linear combination of any group elements G_1, \dots, G_{n-1} that it has seen previously. In our setting, this means that whenever the adversarial sender outputs a group element c representing a commitment, it must additionally output scalar coefficients that explain c in terms of the group elements in ck . An extractor can use these coefficients to reconstruct the underlying polynomial, thus achieving extractability.

Efficiently opening multiple polynomials at the same point. To enable the sender to simultaneously commit to multiple polynomials $[p_i]_{i=1}^n$ of degree at most D and then open these at the same point z , we rely on the fact that the commitments for both variants above are *additively homomorphic*. That is, if commitments $[c_i]_{i=1}^n$ commit to $[p_i]_{i=1}^n$, then $\sum_{i=1}^n c_i$ commits to $\sum_{i=1}^n p_i$ (where $c_1 + c_2$ is defined as $(U_1 + U_2, V_1 + V_2)$).

We take advantage of this by simultaneously verifying the evaluations of each polynomial $p_i \in [p_i]_{i=1}^n$ as follows. Before generating a proof of evaluation for $[p_i]_{i=1}^n$, the sender requests from the receiver a random field element ξ . The sender then uses this to take a random linear combination of the polynomials: $p := \sum_{i=1}^n \xi^i p_i$, and generates a single evaluation proof π for this derived polynomial p .

To verify π , the receiver uses the additive homomorphism of the input commitments to derive the linear combination $c = \sum_{i=1}^n \xi^i c_i$ induced by ξ . It does the same with the claimed evaluations,

thus deriving the evaluation $v = \sum_{i=1}^n \xi^i v_i$. Finally, it checks that the pairing equations are satisfied for c , π , and v .

This works because if the sender is honest, then c is a commitment to $p := \sum_{i=1}^n \xi^i p_i$, and π is a proof of evaluation of p at z . On the other hand, if the sender is dishonest, then with high probability over the choice of ξ , c is not a commitment to p , and the pairing equations would fail.

Hiding. To additionally achieve hiding, we follow the above blueprint, replacing $\text{PolyCommit}_{\text{DL}}$ with the hiding scheme $\text{PolyCommit}_{\text{ped}}$. Extraction now follows from an assumption related to PKE called dPKE (see the full version[CHMMVW20] for details). Our constructions in the full version[CHMMVW20] in fact use both variants to provide optional hiding on a per-polynomial basis. Further, the near-identical form of the commitment variants makes it possible to open a combination of hiding and non-hiding polynomials at the same point.

3.6.2.2 Multiple-bound single-query (see the full version[CHMMVW20] for details)

Thus far, we have focused on commitment schemes for polynomials of degree D where the cost of committing and providing evaluation proofs grows as $\Omega(D)$. However, when working with polynomials of degree $d < D$, we would like to pay a cost that instead grows as $O(d)$. Furthermore, the foregoing schemes only guarantee that committed polynomials have degree at most D , whereas in many cases it is desirable to enforce more specific degree bounds. Below we show how to adapt the foregoing construction to achieve these desirable properties.

To achieve extractability with respect to a different degree bound d_i for each polynomial p_i , we require the sender to commit not only to each p_i , but also to “shifted polynomials” $p'_i(X) := X^{D-d_i} p_i(X)$. During PC.Open, one could then produce an evaluation proofs that attests that if p_i evaluates to v_i at z then p'_i evaluates to $z^{D-d_i} v_i$ at z .

The receiver checks that the commitment for each p'_i corresponds to an evaluation $z^{D-d_i} v_i$ so that, if z is sampled from a super-polynomial subset of \mathbb{F}_q , the probability that $\deg(p_i) \neq d_i$ is negligible. This trick is similar to the one used in [BS08; BCRSVW19] to enforce derive low-degree tests for specific degree bounds.

However, while sound, this approach is inefficient in our setting: the witness polynomial for p'_i has $\Omega(D)$ non-zero coefficients (instead of $O(d_i)$), and so constructing an evaluation proof for it requires $\Omega(D)$ scalar multiplications (instead of $O(d_i)$). To work around this, we instead produce a proof that the related polynomial $p_i^*(X) := p'_i(X) - p_i(z)X^{D-d_i}$ evaluates to 0 at z . As we show in the full version[CHMMVW20], the witness polynomial for this claim has $O(d_i)$ non-zero coefficients, and so constructing the evaluation proof can be done in $O(d_i)$ scalar multiplications. Completeness is preserved because the receiver can check the correct evaluation of p_i^* by subtracting $p_i(z)(\beta^{D-d_i} \mathbb{G})$ from the commitment to the shifted polynomial p'_i , thereby obtaining a commitment to p_i^* , while security is preserved because $p'_i(z) = z^{D-d_i} v_i \iff p_i^*(z) = 0$.

Note that to commit to the shifted polynomial p'_i , the committer must obtain $\{\beta^{D-d_i} \mathbb{G}, \dots, \beta^D \mathbb{G}\}$ from ck, while to adjust the shifted commitment, the receiver must obtain $\beta^{D-d_i} \mathbb{G}$ from rk. Thus PC.Trim must produce (ck, rk) containing these group elements.

3.6.2.3 Multiple-bound multiple-query (see the full version[CHMMVW20] for details)

Assume that we have *any* construction that achieves extractability with respect to individual degree bounds, and evaluation of multiple polynomials $\mathbf{p} = [p_i]_{i=1}^n$ at the same point z .

We extend this construction to support query sets Q consisting of multiple evaluation points (as required in Section 3.6.1). If there are k distinct points $[z_i]_{i=1}^k$ in the query set Q , the sender partitions the polynomials \mathbf{p} into different (possibly overlapping) groups $[p_i]_{i=1}^k$ such that every polynomial in p_i is to be evaluated at the same point z_i . It then runs PC.Open on each p_i , and outputs the resulting list of k evaluation proofs.

We note that [KZG10] describe how one can enable the sender to produce a *single* evaluation proof attesting to the correct evaluation of the same polynomial at multiple points. While we could use this to enable batch evaluation of \mathbf{p} at multiple points, we avoid doing so for efficiency reasons in our setting.

3.7 Preprocessing arguments with universal SRS

An *argument system* [BCC88] is an interactive proof where the soundness property is only required to hold against all efficient adversaries, as opposed to all (possibly computationally unbounded) adversaries. In this paper we consider argument systems for indexed relations (see Section 3.3.1) that have the following features.

- Security is proved, under cryptographic assumptions, in a model where all parties have access to a “long” structured reference string (SRS) that is *universal*. (In fact, the SRS in our constructions will also be *updatable* [GKMMM18] but for simplicity we do not formally discuss this property; see Remark 3.7.1.)
- Anyone can publicly *preprocess* a given index (e.g., a circuit) in an offline phase, in order to avoid incurring costs related to the index in (any number of) subsequent online phases that check different instances.

We refer to argument systems with the above properties as **preprocessing arguments with universal SRS**. All interactive constructions in this paper are public-coin zero-knowledge succinct arguments of knowledge so that, via the Fiat–Shamir transformation [FS86], we obtain their non-interactive analogues: **preprocessing zkSNARKs with universal SRS**. See Section 3.9 for an efficient construction of such a zkSNARK.

A preprocessing argument with universal SRS is a tuple of four algorithms $\text{ARG} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$. The probabilistic polynomial-time generator \mathcal{G} , given a size bound $N \in \mathbb{N}$, samples an SRS srs that supports indices of size up to N . The indexer \mathcal{I} is a *deterministic* polynomial-time algorithm that, given oracle access to srs and an index \mathfrak{i} of size at most N , outputs an index proving key ipk used by the prover \mathcal{P} in place of \mathfrak{i} and an index verification key ivk used by the verifier \mathcal{V} in place of \mathfrak{i} ; the verifier \mathcal{V} will be able to use ivk for significant efficiency gains compared to just using \mathfrak{i} directly. The prover \mathcal{P} and verifier \mathcal{V} are probabilistic polynomial-time interactive algorithms.

Formally, $\text{ARG} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ is a preprocessing argument with universal SRS for an indexed relation \mathcal{R} if the following properties hold.

- **Completeness.** For all size bounds $N \in \mathbb{N}$ and efficient \mathcal{A} ,

$$\Pr \left[\begin{array}{c|c} (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \notin \mathcal{R}_N & \text{srs} \leftarrow \mathcal{G}(1^\lambda, N) \\ \vee & (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \leftarrow \mathcal{A}(\text{srs}) \\ \langle \mathcal{P}(\text{ipk}, \mathfrak{x}, \mathfrak{w}), \mathcal{V}(\text{ivk}, \mathfrak{x}) \rangle = 1 & (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{srs}}(\mathfrak{i}) \end{array} \right] = 1 .$$

- **Soundness.** For all size bounds $N \in \mathbb{N}$ and efficient $\tilde{\mathcal{P}} = (\tilde{\mathcal{P}}_1, \tilde{\mathcal{P}}_2)$,

$$\Pr \left[\begin{array}{c|c} (\mathfrak{i}, \mathfrak{x}) \notin \mathcal{L}(\mathcal{R}_N) & \text{srs} \leftarrow \mathcal{G}(1^\lambda, N) \\ \wedge & (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs}) \\ \langle \tilde{\mathcal{P}}_2(\text{st}), \mathcal{V}(\text{ivk}, \mathfrak{x}) \rangle = 1 & (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{srs}}(\mathfrak{i}) \end{array} \right] = \text{negl}(\lambda) .$$

Our definition of completeness allows $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ to depend on srs , while our formulation of soundness allows $(\mathfrak{i}, \mathfrak{x})$ to depend on srs .

All constructions in this paper achieve the stronger property of *knowledge soundness*, and optionally also the property of (perfect) *zero knowledge*. We define these properties below.

Knowledge soundness. We say that $\text{ARG} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ has knowledge soundness if for every size bound $N \in \mathbb{N}$ and efficient adversary $\tilde{\mathcal{P}} = (\tilde{\mathcal{P}}_1, \tilde{\mathcal{P}}_2)$ there exists an efficient extractor \mathcal{E} such that

$$\Pr \left[\begin{array}{c} (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \notin \mathcal{R}_N \\ \wedge \\ \langle \tilde{\mathcal{P}}_2(\text{st}), \mathcal{V}(\text{ivk}, \mathfrak{x}) \rangle = 1 \end{array} \middle| \begin{array}{c} \text{srs} \leftarrow \mathcal{G}(1^\lambda, N) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs}) \\ \mathfrak{w} \leftarrow \mathcal{E}(\text{srs}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{srs}}(\mathfrak{i}) \end{array} \right] = \text{negl}(\lambda) .$$

Zero knowledge. We say that $\text{ARG} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ has (perfect) zero knowledge if there exists an efficient simulator $\mathcal{S} = (\text{Setup}, \text{Prove})$ such that for every efficient adversary $\tilde{\mathcal{V}} = (\tilde{\mathcal{V}}_1, \tilde{\mathcal{V}}_2)$ it holds that

$$\begin{aligned} & \Pr \left[\begin{array}{c} (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \in \mathcal{R}_N \\ \wedge \\ \langle \mathcal{P}(\text{ipk}, \mathfrak{x}, \mathfrak{w}), \tilde{\mathcal{V}}_2(\text{st}) \rangle = 1 \end{array} \middle| \begin{array}{c} \text{srs} \leftarrow \mathcal{G}(1^\lambda, N) \\ (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}, \text{st}) \leftarrow \tilde{\mathcal{V}}_1(\text{srs}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{srs}}(\mathfrak{i}) \end{array} \right] \\ &= \Pr \left[\begin{array}{c} (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \in \mathcal{R}_N \\ \wedge \\ \langle \mathcal{S}.\text{Prove}(\text{trap}, \mathfrak{i}, \mathfrak{x}), \tilde{\mathcal{V}}_2(\text{st}) \rangle = 1 \end{array} \middle| \begin{array}{c} (\text{srs}, \text{trap}) \leftarrow \mathcal{S}.\text{Setup}(1^\lambda, N) \\ (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}, \text{st}) \leftarrow \tilde{\mathcal{V}}_1(\text{srs}) \end{array} \right] . \end{aligned}$$

Efficiency. We say that $\text{ARG} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ is:

- *index efficient* if the running time of the prover $\mathcal{P}(\text{ipk}, \mathfrak{x}, \mathfrak{w})$ is $\text{poly}_\lambda(|\mathfrak{i}|)$, i.e., it does not depend on the size of the universal structured reference string srs ;
- *proof succinct* if the size of the communication transcript between the prover $\mathcal{P}(\text{ipk}, \mathfrak{x}, \mathfrak{w})$ and verifier $\mathcal{V}(\text{ivk}, \mathfrak{x})$ is $\text{poly}(\lambda)$, i.e., the size is bounded by a universal polynomial in the security parameter λ ;
- *verifier succinct* if the running time of $\mathcal{V}(\text{ivk}, \mathfrak{x})$ is $\text{poly}(\lambda + |\mathfrak{x}|)$, i.e., the time is bounded by a universal polynomial in the security parameter λ and the size of the instance \mathfrak{x} and *does not* depend on the size of the index \mathfrak{i} that led to ivk .

Index efficiency implies that ipk output by \mathcal{I} is of size $\text{poly}_\lambda(|\mathfrak{i}|)$, while verifier succinctness implies that ivk output by \mathcal{I} is of size $\text{poly}(\lambda)$. All constructions in this paper are index efficient, proof succinct, and verifier succinct.

Public coins. We say that $\text{ARG} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ is *public-coin* if every message output by the verifier \mathcal{V} is a uniform random string of some prescribed length. All constructions in this paper are public-coin, and have a (small) constant number of rounds; in particular, they can be “squashed” to non-interactive arguments that are publicly verifiable by additionally using random oracles via the Fiat–Shamir transformation [FS86]. Hence, due to their succinctness, our constructions directly lead to preprocessing zkSNARKs with universal SRS.

Remark 3.7.1 (updatable SRS). An SRS is *updatable* [GKMMM18] if there exists an update algorithm that can be run at any time by anyone to update the SRS, with the guarantee that security holds as long as there is at least one honest updater since the beginning of time. This property significantly simplifies cryptographic ceremonies to sample the SRS. All preprocessing arguments that we construct in this paper have updatable SRS because they only contain “monomial terms”, and thus fall within the framework of [GKMMM18].

Remark 3.7.2 (auxiliary inputs). The definition of knowledge soundness above does not consider auxiliary inputs, for simplicity. One could consider a stronger definition, where the adversary and extractor additionally receive an auxiliary input z sampled from a *fixed* distribution $\mathcal{Z}(1^\lambda)$, or even sampled from *any* distribution $\mathcal{Z}(1^\lambda)$ that belongs to a given class. Such stronger definitions are useful when using argument systems as subroutines within other protocols. When relying on auxiliary inputs, however, one must be careful to ensure that they come from “benign” distributions, or else extraction is impossible, as discussed in [BP15; BCPR16]. We stress that all of our constructions of argument systems directly extend to hold with respect to an auxiliary-input distribution $\mathcal{Z}(1^\lambda)$ under the assumption that the relevant underlying knowledge assumptions are extended to hold with respect to the auxiliary-input distribution $\mathcal{Z}(1^\lambda)$ concatenated with some randomness. (In other words, our security reduction adds to the auxiliary input some random strings.)

3.8 From AHPs to preprocessing arguments with universal SRS

The following theorems capture key properties of our compiler.

Theorem 3.8.1. *Let \mathcal{F} be a field family and let \mathcal{R} be an indexed relation. Consider the following components:*

- AHP = $(k, s, d, \mathbf{I}, \mathbf{P}, \mathbf{V})$ is an AHP over \mathcal{F} for \mathcal{R} with negligible soundness error (see Section 3.4);
- PC = (Setup, Trim, Commit, Open, Check) is a polynomial commitment scheme over \mathcal{F} (see Section 3.6).

Then ARG = $(\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ described in Section 3.8.1 is a preprocessing argument with universal SRS for \mathcal{R} (see Section 3.7). Moreover, if q is the query complexity of AHP, ARG has the following efficiency:

- **round complexity** is $k + 2$;
- **communication complexity** is $O_\lambda(q)$ bits if PC is additionally succinct (see Definition 3.6.3);
- **indexer time** is the sum of the indexer time in AHP and the time to commit to $s(0)$ polynomials in PC;
- **prover time** is the sum of the prover time in AHP, the time to commit to $\sum_{i=1}^k s(i)$ polynomials in PC, the time to produce evaluations that answer the q queries along with a batch evaluation proof for them in PC;
- **verifier time** is the sum of the verifier time in AHP and the time to batch verify q evaluations in PC.

Remark 3.8.2 (updatable SRS). If the SRS for PC is updatable then so is the SRS for ARG. All constructions of polynomial commitments in this paper satisfy this property, including the one used in Section 3.9.

The construction underlying the above theorem preserves knowledge soundness and, if the polynomial commitment scheme is also hiding, preserves zero knowledge.

Theorem 3.8.3. *In Theorem 3.8.1, if AHP has a negligible knowledge soundness error, then ARG has knowledge soundness.*

Theorem 3.8.4. *In Theorem 3.8.1, if PC is hiding and if AHP is zero knowledge with query bound q (the query complexity of AHP) and some polynomial-time query checker \mathbf{C} , then ARG is (perfect) zero knowledge.*

Remark 3.8.5 (the multivariate case). In this paper we give definitions for algebraic holographic proofs and polynomial commitment schemes that are restricted to the case of univariate polynomials, because the constructions that we consider are univariate. Theorems 3.8.1, 3.8.3 and 3.8.4, however, directly extend to the multivariate case when considering an AHP in the general case of multivariate polynomials and a polynomial commitment scheme for multivariate polynomials. This provides a proof of security for several prior works that considered constructions that are special cases of this paradigm but did not prove security (because the polynomial commitment schemes were only assumed to satisfy evaluation binding as discussed in Section 3.1.2).

3.8.1 Construction

We describe the construction behind Theorem 3.8.1, and then discuss its efficiency features.

Generator \mathcal{G} . The generator \mathcal{G} , on input a security parameter $\lambda \in \mathbb{N}$ and size bound $N \in \mathbb{N}$, uses N to compute a maximum degree bound $D \in \mathbb{N}$, samples public parameters $\text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D)$ for the polynomial commitment scheme PC , and then outputs $\text{srs} := \text{pp}$. The integer D is computed to be the maximum degree bound in AHP for indices of size N . In other words,

$$D := \max \left\{ d(N, i, j) \mid i \in \{0, 1, \dots, k(N)\}, j \in \{1, \dots, s(i)\} \right\}. \quad (3.9)$$

Indexer \mathcal{I} . The indexer \mathcal{I} upon input \mathfrak{i} and given oracle access to srs , deduces the field $\mathbb{F} \in \mathcal{F}$ contained in $\text{srs} = \text{pp}$, runs the AHP indexer \mathbf{I} on $(\mathbb{F}, \mathfrak{i})$ to obtain $s(0)$ polynomials $p_{0,1}, \dots, p_{0,s(0)} \in \mathbb{F}[X]$ of degrees at most $d(|\mathfrak{i}|, 0, 1), \dots, d(|\mathfrak{i}|, 0, s(0))$, computes the degree bounds for the index and prover polynomials, invokes PC.Trim on these bounds to compute (ck, rk) that are specialized for these degree bounds, and computes commitments to all of the index polynomials.

Namely, \mathcal{I} calculates the bounds $\mathbf{d} := \{d(|\mathfrak{i}|, i, s(i))\}_{i=0}^{k(|\mathfrak{i}|)}$, invokes $(\text{ck}, \text{rk}) := \text{PC.Trim}^{\text{srs}}(\mathbf{d})$, and then computes $[c_{0,j}]_{j=1}^{s(0)} := \text{PC.Commit}(\text{ck}, [p_{0,j}]_{j=1}^{s(0)}, [d(|\mathfrak{i}|, 0, j)]_{j=1}^{s(0)}; [\omega_{0,j}]_{j=1}^{s(0)})$ for “empty randomness” $[\omega_{0,j}]_{j=1}^{s(0)} := \perp$. The indexer \mathcal{I} outputs $\text{ipk} := (\text{ck}, \mathfrak{i}, [p_{0,j}]_{j=1}^{s(0)}, [c_{0,j}]_{j=1}^{s(0)})$ and $\text{ivk} := (\text{rk}, [c_{0,j}]_{j=1}^{s(0)})$. (Note that $[c_{0,j}]_{j=1}^{s(0)}$ are commitments to non-secret information so no randomness is used in producing them. In particular, \mathcal{I} is a deterministic polynomial-time algorithm, as required. Also see Remark 3.8.6 below for additional considerations.)

Prover \mathcal{P} and verifier \mathcal{V} . The prover \mathcal{P} receives $(\text{ipk}, \mathfrak{x}, \mathfrak{w})$ and the verifier \mathcal{V} receives $(\text{ivk}, \mathfrak{x})$, where (ipk, ivk) is the index key pair output by $\mathcal{I}^{\text{srs}}(\mathfrak{i})$, and $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ is in the indexed relation \mathcal{R} . By construction of \mathcal{I} , ipk contains a trimmed committer key ck and ivk contains a trimmed receiver key rk for the polynomial commitment scheme PC . Let $\mathbb{F} \in \mathcal{F}$ be the field described by (ck, rk) (each of ck and rk individually contain a description of \mathbb{F}), and let $k := k(|\mathfrak{i}|)$ be the number of rounds in AHP. For $i \in \{1, \dots, k\}$, \mathcal{P} and \mathcal{V} simulate the i -th round of the interaction between the AHP prover $\mathbf{P}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ and the AHP verifier $\mathbf{V}(\mathbb{F}, \mathfrak{x})$.

1. \mathcal{V} receives $\rho_i \in \mathbb{F}^*$ from \mathbf{V} , and forwards it to \mathcal{P} .
2. \mathcal{P} forwards ρ_i to \mathbf{P} , which replies with polynomials $p_{i,1}, \dots, p_{i,s(i)} \in \mathbb{F}[X]$ with $\deg(p_{i,j}) \leq d(|\mathfrak{i}|, i, j)$.
3. \mathcal{P} samples commitment randomness $[\omega_{i,j}]_{j=1}^{s(i)}$ and sends to \mathcal{V} the polynomial commitments below

$$[c_{i,j}]_{j=1}^{s(i)} := \text{PC.Commit}(\text{ck}, [p_{i,j}]_{j=1}^{s(i)}, [d(|\mathfrak{i}|, i, j)]_{j=1}^{s(i)}; [\omega_{i,j}]_{j=1}^{s(i)}) .$$

4. \mathcal{V} notifies \mathbf{V} that the i -th round has finished.

The prover \mathcal{P} and verifier \mathcal{V} are done simulating the interactive phase of AHP, and in the remaining two rounds simulate the (non-adaptive) query phase of AHP. Below we use \mathbf{c} to denote the commitments $[[c_{i,j}]_{j=1}^{s(i)}]_{i=0}^k$, \mathbf{p} to denote the polynomials $[[p_{i,j}]_{j=1}^{s(i)}]_{i=0}^k$, \mathbf{d} to denote the degree bounds $[[d(|\mathfrak{i}|, i, j)]_{j=1}^{s(i)}]_{i=0}^k$, and $\boldsymbol{\omega}$ to denote the randomness $[[\omega_{i,j}]_{j=1}^{s(i)}]_{i=0}^k$ with $[\omega_{0,j}]_{j=1}^{s(0)} := \perp$. Note that

these three vectors include the commitments, polynomials, degrees, and randomness of the “0-th round”.

- \mathcal{V} sends a message $\rho_{k+1} \in \mathbb{F}^*$ that represents randomness for the query phase of $\mathbf{V}(\mathbb{F}, \mathbb{X})$ to \mathcal{P} .
- \mathcal{P} uses the query algorithm of \mathbf{V} to compute the query set $Q := \mathbf{Q}_{\mathbf{V}}(\mathbb{F}, \mathbb{X}; \rho_1, \dots, \rho_k, \rho_{k+1})$.
- \mathcal{P} replies with answers $\mathbf{v} := \mathbf{p}(Q)$.
- \mathcal{V} samples and sends an opening challenge $\xi \in \mathbb{F}$ to \mathcal{P} .
- \mathcal{P} replies with an evaluation proof to demonstrate correctness of all claimed evaluations:

$$\pi := \text{PC.Open}(\text{ck}, \mathbf{p}, \mathbf{d}, Q, \xi; \omega) .$$

- \mathcal{V} accepts if and only if the following conditions hold:
 - the decision algorithm of \mathbf{V} accepts the answers, i.e., $\mathbf{D}_{\mathbf{V}}(\mathbb{F}, \mathbb{X}, \mathbf{v}; \rho_1, \dots, \rho_k, \rho_{k+1}) = 1$;
 - the alleged answers pass the test, i.e., $\text{PC.Check}(\text{rk}, \mathbf{c}, \mathbf{d}, Q, \mathbf{v}, \pi, \xi) = 1$.

Completeness of the preprocessing argument ARG follows in a straightforward way from completeness of the AHP AHP and completeness of the polynomial commitment scheme PC.

We now discuss the efficiency features of the construction above.

- *Round complexity.* The first k rounds simulate the interactive phase of AHP, with polynomials sent as commitments; one round is to answer the desired queries; and one round is to certify the queries’ answers.
- *Communication complexity.* The argument prover \mathcal{P} sends $\sum_{i=1}^k s(i)$ commitments, q field elements representing query answers, and an evaluation proof that certifies the q answers. The argument verifier \mathcal{V} sends $|\rho_1| + \dots + |\rho_k| + |\rho_{k+1}| + 1$ field elements. In Theorem 3.8.1 we state that the communication complexity is $O_{\lambda}(q)$ because typically it holds that $\sum_{i=0}^k s(i) \leq q$ (each polynomial is queried at least once) and $|\rho_1| + \dots + |\rho_k| + |\rho_{k+1}|$ is a small constant (each verifier message is a few field elements).
- *Indexer time.* The time complexity of \mathcal{I} equals the time complexity of the AHP indexer \mathbf{I} plus the time to trim the PC public parameters pp , and then to commit to the $s(0)$ polynomials output by \mathbf{I} .
- *Prover time.* The time complexity of \mathcal{P} equals the time complexity of the AHP prover \mathbf{P} plus the time to commit to the $\sum_{i=1}^k s(i)$ polynomials output by \mathbf{P} , evaluate $\sum_{i=0}^k s(i)$ polynomials at the query set Q , and produce an evaluation proof that certifies the correctness of these evaluations.
- *Verifier time.* The time complexity of \mathcal{V} equals the time complexity of the AHP verifier \mathbf{V} plus the time to verify the batch evaluation proof for the q evaluations that provide answers to the query set Q .

Remark 3.8.6 (commitments to index polynomials). The construction described above uses the same polynomial commitment scheme PC for committing to polynomials output by the AHP indexer and to polynomials output by the AHP prover. This simplifies exposition, and allows for a single evaluation proof to certify all query answers. For security, however, it would suffice (even for Theorem 3.8.4) to commit to index polynomials via a commitment scheme that merely satisfies

“evaluation binding” (the full version[CHMMVW20]), which is strictly weaker than the notion of extractability that we use for the other commitments. This is because the commitments in the index verification key are honestly produced in the preprocessing phase. Moreover, for Theorem 3.8.3 to hold we do *not* need the commitments to index polynomials to be hiding.

3.8.2 Proof of Theorem 3.8.1

Suppose that $\tilde{\mathcal{P}} = (\tilde{\mathcal{P}}_1, \tilde{\mathcal{P}}_2)$ is an efficient adversarial prover for ARG that wins with probability at least ϵ , that is,

$$\Pr \left[\begin{array}{c} (\mathfrak{i}, \mathfrak{x}) \notin \mathcal{L}(\mathcal{R}_N) \\ \wedge \\ \langle \tilde{\mathcal{P}}_2(\text{st}), \mathcal{V}(\text{ivk}, \mathfrak{x}) \rangle = 1 \end{array} \mid \begin{array}{c} \text{srs} \leftarrow \mathcal{G}(1^\lambda, N) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{srs}}(\mathfrak{i}) \end{array} \right] \geq \epsilon(\lambda) .$$

We assume without loss of generality that st output by $\tilde{\mathcal{P}}_1$ contains the public parameters $\text{srs} = \text{pp}$. Also note that $\tilde{\mathcal{P}}_2$ can be represented via its $k + 2$ next-message functions:

$$\tilde{\mathcal{P}}_2(\text{st}; \rho_1) , \tilde{\mathcal{P}}_2(\text{st}; \rho_1, \rho_2) , \dots , \tilde{\mathcal{P}}_2(\text{st}; \rho_1, \dots, \rho_k) , \tilde{\mathcal{P}}_2(\text{st}; \rho_1, \dots, \rho_k, Q) , \tilde{\mathcal{P}}_2(\text{st}; \rho_1, \dots, \rho_k, Q, \xi) .$$

We describe how to construct a prover $\tilde{\mathcal{P}}$, which is admissible for AHP, and an efficient adversary \mathcal{A}_{PC} against the extractability of PC such that

$$\Pr \left[\begin{array}{c} (\mathfrak{i}, \mathfrak{x}) \notin \mathcal{L}(\mathcal{R}_N) \\ \wedge \\ \langle \tilde{\mathcal{P}}(\text{st}), \mathbf{V}^{\mathbb{I}(\mathbb{F}, \mathfrak{i})}(\mathbb{F}, \mathfrak{x}) \rangle = 1 \end{array} \mid \begin{array}{c} \text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D) \\ \mathbb{F} \leftarrow \text{field}(\text{pp}) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{pp}) \end{array} \right] + \Pr \left[\begin{array}{c} \mathcal{A}_{\text{PC}} \text{ wins the} \\ \text{extractability game} \end{array} \right] \geq \epsilon(\lambda) ,$$

Above, D is computed according to Eq. (3.9) and \mathbb{F} is the field described in pp . This concludes the proof because if $\epsilon(\lambda)$ were to be non-negligible then either: (i) by averaging there would exist a choice of public parameters pp that yields a state st , field $\mathbb{F} \in \mathcal{F}$, and $(\mathfrak{i}, \mathfrak{x}) \notin \mathcal{L}(\mathcal{R})$ for which $\Pr[\langle \tilde{\mathcal{P}}(\text{st}), \mathbf{V}^{\mathbb{I}(\mathbb{F}, \mathfrak{i})}(\mathbb{F}, \mathfrak{x}) \rangle = 1]$ is non-negligible, contradicting our hypothesis AHP has negligible soundness error; or (ii) there would exist an efficient adversary \mathcal{A}_{PC} that, for any given efficient extractor, succeeds in the extractability game for PC (Definition 3.6.2) with non-negligible probability, contradicting our hypothesis PC is extractable.

Constructing \mathcal{A}_{PC} . The adversary \mathcal{A}_{PC} is built from the argument prover $\tilde{\mathcal{P}}$ (and the argument indexer \mathcal{I} and degree bounds d) as follows. For round $i \in \{0, \dots, k\}$ and verifier messages ρ_0, \dots, ρ_i :

$\mathcal{A}_{\text{PC}}(\text{ck}, \text{rk}, \rho_0, \rho_1, \dots, \rho_i)$:

1. Set $\text{srs} := \text{pp}$ and compute $(\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs})$.
2. If $i = 0$, ignore ρ_0 , compute index keys $(\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{srs}}(\mathfrak{i})$, and parse ivk as polynomial commitments $[c_{0,j}]_{j=1}^{\text{s}(0)}$. If $i > 0$, compute polynomial commitments $[c_{i,j}]_{j=1}^{\text{s}(i)} \leftarrow \tilde{\mathcal{P}}_2(\text{st}; \rho_1, \dots, \rho_i)$.
3. For each $j \in \{1, \dots, \text{s}(i)\}$, compute the degree $d_{i,j} := d(|\mathfrak{i}|, i, j)$.
4. Output $([c_{i,j}]_{j=1}^{\text{s}(i)}, [d_{i,j}]_{j=1}^{\text{s}(i)})$.

Since $\tilde{\mathcal{P}}, \mathcal{I}, d$ are all efficient, so is \mathcal{A}_{PC} . Let \mathcal{E}_{PC} be the extractor for \mathcal{A}_{PC} . Note that in the “0-th round”, \mathcal{A}_{PC} outputs the commitments generated by the indexer \mathcal{I} . To capture that these “0-th round” commitments need only satisfy evaluation binding (unlike the commitments in all other rounds), we consider an extractor \mathcal{E}'_{PC} that works as follows. For round $i \in \{0, \dots, k\}$ and verifier messages ρ_0, \dots, ρ_i :

- $\mathcal{E}'_{\text{PC}}(\text{pp}, \rho_0, \rho_1, \dots, \rho_i)$:
1. Set $\text{srs} := \text{pp}$ and compute $(\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs})$.
Obtain the field description $\mathbb{F} \leftarrow \text{field}(\text{pp})$.
 2. If $i = 0$, output polynomials $[p_{0,j}]_{j=1}^{s(0)} \leftarrow \mathbf{I}(\mathbb{F}, \mathfrak{i})$.
If $i > 0$, output polynomials $[p_{i,j}]_{j=1}^{s(i)} \leftarrow \mathcal{E}_{\text{PC}}(\text{st}; \rho_1, \dots, \rho_i)$.

Observe that the probability that \mathcal{E}'_{PC} succeeds for \mathcal{A}_{PC} is at least the probability that \mathcal{E}_{PC} succeeds for \mathcal{A}_{PC} .

Constructing $\tilde{\mathbf{P}}$. We define $\tilde{\mathbf{P}}$ via its k next-message functions, by relying on the polynomial commitment extractor \mathcal{E}'_{PC} defined above. For round number $i \in \{1, \dots, k\}$ and verifier messages ρ_1, \dots, ρ_i :

- $\tilde{\mathbf{P}}(\text{st}; \rho_1, \dots, \rho_i)$:
1. Set $\rho_0 := \perp$ and run $\mathcal{E}'_{\text{PC}}(\text{pp}, \rho_0, \rho_1, \dots, \rho_i)$ to obtain polynomials $p_{i,1}, p_{i,2}, \dots, p_{i,s(i)} \in \mathbb{F}[X]$.
 2. Check that for every $j \in [s(i)]$ it holds that $\deg(p_{i,j}) \leq d(|\mathfrak{i}|, i, j)$. (If not, output \perp .)
 3. Output the polynomials $p_{i,1}, p_{i,2}, \dots, p_{i,s(i)}$.

Observe that, by construction, $\tilde{\mathbf{P}}$ is an admissible prover for AHP.

Analyzing $\tilde{\mathbf{P}}$ and \mathcal{A}_{PC} . Define $\epsilon_{\text{PC}}(\lambda) := \Pr[\mathcal{A}_{\text{PC}} \text{ wins the extractability game}]$. We want to argue that

$$\begin{aligned} & \Pr \left[\begin{array}{l} (\mathfrak{i}, \mathfrak{x}) \notin \mathcal{L}(\mathcal{R}_{\mathbf{N}}) \\ \wedge \\ \langle \tilde{\mathcal{P}}_2(\text{st}), \mathcal{V}(\text{ivk}, \mathfrak{x}) \rangle = 1 \end{array} \left| \begin{array}{l} \text{srs} \leftarrow \mathcal{G}(1^\lambda, \mathbf{N}) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{srs}}(\mathfrak{i}) \end{array} \right. \right] \\ & \leq \Pr \left[\begin{array}{l} (\mathfrak{i}, \mathfrak{x}) \notin \mathcal{L}(\mathcal{R}_{\mathbf{N}}) \\ \wedge \\ \langle \tilde{\mathbf{P}}(\text{st}), \mathbf{V}^{\mathbf{I}(\mathbb{F}, \mathfrak{i})}(\mathbb{F}, \mathfrak{x}) \rangle = 1 \end{array} \left| \begin{array}{l} \text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D) \\ \mathbb{F} \leftarrow \text{field}(\text{pp}) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{pp}) \end{array} \right. \right] + \epsilon_{\text{PC}}(\lambda) . \end{aligned}$$

First recall that by construction it holds that $\mathcal{G}(1^\lambda, \mathbf{N}) = \text{PC.Setup}(1^\lambda, D)$. It follows that the distributions of $\text{srs/pp}, \mathfrak{i}, \mathfrak{x}, \text{st}$, as well as the underlying field \mathbb{F} , are identical in the two probability expressions above.

Next recall that we have constructed $\tilde{\mathbf{P}}$ in such a way that, in round $i \in \{1, \dots, k\}$, $\tilde{\mathbf{P}}$ outputs polynomials that (provided \mathcal{E}'_{PC} has succeeded) correspond to the commitments output in round i by $\tilde{\mathcal{P}}_2$. At the same time, we have constructed \mathcal{V} in such a way that in the first k rounds \mathcal{V} behaves exactly as \mathbf{V} , and in the remaining two rounds \mathcal{V} uses the polynomial commitment scheme to

validate, against commitments received from the prover and contained in ivk , the answers claimed by $\tilde{\mathcal{P}}_2$ in response to \mathbf{V} 's query set Q , and then checks that \mathbf{V} accepts these answers.

Hence, as long as $\tilde{\mathcal{P}}$ provides correct evaluations for polynomials committed to in ivk , and as long as $\tilde{\mathbf{P}}$ outputs polynomials that correspond to the commitments output by $\tilde{\mathcal{P}}_2$, it holds that \mathbf{V} accepts whenever \mathcal{V} accepts. Since $\tilde{\mathbf{P}}$ relies on the extractor \mathcal{E}'_{PC} for the polynomial commitment to find such polynomials (if they exist) and to correctly answer queries to polynomials in ivk , $\tilde{\mathbf{P}}$ “works” whenever \mathcal{E}'_{PC} and $\tilde{\mathcal{P}}$ do.

We now argue that, whenever \mathcal{V} accepts, \mathcal{E}'_{PC} has succeeded, up to the error $\epsilon_{\text{PC}}(\lambda)$. This is because the interaction between $\tilde{\mathcal{P}}_2$ and \mathcal{V} can be re-cast as an extractability game for PC, as we now explain. Define a public-coin challenger \mathcal{C} to output randomness $\rho_0 := \perp$ in the 0-th round, and to equal the interactive phase of $\mathbf{V}(\mathbb{F}, \mathbb{x})$ in the remaining rounds. This means that in the i -th round (for $i \in \{1, \dots, k\}$) the challenger \mathcal{C} will output the randomness ρ_i output by $\mathbf{V}(\mathbb{F}, \mathbb{x})$ in round i . Also, define a query sampler \mathcal{Q} to equal the query phase of $\mathbf{V}(\mathbb{F}, \mathbb{x})$: given all challenger outputs $[\rho_j]_{j=1}^k$ so far and auxiliary input ρ_{k+1} , compute the query set $Q := \mathbf{Q}_{\mathbf{V}}(\mathbb{F}, \mathbb{x}; \rho_1, \dots, \rho_k, \rho_{k+1})$. Finally, let $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ be the adversary defined below.

$$\begin{array}{ll} \mathcal{B}_1(\text{pp}, [\rho_j]_{j=0}^k, Q): & \mathcal{B}_2(\text{st}_{\text{PC}}, \xi): \\ 1. \text{ Set } \text{srs} := \text{pp}. & 1. \text{ Parse } \text{st}_{\text{PC}} \text{ as } (\text{st}, \rho_1, \dots, \rho_k, Q) \\ 2. \text{ Compute } (\mathfrak{i}, \mathbb{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs}). & 2. \text{ Compute } \pi \leftarrow \\ 3. \text{ Compute } \mathbf{v} \leftarrow \tilde{\mathcal{P}}_2(\text{st}; \rho_1, \dots, \rho_k, Q). & \tilde{\mathcal{P}}_2(\text{st}; \rho_1, \dots, \rho_k, Q, \xi). \\ 4. \text{ Set } \text{st}_{\text{PC}} := (\text{st}, \rho_1, \dots, \rho_k, Q). & 3. \text{ Output } \pi. \\ 5. \text{ Output } (\mathbf{v}, \text{st}_{\text{PC}}). & \end{array}$$

Using the above definitions of \mathcal{C} , \mathcal{Q} , \mathcal{B} , and \mathcal{E}'_{PC} we obtain the following inequality:

$$\begin{array}{l} \Pr \left[\begin{array}{l} (\mathfrak{i}, \mathbb{x}) \notin \mathcal{L}(\mathcal{R}_{\mathbf{N}}) \\ \wedge \\ \langle \tilde{\mathcal{P}}_2(\text{st}), \mathcal{V}(\text{ivk}, \mathbb{x}) \rangle = 1 \end{array} \mid \begin{array}{l} \text{srs} \leftarrow \mathcal{G}(1^\lambda, \mathbf{N}) \\ (\mathfrak{i}, \mathbb{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{srs}}(\mathfrak{i}) \end{array} \right] \\ \leq \Pr \left[\begin{array}{l} (\mathfrak{i}, \mathbb{x}) \notin \mathcal{L}(\mathcal{R}_{\mathbf{N}}) \\ \wedge \\ \text{PC.Check}(\text{rk}, \mathbf{c}, \mathbf{d}, Q, \mathbf{v}, \pi, \xi) = 1 \\ \wedge \\ \deg(\mathbf{p}) \leq \mathbf{d} \leq D \text{ and } \mathbf{v} = \mathbf{p}(Q) \\ \wedge \\ \langle \tilde{\mathbf{P}}(\text{st}), \mathbf{V}^{\mathbf{I}(\mathbb{F}, \mathfrak{i})}(\mathbb{F}, \mathbb{x}; \rho_1, \dots, \rho_k, \rho_{k+1}) \rangle = 1 \end{array} \mid \begin{array}{l} \text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D) \\ \mathbb{F} \leftarrow \text{field}(\text{rk}) \\ (\mathfrak{i}, \mathbb{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{pp}) \\ \hline \text{For } i = 0, \dots, k: \\ \rho_i \leftarrow \mathcal{C}(\text{pp}, i) \\ ([c_{i,j}]_{j=1}^{s(i)}, [d_{i,j}]_{j=1}^{s(i)}) \leftarrow \mathcal{A}_{\text{PC}}(\text{pp}, [\rho_j]_{j=0}^i) \\ [p_{i,j}]_{j=1}^{s(i)} \leftarrow \mathcal{E}'_{\text{PC}}(\text{pp}, [\rho_j]_{j=0}^i) \\ \hline Q \leftarrow \mathcal{Q}(\text{pp}, [\rho_j]_{j=0}^k; \rho_{k+1}) \\ (\mathbf{v}, \text{st}) \leftarrow \mathcal{B}_1(\text{pp}, [\rho_j]_{j=0}^k, Q) \\ \text{Sample opening challenge } \xi \\ \pi \leftarrow \mathcal{B}_2(\text{st}, \xi) \\ \text{Set } \mathbf{c} := [[c_{i,j}]_{j=1}^{s(i)}]_{i=0}^k, \mathbf{d} := [[d_{i,j}]_{j=1}^{s(i)}]_{i=0}^k \\ (\text{ck}, \text{rk}) \leftarrow \text{PC.Trim}^{\text{pp}}(1^\lambda, \mathbf{d}) \end{array} \right] + \epsilon_{\text{PC}}(\lambda) . \end{array}$$

As argued above, whenever \mathcal{E}'_{PC} and $\tilde{\mathcal{P}}$ succeed, $\tilde{\mathbf{P}}$ does. The first term after the inequality captures the case where the AHP verifier \mathbf{V} is convinced to accept a pair $(\mathfrak{i}, \mathfrak{x})$ not in the indexed language $\mathcal{L}(\mathcal{R}_{\mathbf{N}})$. If \mathcal{A}_{PC} succeeds, then there is still some chance that $\tilde{\mathbf{P}}$ succeeds assuming it holds that $\deg(\mathbf{p}) \leq \mathbf{d} \leq D$ for the polynomials output by \mathcal{E}'_{PC} (otherwise $\tilde{\mathbf{P}}$ outputs \perp). This joint success probability is upper bounded by the probability that just \mathcal{A}_{PC} succeeds, which is in turn upper bounded by $\epsilon_{\text{PC}}(\lambda)$. Hence the $\epsilon_{\text{PC}}(\lambda)$ term above and the inequality rather than equality above. Since the above inequality implies our claim, we have concluded the proof.

3.8.3 Proof of Theorem 3.8.3

Let \mathbf{E} be the extractor for AHP, which by hypothesis has a negligible knowledge soundness error $\epsilon_{\text{AHP}}(\lambda)$. Suppose that $\tilde{\mathcal{P}} = (\tilde{\mathcal{P}}_1, \tilde{\mathcal{P}}_2)$ is an efficient adversary for ARG. We use $\tilde{\mathcal{P}}$ to construct an admissible prover $\tilde{\mathbf{P}}$ for AHP, exactly as in the proof of soundness (see Section 3.8.2). Then we define the extractor \mathcal{E} for $\tilde{\mathcal{P}}$ to be as follows.

$\mathcal{E}(\text{srs})$:

1. Compute $(\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs})$.
2. Compute $\mathbb{F} \leftarrow \text{field}(\text{srs})$.
3. Compute $\mathfrak{w} \leftarrow \mathbf{E}^{\tilde{\mathbf{P}}(\text{st})}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, 1^{l(|\mathfrak{i}|)})$.
4. Output \mathfrak{w} .

Observe that by construction we have the equality:

$$\begin{aligned} & \Pr \left[\begin{array}{l} (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \notin \mathcal{R}_{\mathbf{N}} \\ \wedge \\ \langle \tilde{\mathcal{P}}_2(\text{st}), \mathcal{V}(\text{ivk}, \mathfrak{x}) \rangle = 1 \end{array} \middle| \begin{array}{l} \text{srs} \leftarrow \mathcal{G}(1^\lambda, \mathbf{N}) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{srs}}(\mathfrak{i}) \\ \mathfrak{w} \leftarrow \mathcal{E}(\text{srs}) \end{array} \right] \\ &= \Pr \left[\begin{array}{l} (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \notin \mathcal{R}_{\mathbf{N}} \\ \wedge \\ \langle \tilde{\mathcal{P}}_2(\text{st}), \mathcal{V}(\text{rk}, \text{ivk}, \mathfrak{x}) \rangle = 1 \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D) \\ \mathbb{F} \leftarrow \text{field}(\text{pp}) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{pp}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{pp}}(\mathfrak{i}) \\ \mathfrak{w} \leftarrow \mathbf{E}^{\tilde{\mathbf{P}}(\text{st})}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, 1^{l(|\mathfrak{i}|)}) \end{array} \right]. \end{aligned}$$

Similarly to the proof of soundness (see Section 3.8.2), we can argue the following inequality:

$$\Pr \left[\begin{array}{l} (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \notin \mathcal{R}_N \\ \wedge \\ \langle \tilde{\mathcal{P}}_2(\text{st}), \mathcal{V}(\text{ivk}, \mathfrak{x}) \rangle = 1 \end{array} \left| \begin{array}{l} \text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D) \\ \mathbb{F} \leftarrow \text{field}(\text{pp}) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{pp}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}^{\text{PP}}(\mathfrak{i}) \\ \mathfrak{w} \leftarrow \mathbf{E}^{\tilde{\mathcal{P}}(\text{st})}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, 1^{l(|\mathfrak{i}|)}) \end{array} \right. \right] \\ \leq \Pr \left[\begin{array}{l} (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \notin \mathcal{R}_N \\ \wedge \\ \langle \tilde{\mathcal{P}}(\text{st}), \mathbf{V}^{\mathbf{I}(\mathbb{F}, \mathfrak{i})}(\mathbb{F}, \mathfrak{x}) \rangle = 1 \end{array} \left| \begin{array}{l} \text{pp} \leftarrow \text{PC.Setup}(1^\lambda, D) \\ \mathbb{F} \leftarrow \text{field}(\text{pp}) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{pp}) \\ \mathfrak{w} \leftarrow \mathbf{E}^{\tilde{\mathcal{P}}(\text{st})}(\mathbb{F}, \mathfrak{i}, \mathfrak{x}, 1^{l(|\mathfrak{i}|)}) \end{array} \right. \right] + \epsilon_{\text{PC}}(\lambda) .$$

The knowledge soundness of AHP implies that the probability above is at most $\epsilon_{\text{AHP}}(\lambda)$. Since $\epsilon_{\text{AHP}}(\lambda) + \epsilon_{\text{PC}}(\lambda)$ is negligible, we have established that the extractor \mathcal{E} for $\tilde{\mathcal{P}}$ works.

3.8.4 Proof of Theorem 3.8.4

Let \mathbf{S} be the zero knowledge simulator for AHP (see definition in Section 3.4), and let \mathcal{S}_{PC} be the simulator for PC (see definition in Section 3.6). We describe how to construct a (perfect) zero knowledge simulator $\mathcal{S} = (\text{Setup}, \text{Prove})$ for ARG (see definition in Section 3.7). Let $\tilde{\mathcal{V}} = (\tilde{\mathcal{V}}_1, \tilde{\mathcal{V}}_2)$ be any malicious verifier.

The simulated setup algorithm $\mathcal{S}.\text{Setup}$ receives a security parameter $\lambda \in \mathbb{N}$ and size bound $N \in \mathbb{N}$ as input, and then proceeds as follows. First, $\mathcal{S}.\text{Setup}$ uses N to compute the same maximum degree bound $D \in \mathbb{N}$ computed by the generator \mathcal{G} (see Eq. (3.9)). Second, it runs $\mathcal{S}_{\text{PC}}.\text{Setup}(1^\lambda, D)$ to sample simulated public parameters pp for the polynomial commitment and their trapdoor trap, and outputs $(\text{srs}, \text{trap}) := (\text{pp}, \text{trap})$. Let $\mathbb{F} \in \mathcal{F}$ be the field described in the public parameters pp .

The zero knowledge game states that first $\tilde{\mathcal{V}}_1$ receives srs , and then outputs an index-instance-witness tuple $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ and a state st to pass onto $\tilde{\mathcal{V}}_2$. The proving subroutine of the simulator, $\mathcal{S}.\text{Prove}$, receives $(\text{trap}, \mathfrak{i}, \mathfrak{x})$ as input, and interacts with $\tilde{\mathcal{V}}_2(\text{st})$ over $k + 2$ rounds. We construct $\mathcal{S}.\text{Prove}$ as follows.

1. For $i \in \{1, \dots, k\}$, simulate the polynomial commitments for round i as follows:
 - a) Receive a message $\rho_i \in \mathbb{F}^*$ from $\tilde{\mathcal{V}}_2$, and forward it to the AHP simulator $\mathbf{S}(\mathbb{F}, \mathfrak{i}, \mathfrak{x})$.
 - b) Sample commitment randomness $[\omega_{i,j}]_{j=1}^{\text{s}(i)}$, and then send to $\tilde{\mathcal{V}}_2$ the simulated commitments below

$$[c_{i,j}]_{j=1}^{\text{s}(i)} \leftarrow \mathcal{S}_{\text{PC}}.\text{Commit}(\text{trap}, [d(|\mathfrak{i}|, i, j)]_{j=1}^{\text{s}(i)}; [\omega_{i,j}]_{j=1}^{\text{s}(i)}) .$$

2. Simulate the evaluations in round $k + 1$ as follows:
 - a) Receive a message $\rho_{k+1} \in \mathbb{F}^*$ from $\tilde{\mathcal{V}}_2$.

- b) Use the (honest) query algorithm of AHP to compute the query set

$$Q := \mathbf{Q}_V(\mathbb{F}, \mathbb{x}; \rho_1, \dots, \rho_k, \rho_{k+1})$$

, and abort if any query does not satisfy the query checker \mathbf{C} . (The honest prover would also abort.)

- c) We need to assemble a list of evaluations \mathbf{v} , containing actual evaluations of index polynomials and simulated evaluations of prover polynomials. In more detail, first run the AHP indexer $\mathbf{I}(\mathbb{F}, \mathfrak{i})$ to obtain polynomials $[p_{0,j}]_{j=1}^{s(0)}$, and evaluate these on (the relevant queries in) the query set Q . Next, forward the query set Q to the AHP simulator $\mathbf{S}(\mathbb{F}, \mathfrak{i}, \mathbb{x})$ in order to obtain a simulated view, which in particular contains simulated answers for queries to the AHP prover's polynomials.

3. Simulate the evaluation proof in round $k + 2$ as follows:

- a) Receive a challenge ξ from $\tilde{\mathcal{V}}_2$.
- b) Compute $\pi \leftarrow \mathcal{S}_{\text{PC}}.\text{Open}(\text{trap}, [[p_{i,j}]_{j=1}^{s(i)}]_{i=0}^k, \mathbf{v}, [[d(|\mathfrak{i}|, i, j)]_{j=1}^{s(i)}]_{i=0}^k, Q, \xi; [[\omega_{i,j}]_{j=1}^{s(i)}]_{i=0}^k)$ where all polynomials $[p_{i,j}]_{j=1}^{s(i)}$ with $i > 0$ are defined to be zero and the randomness $[\omega_{0,j}]_{j=1}^{s(0)}$ is set to \perp .
- c) Send π to $\tilde{\mathcal{V}}_2$.

Lemma 3.8.7. *The view of the malicious verifier $\tilde{\mathcal{V}} = (\tilde{\mathcal{V}}_1, \tilde{\mathcal{V}}_2)$ while interacting with the honest prover is identically distributed as its view while interacting with the simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ described above.*

Proof. The zero knowledge property of AHP states that interaction with the honest prover $\mathbf{P}(\mathbb{F}, \mathfrak{i}, \mathbb{x}, \mathbb{w})$ can be replaced with interaction with the simulator $\mathbf{S}(\mathbb{F}, \mathfrak{i}, \mathbb{x})$, which adaptively answers oracle queries of the malicious verifier to prover oracles, *provided* the number of oracle queries is below the zero knowledge query bound and each query satisfies the query checker. In our setting, the number of oracle queries is bounded by the query complexity q of the *honest* AHP verifier, because the query set Q is derived via the honest query algorithm run on the messages sent by the malicious argument verifier. Moreover, the honest prover and simulator ensure that each query in Q satisfies the query checker. This explains why the zero knowledge query bound in Theorem 3.8.4 is q , and why we consider any *polynomial-time* query checker in Theorem 3.8.4.

Next, given that $\mathbf{S}(\mathbb{F}, \mathfrak{i}, \mathbb{x})$ provides oracle responses that are identically distributed to those of polynomials output by $\mathbf{P}(\mathbb{F}, \mathfrak{i}, \mathbb{x}, \mathbb{w})$, we are left to discuss the other information received by the malicious verifier: the commitments (in the first k rounds) and the evaluation proof (in round $k + 2$). The hiding property of the polynomial commitment scheme ensures that the simulator \mathcal{S}_{PC} , by using the trapdoor trap, can perfectly simulate these commitments and this evaluation proof. \square

3.9 MARLIN: an efficient preprocessing zkSNARK with universal SRS

We describe how to obtain a preprocessing zkSNARK with universal and updatable SRS that achieves the efficiency reported in Fig. 3.1.

The first step is to apply our compiler (Section 3.8) to two ingredients: the AHP described in Section 3.5, and the AGM-based polynomial commitment scheme described in Section 3.6.2 and the full version [CHMMVW20]. The second step is to apply the Fiat–Shamir transformation to the resulting public-coin preprocessing argument. These “generic” steps immediately yield a *preprocessing zkSNARK with universal and updatable SRS that has the same asymptotics as Sonic [MBKM19]*.¹¹ Moreover, in terms of concrete efficiency, this zkSNARK achieves argument size comparable to Sonic [MBKM19], and also achieves proving and verification times that are close to the state of the art for *circuit-specific* zkSNARKs [Gro16].

Below in Sections 3.9.1 and 3.9.2 we describe optimizations that further reduce argument size, and as a positive side effect also reduce prover and verifier costs. Fig. 3.1 includes these optimizations.

Before we discuss optimizations, we summarize the argument size that we obtain directly from the compilation mentioned above. Recall that in the offline phase, the AHP indexer, given an index $\mathfrak{i} = (\mathbb{F}, H, K, A, B, C)$, outputs for each matrix $M \in \{A, B, C\}$ three polynomials that together define the low-degree extension of M . Then, during the interactive online phase, the prover outputs twelve proof oracles. The verifier queries each of the nine indexer polynomials and the twelve prover polynomials at exactly one location, which amounts to 21 queries.

After compilation, the argument indexer outputs 9 polynomial commitments, and the argument prover outputs 12 commitments, 21 evaluations, and 3 evaluation proofs. In more detail, the argument indexer outputs commitments to $\mathit{row}_M, \mathit{col}_M, \mathit{val}_M$ for each $M \in \{A, B, C\}$; and the argument prover outputs commitments to the following twelve polynomials: $\hat{w}, \hat{z}_A, \hat{z}_B, \hat{z}_C, h_0, s, h_1, g_1, h_2, g_2, h_3, g_3$. The polynomials $\hat{w}, \hat{z}_A, \hat{z}_B, \hat{z}_C, h_0, s, h_1, g_1$ are all evaluated at the same point β_1 ; h_2 and g_2 are evaluated at the same point β_2 ; and $h_3, g_3, \mathit{row}_M, \mathit{col}_M, \mathit{val}_M$ for each $M \in \{A, B, C\}$ are all evaluated at the same point β_3 . Overall our argument consists of 27 \mathbb{G}_1 elements and 24 \mathbb{F}_q elements.

3.9.1 Optimizations for the AHP

Eliminating h_0 and \hat{z}_C . The AHP prover \mathbf{P} sends a polynomial $h_0(X)$ in the first round, and the AHP verifier \mathbf{V} checks the polynomial equation $\hat{z}_A(X)\hat{z}_B(X) - \hat{z}_C(X) = h_0(X)v_H(X)$ at a random point. This is a standard technique from the probabilistic proof literature to ensure that $\hat{z}_A(X)\hat{z}_B(X)$ and $\hat{z}_C(X)$ agree on H . An alternative (used, e.g., in [BCGRS17]) is to replace each

¹¹The SRS of the zkSNARK is updatable because the SRS of the polynomial commitment scheme is updatable (see Remark 3.8.2 and Section 3.6.2). Note also that the query algorithm in the AHP fulfills the admissibility requirement imposed by the polynomial commitment scheme (see Section 3.6.2), as each query location is sampled at random from a set of superpolynomial size.

occurrence of $\hat{z}_C(X)$ in the protocol with the product $\hat{z}_A(X)\hat{z}_B(X)$, which “forces” the desired property without any checks. This increases the degree of certain expressions by $\deg(\hat{z}_C) = |H| - 1$, but this cost in our setting is negligible because it leads to a negligible increase in the soundness error. This eliminates the need to commit to $h_0(X)$ and $\hat{z}_C(X)$ and later reveal their evaluations, which reduces argument size by two polynomial commitments and two field elements.

Minimal zero knowledge query bound. The query algorithm of the AHP verifier \mathbf{V} queries each prover polynomial at exactly one location, regardless of the randomness used to generate the queries. In particular, $\hat{w}(X), \hat{z}_A(X), \hat{z}_B(X), \hat{z}_C(X)$ are queried at exactly one location. So it suffices to set the parameter $b := 1$.

Eliminating σ_1 . We can sample the random polynomial $s(X)$ conditioned on it summing to zero on H . The prover can thus omit σ_1 , because it will always be zero, without affecting zero knowledge.

Single low-degree extension for each matrix (unimplemented). The AHP indexer \mathbf{I} constructs the low-degree extensions of the nine functions $\{\text{row}_M, \text{col}_M, \text{val}_M\}_{M \in \{A, B, C\}}$, which define the low-degree extensions of A, B, C . The AHP verifier \mathbf{V} queries each of these at a single location. This means that, after compilation, the argument prover must provide *nine* field elements (the evaluations) as part of the proof.

We can reduce this to only *three* field elements as follows. We modify the AHP indexer \mathbf{I} to construct, for each $M \in \{A, B, C\}$, a *single* low-degree extension of the functions $\text{row}_M, \text{col}_M, \text{val}_M$. Namely, let $s_1, s_2 \in \mathbb{F}$ be “shifts” such that $K, K + s_1$, and $K + s_2$ are pairwise disjoint, and define the set $\bar{K} := K \cup (K + s_1) \cup (K + s_2)$. Define the function $m_M: \bar{K} \rightarrow \mathbb{F}$ where

$$m_M(\kappa) := \begin{cases} \text{row}_M(\kappa) & \kappa \in K \\ \text{col}_M(\kappa - s_1) & \kappa \in K + s_1 \\ \text{val}_M(\kappa - s_2) & \kappa \in K + s_2 \end{cases} .$$

Then Eq. (3.1) can be rewritten as

$$\hat{M}(X, Y) := \sum_{\kappa \in K} u_H(X, \hat{m}_M(\kappa)) u_H(Y, \hat{m}_M(\kappa + s_1)) \hat{m}_M(\kappa + s_2) . \quad (3.10)$$

The modified AHP indexer \mathbf{I} constructs the three polynomials $\hat{m}_A, \hat{m}_B, \hat{m}_C$, and the modified AHP verifier \mathbf{V} will query each of these at a single location. Thus, after compilation, the argument prover will only need to provide three field elements, instead of nine, as part of the proof. Note that this optimization triples the degree of the polynomials output by the AHP indexer \mathbf{I} , which after compilation increases the SRS size. Even given this tradeoff our SRS is still shorter than prior work, and furthermore it represents a one-time offline cost (in contrast to argument size, which is a recurring online cost).

A more efficient holographic lincheck. Based on an earlier draft of this work, Chiesa, Ojha, and Spooner [COS20b] devised a more efficient holographic lincheck, which saves one round of interaction, in a different model of proof system. In the full version [CHMMVW20] we show how to incorporate their ideas into our 7-message AHP for R1CS (see Fig. 3.5) to obtain a 5-message AHP for R1CS (see the full version [CHMMVW20]). This optimization has no downsides.

3.9.2 Optimizations for the polynomial commitment scheme

Reducing the cost of hiding commitments. The hiding property that we adopt for polynomial commitments (Definition 3.6.4) ensures that no information is revealed about the committed polynomial regardless of how many evaluations are revealed. Achieving this strong notion has a cost: in our constructions we randomize a commitment c to a polynomial p by additionally committing to a random polynomial \bar{p} of degree $\deg(p)$. Compared to the non-hiding variant, this requires $\deg(p)$ additional elements in the SRS, and also requires PC.Commit and PC.Open to perform an additional variable-base MSM of size $\deg(p)$.

In our compiler, however, the only evaluations the argument verifier sees are those sent by the argument prover, and these are determined by the query sets produced by the query algorithm. This, together with the fact in our AHP each polynomial is queried at exactly one location, implies that we can relax our construction to provide hiding only for a single evaluation per polynomial. Concretely, we can set \bar{p} to have degree 1. (Note that \bar{p} cannot be a constant because it is used to hide both the commitment to p and to hide the commitment to the witness polynomial w .) This allows us to eliminate (most of) the additional generators from the SRS, and the additional variable-base MSM for PC.Commit and PC.Open.

Reducing the number of hiding commitments. Each hiding commitment, even taking into account the above optimizations, requires an evaluation proof that is one field element larger than a proof in the non-hiding case. We reduce this overhead by using the fact that only certain polynomials reveal information about the witness and necessitate hiding. In particular, only the polynomials \hat{w} , \hat{z}_A , \hat{z}_B , \hat{z}_C , s , h_1 , and g_1 need hiding commitments. All other polynomials can rely on non-hiding commitments because they can be derived in polynomial-time from the index i . This observation removes a further 1 field element from the proof.

Eliminating unnecessary degree checks. The notion of polynomial commitment scheme that we consider enables each commitment to guarantee a chosen degree bound that is up to the maximum degree bound chosen for the SRS. This flexibility has a cost: ensuring a degree bound strictly less than the maximum degree bound requires two group elements per commitment, corresponding to unshifted and shifted polynomials respectively. When compiling our AHP, we need this feature only when committing to g_1, g_2, g_3 (the exact degree bound matters for soundness) but for all other polynomials it suffices to rely on the maximum degree bound and so for them we omit the shifted polynomials altogether. This increases the soundness error by a negligible amount (which is fine), and lets us reduce argument size by 9 group elements.

Batching pairing equations. We can reduce the cost of the argument verifier by batching pairing equations. Recall that, to verify an evaluation proof with evaluation v and point z , PC.Check needs to check the pairing equation $e(U - vG - \gamma\bar{v}G, H) = e(w, \beta H - zH)$. In our compiled zkSNARK, PC.Check is invoked three times, each with different values of U , w , z , and v . This results in 3 pairing equations. To reduce the number of pairing equations needed down to just one, we use the

following reduction that ensures that the \mathbb{G}_2 argument to every pairing is constant:

$$\begin{aligned} e(U - vG - \gamma\bar{v}G, H) &= e(\mathbf{w}, \beta H - zH) \\ &= e(\mathbf{w}, \beta H) \cdot e(\mathbf{w}, -zH) \\ &= e(\mathbf{w}, \beta H) \cdot e(-z\mathbf{w}, H) . \end{aligned}$$

Hence, we have that

$$e(U - vG - \gamma\bar{v}G + z\mathbf{w}, H) = e(\mathbf{w}, \beta H) .$$

Because we have three proofs to check, the verifier has to check three of the above equations. These equations can be batch verified together as follows. The verifier samples a random field element r , and then uses the identity $\prod_i e(G_i, H)^{r^i} = e(\sum_i r^i G_i, H)$ to check the following equation:

$$e(\sum_i r^i (C_{0,i} - v_i G - \gamma\bar{v}_i G + z_i \mathbf{w}_i), H) = e(\sum_i r^i \mathbf{w}_i, \beta H) .$$

By properties of random linear combinations, the above equation holds only if each of the individual equations also hold (up to a negligible soundness error). In sum, the verifier only needs to evaluate two pairings.

Opening linear combinations of polynomials. The decision procedure of the AHP verifier checks polynomial equations such as

$$p_1(X) + p_2(X)p_3(X) = p_4(X) . \quad (3.11)$$

It does so by querying the polynomials p_1, \dots, p_4 at a random point $z \in \mathbb{F}$, and then checking that the above equation holds with respect to the resulting evaluations $p_1(z), \dots, p_4(z)$. To enable the compiled SNARK verifier to invoke the AHP decision procedure, the SNARK proof must also contain these evaluations. However, if we instead enable the AHP verifier to query *linear combinations* of polynomial oracles, then one can avoid providing all these evaluations. For example, we can rewrite the check in Equation (3.11) as follows:

$$p_2(z) = v_2 \quad \text{and} \quad p_5(X) := p_1(X) + v_2 p_3(X) - p_4(X) = 0 .$$

Then the AHP decision procedure only needs the evaluation $p_2(z)$, which means that the corresponding SNARK proof will contain only 1 field element, instead of 4. For this, we need that the polynomial commitment scheme allows checking evaluations of linear combinations of committed polynomials. The schemes constructed in the full version[CHMMVW20] have linearly homomorphic commitments, and so support this feature.

Applying this optimization to the equations in our AHP reduces the proof size of the corresponding compiled SNARK by 10 field elements.

Chapter 4

Gemini: Elastic SNARKs for Diverse Environments

We introduce and study *elastic SNARKs*, a class of succinct arguments where the prover has multiple configurations with different time and memory tradeoffs, which can be selected depending on the execution environment and the proved statement. The output proof is independent of the chosen configuration.

We construct an elastic SNARK for rank-1 constraint satisfiability (R1CS). In a time-efficient configuration, the prover uses a linear number of cryptographic operations and a linear amount of memory. In a space-efficient configuration, the prover uses a quasilinear number of cryptographic operations and a logarithmic amount of memory. A key component of our construction is an elastic probabilistic proof. Along the way, we also formulate a streaming framework for R1CS that we deem of independent interest.

We additionally contribute Gemini, a Rust implementation of our protocol. Our benchmarks show that Gemini, on a single machine, supports R1CS instances with tens of billions of constraints.

This work was previously published in [BCHO22].

4.1 Introduction

Succinct non-interactive arguments of knowledge (SNARKs) allow for efficient verification of NP statements. Recent years have seen a surge of interest in SNARKs, catalyzed by several real-world applications. While reducing argument size and verification time were an initial focus, the cost of running the prover algorithm has now emerged as a critical bottleneck. This is particularly important as the size of proved computations increases, and recent applications demand proving large computations.

For example, popular scaling solutions for blockchains (*roll-up architectures*) require regularly producing SNARKs attesting to the validity of large batches of transactions, which translates to proving the correctness of billions of gates. As another example, the Filecoin network generates proofs for about 930 billion constraints every day.¹ In both cases, efficiently producing SNARKs attesting to the correctness of large computations is critical, yet many SNARK implementations today do not scale to large computations because of the prohibitive memory requirements of the proving algorithm. Indeed, research that focuses on the time complexity of the prover algorithm has achieved notable theoretical and practical improvements [BCGGHJ17; BCGJM18; XZZPS19; Set20; BCG20; Lee20; KMP20; Zha+21; BCL22; GLSTW21; RR22], but with linear space complexity. These constructions rely on, among other things, a component that achieves linear-time proving via dynamic programming techniques [Tha13], which demands storing in memory the proved computation.

The notion of *complexity-preserving SNARKs* introduced in [BC12] aims to *simultaneously* optimize time and space: it requires that the prover’s time and space complexity are at most polylogarithmic factors away from those of the proved computation. Complexity-preserving SNARKs were subsequently studied (and improved) in a line of works [BCCT13; HR18; BHRS20; BHRS21]. The holy grail would be to preserve time and space complexity up to constant factors, but known constructions are far from this goal, and achieve improved space complexity at the expense of time complexity.

In sum, a line of works achieves excellent time complexity at the expense of space complexity, and a different line of works achieves excellent space complexity at the expense of time complexity. It remains a challenging open question to construct SNARKs that simultaneously do well in both parameters. In this paper we do not answer this question, but instead introduce and achieve a notion that meaningfully relaxes this goal: a single SNARK that can be configured to optimize for time complexity *or* optimize for space complexity.

4.1.1 Our results

(1) Elastic SNARKs. We advocate the study of SNARKs whose prover admits two different realizations:

- a *time-efficient* prover that receives as input instance and witness;
- a *space-efficient* prover with *streaming* access to these same inputs.

¹<https://research.protocol.ai/sites/snarks/>

These *elastic* provers can choose which realization to use, and allocate resources depending on the execution environment and the instance size. In addition, the two algorithms are compatible in such a way that during the execution of the protocol the space-efficient prover can pause and transcribe a prover state, and then the protocol can continue with the time-efficient prover (thereafter enjoying the benefits of the faster prover).

We build on the notion of streams in [BHRRS20] to study the above goal. We study stream composition, and propose a definitional framework for streaming instances of Rank-1 Constraint Satisfiability (R1CS). Within this framework we contribute an elastic SNARK for R1CS that we describe next.

(2) An elastic SNARK for R1CS. We realize the above notion by constructing an elastic (preprocessing) SNARK for R1CS, that we name Gemini. In time-efficient mode the prover uses a linear number of cryptographic operations and linear space, and in space-efficient mode the prover uses a quasilinear number of cryptographic operations and logarithmic space. When referring to time efficiency, we use the asymptotic notation O_λ to denote cryptographic operations, so to distinguish them from (less expensive) field operations for which we use the asymptotic notation O . Our main result is the (informally stated) theorem below.

Definition 2. *The R1CS problem asks: given a finite field \mathbb{F} , coefficient matrices $A, B, C \in \mathbb{F}^{N \times N}$ each containing at most $M = \Omega(N)$ non-zero entries,² and an instance vector \mathbf{x} over \mathbb{F} , is there a witness vector \mathbf{w} such that $A\mathbf{z} \circ B\mathbf{z} = C\mathbf{z}$, for $\mathbf{z} := (\mathbf{x}, \mathbf{w}) \in \mathbb{F}^N$? (Here, “ \circ ” denotes the entry-wise product.)*

Theorem 3 (informal). *There exists an elastic SNARK for $\mathcal{R}_{\text{R1CS}}$ whose prover admits two realizations:*

- a time-efficient prover that runs in $O_\lambda(M)$ time and $O(M)$ space;
- a space-efficient prover that runs in $O_\lambda(M \log^2 M)$ time and $O(\log M)$ space.

Verification time is $O_\lambda(|\mathbf{x}| + \log M)$ time and proof size is $O(\log M)$.

The above SNARK is obtained via a popular paradigm that combines a polynomial IOP and a polynomial commitment scheme in order to obtain an interactive argument, and then relies on the Fiat–Shamir paradigm to make the protocol non-interactive. The (omitted) cryptographic assumptions in the informal statement are inherited from those for the underlying polynomial commitment scheme, which in our case is [KZG10].

Briefly, after observing that the polynomial commitment scheme in [KZG10] can be realized elastically, our main contribution is achieving an elastic polynomial IOP. A key component of this latter is an elastic scalar product protocol, which runs in linear-time and linear-space or quasilinear-time and log-space. Our scalar product argument is based on the sumcheck protocol, which, thanks to its recursive nature, facilitates migrating from a space-efficient instance to a time-efficient one.

²Note that $M = \Omega(N)$ without loss of generality because if $M < N/3$ then there are variables of \mathbf{z} that do not participate in any constraint, which can be dropped. Thus the main size measure for R1CS is the sparsity parameter M .

(3) Implementation. We implement the construction of Theorem 3 in Rust using the arkworks ecosystem [ark], replicating the modularity of the IOP construction. In particular, we develop new streaming-friendly primitives that we believe could be of independent interest for future projects realizing space-efficient cryptographic proofs. Our implementation additionally includes a simpler SNARK that is *not* preprocessing (the verification procedure reads R1CS instances without providing succinct verification). In Section 4.2.7, we summarize our design choices and algorithmic optimizations for the implementation.

(4) Evaluation. Most benchmarks for time-efficient SNARKs in the literature do not consider large circuits, due to prohibitive memory usage. Our benchmarks, discussed further in Section 4.2.8, show the following.

- Gemini is able to prove instances of arbitrary size. On a single machine with a memory budget of around 1 GB, we ran the prover of the preprocessing SNARK for instances of size 2^{32} and the prover of the non-preprocessing SNARK (where the verifier reads the R1CS instance in full) for instances of size 2^{35} . We “stopped” at these sizes only due to time constraints.

In contrast, the largest instance size reported in the literature is in DIZK [WZCPS18], where a distributed realization of the preprocessing SNARK (with circuit-specific setup) of [Gro16] is run for an R1CS instance of size 2^{31} over a cluster of 20 machines with 256 executors.

- Gemini is concretely and economically efficient. The preprocessing SNARK can prove instances of size 2^{31} in two days and costs about 82% (about 400 USD) less than DIZK on Amazon EC2.
- Gemini provides succinct proofs and verification. For instances of size 2^{35} , the proof size is about 27 KB and the verification time is below 30 ms.

4.2 Techniques

We summarize the main ideas behind our results. In Section 4.2.1 we outline the streaming model that we use to express space-efficient algorithms. In Section 4.2.2 we describe how to construct elastic SNARKs from elastic polynomial commitment schemes and elastic probabilistic proofs. We describe an example of an elastic polynomial commitment scheme in Section 4.2.3. Then, in Sections 4.2.4 to 4.2.6 we sketch our elastic probabilistic proof. We conclude in Section 4.2.7 and Section 4.2.8 by discussing our implementation and evaluation.

4.2.1 Elasticity and a streaming model

The notion of elasticity refers to having multiple realizations of the same algorithm (more precisely, function) for use in different situations. Specifically in this work:

Elasticity means that we aim for two realizations: a **time-efficient realization** for a setting where time complexity is most important, possibly at the expense of space complexity; and a **space-efficient realization** for a setting where space complexity (i.e., memory consumption) is most important, possibly at the expense of time complexity.

This means that in theorem statements, and in their proofs, we will consider two realizations with different complexities for the same function (e.g., the SNARK prover).

Time-efficient algorithms are a familiar concept. Space-efficient algorithms in this paper are *streaming algorithms*: algorithms that receive their inputs in *streams* (small pieces at a time) so that they can use less memory than the size of their inputs. Below we elaborate on: (i) streams; and (ii) streaming algorithms.

Streams and streaming oracles. A *stream* is a sequence $K \in \Sigma^I$, where Σ is an alphabet and I is a well-ordered countable set. Streams are accessed via oracles: if K is a sequence, the *streaming oracle* $\mathcal{S}(K)$ of K takes two input commands, *start* and *next*; the oracle responds to the i -th *next* command with the i -th element of K ; in case earlier elements of the stream need to be read again, the *start* command resets the oracle to the first element in the sequence. The oracle does *not* allow random access to elements of K .

Streaming algorithms. A *streaming algorithm* is an algorithm that has access to its inputs via streaming oracles and produces a stream as its output, by outputting the next element upon receiving the next command. The complexity of a streaming algorithm is measured in terms of its time complexity, space complexity, and the number of passes that it makes over each input stream (via the *start* command).

Any binary operation over an alphabet can be viewed as a streaming algorithm which takes as input two sequences K and K' over the same alphabet Σ that are indexed by the same set I . The binary operation acts on successive pairs of elements of K and K' , to produce a new stream on the fly. For instance, let \mathbf{f}, \mathbf{g} be two vectors over a finite field \mathbb{F} , and $\mathcal{S}(\mathbf{f}), \mathcal{S}(\mathbf{g})$ their canonical streams. (The canonical stream of a vector is the sequence of its entries, from last to first.) For a scalar $\rho \in \mathbb{F}$, the stream $\mathcal{S}(\mathbf{f} + \rho\mathbf{g})$ can be evaluated as a new stream using $\mathcal{S}(\mathbf{f})$ and $\mathcal{S}(\mathbf{g})$, by responding to each

next query in the following way: first query $\mathcal{S}(\mathbf{f})$ to obtain the i -th entry f_i of \mathbf{f} ; then query $\mathcal{S}(\mathbf{g})$ to obtain the i -th entry g_i of \mathbf{g} ; and finally respond with $f_i + \rho g_i$.

Since a streaming algorithm produces a stream as output, multiple streaming algorithms can be *composed* so that the output stream produced by one algorithm is the input stream for the next algorithm. The time and space complexity and number of input passes of streaming algorithms behave predictably under composition. If \mathcal{A} is a streaming algorithm with time complexity $t_{\mathcal{A}}$, space complexity $s_{\mathcal{A}}$, and $k_{\mathcal{A}}$ input passes, and \mathcal{B} is a streaming algorithm with time complexity $t_{\mathcal{B}}$, space complexity $s_{\mathcal{B}}$, and $k_{\mathcal{B}}$ input passes, then \mathcal{A} composed with \mathcal{B} has time complexity $t_{\mathcal{A}} + k_{\mathcal{A}}t_{\mathcal{B}}$, space complexity $s_{\mathcal{A}} + s_{\mathcal{B}}$, and $k_{\mathcal{A}}k_{\mathcal{B}}$ input passes.

4.2.2 A modular construction of elastic SNARKs

Many succinct arguments are built in two steps. First, construct an information-theoretic probabilistic proof in a model where the verifier has a certain type of query access to the prover’s messages. Second, compile the probabilistic proof into an interactive succinct argument, via a cryptographic commitment scheme that “supports” this query access.³ Finally, if non-interactivity is desired, apply the Fiat–Shamir transformation [FS86]. This modular approach has enabled researchers to study the efficiency and security of simpler components, which has facilitated much progress in succinct arguments.

We observe that the approach used in [CHMMVW20; BFS20] to construct SNARKs *preserves elasticity*: if the ingredients to the approach are elastic then the resulting SNARK is elastic. There are two ingredients.

- *Polynomial IOPs.* A probabilistic proof in which the prover sends polynomial oracles to the verifier, who accesses them via polynomial evaluation queries. This is an interactive oracle proof [BCS16; RRR16] where query access to prover messages is changed from “point queries” to “polynomial evaluation queries”.
- *Polynomial commitments.* A cryptographic primitive that enables a sender to commit to a polynomial $\mathbf{f} \in \mathbb{F}[X]$ of bounded degree, and later prove that $\mathbf{f}(z) = v$ for given $z, v \in \mathbb{F}$.

If the polynomial IOP is additionally *holographic* then the resulting succinct argument is a *preprocessing argument*, which means that it is possible, in an offline phase, to perform a public computation that enables sub-linear verification in a later online phase. The lemma below summarizes how elasticity is preserved. The formal statement (and its proof) are relative to the formalism for streaming algorithms outlined in Section 4.2.1.

Theorem 4 (informal). *Suppose that we are given the following ingredients.*

³The argument prover and argument verifier emulate the underlying probabilistic proof, with the argument prover sending commitments to proof messages and sending answers to queries together with commitment openings to authenticate those answers.

- A public-coin polynomial IOP for a relation \mathcal{R} with: (i) time-efficient prover time $t_{\mathcal{P}}$; (ii) space-efficient prover space $s_{\mathcal{P}}$ with $k_{\mathcal{P}}$ passes; (iii) O oracles; (iv) query complexity q ; (v) verifier complexity $t_{\mathcal{V}}$.
- A polynomial commitment scheme with: (i) time-efficient commit (and open) time $t_{\text{PC.Com}}$; (ii) space-efficient commit (and open) space $s_{\text{PC.Com}}$ with $k_{\text{PC.Com}}$ passes; (iii) checking time $t_{\text{PC.Check}}$.

Then there exists an interactive argument system for the relation \mathcal{R} with: (i) time-efficient prover time $t_{\mathcal{P}} + O \cdot t_{\text{PC.Com}} + q \cdot t_{\text{PC.Com}}$; (ii) space-efficient prover space $s_{\mathcal{P}} + O \cdot s_{\text{PC.Com}}$ with $q \cdot k_{\mathcal{P}} \cdot k_{\text{PC.Com}}$ passes; (iii) verifier complexity $t_{\mathcal{V}} + q \cdot t_{\text{PC.Check}}$. Moreover, the argument system is preprocessing if the given polynomial IOP is holographic (with time and space properties similarly preserved by the transformation).

Informally, the argument prover commits to each polynomial oracle via the polynomial commitment scheme, and answers polynomial evaluation queries by sending the requested evaluation along with a proof that it is consistent with the corresponding polynomial commitment. The security and most efficiency measures are studied in [CHMMVW20; BFS20]. Less obvious is how space complexity is affected.

A streaming implementation of the PIOP prover does not necessarily produce all of its output polynomial streams one by one, and therefore the space complexity of the resulting argument prover is not, e.g., just the sum $s_{\mathcal{P}} + s_{\text{PC.Com}}$ of the PIOP prover space and the PC commitment algorithm space. When two (or more) of the PIOP prover’s message polynomials all depend on the same input stream, the prover may avoid extra passes over the input stream by producing both of them at the same time which would require space $s_{\mathcal{P}} + 2s_{\text{PC.Com}}$.⁴ Furthermore, the commitment algorithm requires several passes over a single input polynomial, so that the argument prover must run the PIOP prover several times in order to complete the commitment to each polynomial, keeping partially computed commitments to each polynomial in memory. Such considerations lead to the space-efficient argument prover having space complexity $s_{\mathcal{P}} + O \cdot s_{\text{PC.Com}}$ with $q \cdot k_{\mathcal{P}} \cdot k_{\text{PC.Com}}$ passes. Fortunately, the PIOP constructions in this paper actually satisfy the strong property that each polynomial can be produced independently without rerunning the entire prover algorithm, which reduces the space complexity to $s_{\mathcal{P}} + s_{\text{PC.Com}}$.

Remark 4.2.1 (types of polynomials). The above discussion is deliberately ambiguous about certain aspects: are the polynomials univariate or multivariate? are the polynomials represented as vectors of coefficients or as vectors of evaluations (or vectors in some other basis)? These details do not matter for Theorem 4 as long as the two ingredients “match up”: if the PIOP outputs polynomials represented in a way that is compatible with how the PC scheme expects inputs. Nevertheless, in this paper we focus on the case of univariate polynomials represented as vectors of coefficients, because our construction and implementation are in this setting.

⁴For example, if one polynomial consists of all of the even coefficients of another, one can produce streams of the coefficients of both polynomials simultaneously, in half the number of passes required to compute streams of each polynomial one at a time.

Remark 4.2.2 (multilinear vs. univariate). The fact that the approach in [CHMMVW20; BFS20] preserves space efficiency in the case of multilinear polynomials represented over the boolean hypercube was used in [BHRRS20; BHRRS21]. Theorem 4 is a straightforward observation about [CHMMVW20; BFS20] that additionally preserves elasticity. In particular, we believe that the constructions in [BHRRS20; BHRRS21] could be shown to have elastic realizations, by showing that the underlying multilinear PIOP and multilinear PC schemes have elastic realizations. We choose to work with univariate polynomials, instead of multilinear polynomials, because they have received more interest by practitioners, and thus focus our investigation on the concrete efficiency of elastic SNARKs based on univariate polynomials. We leave the study of concrete efficiency of elastic SNARKs based on multilinear polynomials to future work.

Remark 4.2.3 (elastic setup and indexer). For any succinct argument, elasticity is a desirable property as the size of the statement to be proven increases. In this paper we focus on elasticity of the prover, which is the main bottleneck for proving large instances. We briefly comment on elasticity for other algorithms.

- *Setup*. The setup algorithm samples the public parameters of the argument system. While the complexity of the setup algorithm can be linear (or more!) in the statement size, we do not discuss setup algorithms in this paper for two reasons: (i) known setup algorithms have straightforward realizations that are simultaneously efficient in time and space (there is less of a tension between optimizing for time or for space as there is for the prover); (ii) public parameters are typically sampled via “cryptographic ceremonies” that realize the setup functionality via secure multi-party protocols [BGM17], and so it is more relevant to discuss the time and space efficiency of the protocols that realize these ceremonies.
- *Indexer*. In the case of preprocessing arguments, an indexer algorithm produces the *proving key* and *verification key*. The indexer in our construction and implementation is elastic, but we do not discuss it since all ideas relevant for the indexer can be straightforwardly inferred from those relevant for the prover.

4.2.3 An elastic realization of the KZG polynomial commitment scheme

We use a univariate polynomial commitment scheme from [KZG10] to construct our SNARK (see Section 4.2.2). Below we review this scheme and explain how to realize it elastically.

Review: a polynomial commitment from [KZG10]. The setup algorithm samples and outputs public parameters for the scheme to support polynomials of degree at most $D \in \mathbb{N}$: the description of a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, G, H, e)$;⁵ the commitment key $\text{ck} := (G, \tau G, \dots, \tau^D G) \in \mathbb{G}_1^{D+1}$ for a random field element $\tau \in \mathbb{F}_q$; and the receiver key $\text{rk} := (G, H, \tau H) \in \mathbb{G}_1 \times \mathbb{G}_2^2$. The commitment to a polynomial $\mathbf{p} \in \mathbb{F}_q[X]$ of degree $d \leq D$ is computed as $C := \langle \mathbf{p}, \text{ck} \rangle = \mathbf{p}(\tau)G \in \mathbb{G}_1$. Subsequently, to prove that the committed polynomial \mathbf{p} evaluates to v at $z \in \mathbb{F}_q$, the committer computes the witness polynomial $\mathbf{w}(X) := (\mathbf{p}(X) - \mathbf{p}(z))/(X - z)$, and outputs the evaluation

⁵Here $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T| = q$, G generates \mathbb{G}_1 , H generates \mathbb{G}_2 , and $e: \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is a non-degenerate bilinear map.

proof $\pi := \langle \mathbf{w}, \text{ck} \rangle = \mathbf{w}(\tau)G \in \mathbb{G}_1$. Finally, to verify the evaluation proof, the receiver checks that $e(C - vG, H) = e(\pi, \tau H - zH)$.

Elastic realization. An elastic realization of the above scheme requires a time-efficient realization and a space-efficient realization for each relevant algorithm of the scheme. Here we do not discuss the setup algorithm, as it has a natural time-and-space-efficient realization (see Remark 4.2.3). We do not discuss the verification algorithm either, because it only involves a constant number of scalar multiplications and pairings. Our focus is thus on the commitment and opening algorithm.

- *Commitment algorithm.* For $d \leq D$, we are given streams of the commitment key elements $(G, \tau G, \dots, \tau^d G)$ and of the coefficients $(p_i)_{i=0}^d$ of the polynomial $\mathbf{p}(X) = \sum_{i=0}^d p_i X^i$ to be committed. We compute the commitment $C = \sum_{i=0}^d p_i \tau^i G$ by multiplying each coefficient-key pair $(p_i, \tau^i G)$ together and adding them to a running total. Each scalar-multiplication of $p_i \cdot \tau^i G$ is performed in linear time and constant space.
- *Opening algorithm.* We are given the same streams as above, and an opening location z . By rearranging the expression for the witness polynomial $\mathbf{w}(X) = (\mathbf{p}(X) - \mathbf{p}(z))/(X - z)$, we stream the coefficients $(w_i)_{i=0}^{d-1}$ of $\mathbf{w}(X)$ via Ruffini's rule: $w_i := p_{i+1} + w_{i+1}z$. The evaluation proof $\pi = \sum_{i=0}^{d-1} w_i \tau^i G$ is computed in the same way as the commitment algorithm.

We discuss optimizations on the above streaming approach in Section 4.2.7.2.

Note that the recurrence relation in the opening algorithm uses w_{j+1} to compute w_j , which means that $\mathbf{w}(X)$ is computed from its highest-degree coefficient to its lowest-degree coefficient. In turn, this means that the commitment key ck and the polynomial $\mathbf{p}(X)$ are streamed from highest-degree to lowest-degree coefficient. The setup and commitment algorithms are agnostic to the streaming order.

The above discussion implies the following (informal) lemma.

Lemma 4.2.4 (informal). *The polynomial commitment scheme of [KZG10] has an elastic realization.*

4.2.4 An elastic scalar-product protocol

A scalar-product protocol enables the prover to convince the verifier that the scalar product of two committed vectors equals a certain target value. Many constructions of succinct arguments for NP crucially rely on scalar-product protocols [BCCGP16; PLS19; BCG20]. The PIOP for R1CS that we construct in Sections 4.2.5 and 4.2.6 relies on a PIOP for scalar products where the prover has two realizations: (i) one that runs in linear-time and linear-space; and (ii) one that runs in quasilinear-time and logarithmic-space.

Definition 3. *A PIOP for scalar products is a PIOP where the verifier receives as input (\mathbb{F}, N, u) and has (polynomial evaluation) query access to $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$, and checks with the help of the prover that $\langle \mathbf{f}, \mathbf{g} \rangle = u$.*

Theorem 4.2.5 (informal). *For every finite field \mathbb{F} , there is a PIOP for scalar products over \mathbb{F} with the following parameters:*

- *soundness error* $O(N/|\mathbb{F}|)$;
- *round complexity* $O(\log N)$;
- *proof length* $O(N)$ and *query complexity* $O(\log N)$;
- *a time-efficient prover that runs in time* $O(N)$ and *space* $O(N)$;
- *a space-efficient prover that runs in time* $O(N \log N)$ and *space* $O(\log N)$ (with $O(\log N)$ input passes);
- *a verifier that runs in time* $O(\log N)$ and *space* $O(\log N)$.

Below we outline the scalar-product protocol, deferring to Section 4.6 security proofs and a more in-depth discussion of the protocol. We also note that our PIOP uses two slightly different protocols: one for *twisted scalar-products* $\langle \mathbf{f} \circ \mathbf{y}, \mathbf{g} \rangle = u$ for a vector \mathbf{y} of the form $(1, \rho_0) \otimes (1, \rho_1) \otimes \cdots \otimes (1, \rho_{n-1})$ where $n := \log N$ (by \log we denote the ceiling of the logarithm base 2); and one for Hadamard products $\mathbf{f} \circ \mathbf{g} = \mathbf{h}$. These follow from simple modifications to the scalar-product protocol.

We proceed in three steps. In Section 4.2.4.1 we describe how to reduce checking a scalar product to checking tensor products of univariate polynomials. In Section 4.2.4.2 we describe a tensor product protocol. In Section 4.2.4.3 we describe how to realize this latter protocol in an elastic way.

4.2.4.1 Verifying scalar products using the sumcheck protocol

Consider two vectors $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$ with $\langle \mathbf{f}, \mathbf{g} \rangle = u$ as in Definition 3. The verifier has polynomial evaluation query access to \mathbf{f} and \mathbf{g} (the verifier can obtain any evaluations of the polynomials $\mathbf{f}(X) = \sum_{i=0}^{N-1} f_i X^i$ and $\mathbf{g}(X) = \sum_{i=0}^{N-1} g_i X^i$). The product polynomial $\mathbf{h}(X) := \mathbf{f}(X) \cdot \mathbf{g}(X^{-1})$ has $\langle \mathbf{f}, \mathbf{g} \rangle = \sum_{i=0}^{N-1} f_i g_i$ as the coefficient of X^0 , because for every $i, j \in [N]$ the powers of X associated with f_i and g_j multiply together to give X^0 if and only if $i = j$. Therefore, to check the scalar-product $\langle \mathbf{f}, \mathbf{g} \rangle = u$, it suffices to check that the coefficient of X^0 in the product polynomial $\mathbf{h}(X)$ equals u .

However, this check must somehow be performed *without the prover actually computing* $\mathbf{h}(X)$. This is because the fastest algorithm for computing $\mathbf{h}(X)$ requires $O(N \log N)$ time and $O(N)$ space (via FFTs), which is neither time-efficient nor space-efficient. On the other hand, the scalar product $\langle \mathbf{f}, \mathbf{g} \rangle = u$ can be checked (directly) in time $O(N)$ and space $O(1)$, which leaves open the possibility of a scalar-product protocol where the prover does better than computing $\mathbf{h}(X)$ explicitly (and then running some protocol).

This issue is addressed in prior work, if the verifier can query the *multilinear* polynomials $\widehat{\mathbf{f}}(\mathbf{X})$ and $\widehat{\mathbf{g}}(\mathbf{X})$ associated to the vectors $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$: we index the entries of \mathbf{f} using binary vectors, and $f_i = f_{b_0, \dots, b_{n-1}}$ is the coefficient of $X_0^{b_0} \cdots X_{n-1}^{b_{n-1}}$, where (b_0, \dots, b_{n-1}) is the binary decomposition of i . Prior work [Tha13; XZZPS19; BCG20] yields the following lemma.

Lemma 4.2.6. *Let \mathbb{F} be a finite field and N be a positive integer; set $n := \log N$. Let $\widehat{\mathbf{f}}(X_0, \dots, X_{n-1})$ and $\widehat{\mathbf{g}}(X_0, \dots, X_{n-1})$ be multilinear polynomials. The sumcheck protocol (as a reduction to claims*

about polynomial evaluations) for the claim

$$\frac{1}{2^n} \sum_{\omega \in \{-1,1\}^n} (\hat{\mathbf{f}} \cdot \hat{\mathbf{g}})(\omega) = u$$

has the following properties: soundness error is $O(\log N/|\mathbb{F}|)$; round complexity is $O(\log N)$; prover time $O(N)$; and verifier time $O(\log N)$.

One can use the (multivariate) sumcheck protocol of [LFKN92] to reduce $\langle \mathbf{f}, \mathbf{g} \rangle = u$ to two evaluation queries $\hat{\mathbf{f}}(\boldsymbol{\rho})$ and $\hat{\mathbf{g}}(\boldsymbol{\rho})$, where $\boldsymbol{\rho} := (\rho_0, \dots, \rho_{n-1}) \in \mathbb{F}^n$ are the random verifier challenges used in the sumcheck protocol. Crucially, the prover algorithm in the sumcheck protocol applied to the product of two multilinear polynomials also has a space-efficient realization which runs in time $O(N \log N)$ and space $O(\log N)$ [CMT12], which would provide an elastic solution in this multilinear regime.

In our setting the verifier can only query the *univariate* polynomials $\mathbf{f}(X)$ and $\mathbf{g}(X)$ associated with the vectors $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$. Nevertheless, we follow a similar approach, by running the sumcheck protocol on the *multivariate* polynomials $\hat{\mathbf{f}}(\mathbf{X})$ and $\hat{\mathbf{g}}(\mathbf{X})$, producing two claimed evaluations $\hat{\mathbf{f}}(\boldsymbol{\rho}) = u$ and $\hat{\mathbf{g}}(\boldsymbol{\rho}) = u'$. We check that these claimed evaluations are consistent with \mathbf{f} and \mathbf{g} using evaluations of the *univariate* polynomials $\mathbf{f}(X)$ and $\mathbf{g}(X)$ in the *tensor product protocol* of the following section.

Remark 4.2.7 (unstructured fields). Many probabilistic proofs using univariate polynomials (e.g., the low-degree test in [BBHR18]) require the size (of the multiplicative group) of the field \mathbb{F} to be *smooth*, so that the field contains high-degree roots of unity. In contrast, the scalar-product protocol in this paper (indeed, all the PIOPs in this paper) work with univariate polynomials over *any* field \mathbb{F} that is sufficiently large.

4.2.4.2 A tensor-product protocol

We seek a protocol for checking the multilinear evaluation $\hat{\mathbf{f}}(\boldsymbol{\rho}) = v$ while having access to $\mathbf{f}(X)$ (and possibly other polynomials sent by the prover) via univariate polynomial evaluations. Observe that $\hat{\mathbf{f}}(\mathbf{X})$ and $\mathbf{f}(X)$ have the same coefficients, and moreover the polynomial $\hat{\mathbf{f}}(\rho_0, X_1, \dots, X_{\log N-1})$ (partially evaluating $\hat{\mathbf{f}}(\mathbf{X})$ by setting X_0 equal to ρ_0) has the same coefficients as the polynomial $\mathbf{f}'(X) := \mathbf{f}_e(X) + \rho_0 \cdot \mathbf{f}_o(X)$. Here, $\mathbf{f}_e(X)$ and $\mathbf{f}_o(X)$ are the odd and even parts defined by $\mathbf{f}(X) = \mathbf{f}_e(X^2) + X\mathbf{f}_o(X^2)$.

This suggests a protocol where the prover sends $\mathbf{f}'(X)$ to the verifier. If the verifier can check that $\mathbf{f}'(X)$ was correctly computed from $\mathbf{f}(X)$, then checking consistency between $\mathbf{f}(X)$ and an evaluation of $\hat{\mathbf{f}}(X_0, \dots, X_{\log N-1})$ is reduced to checking consistency between $\mathbf{f}'(X)$ and an evaluation of $\hat{\mathbf{f}}(\rho_0, X_1, \dots, X_{\log N-1})$. Repeating this reduction with every value ρ_j , the prover and verifier arrive at a claim about constant-degree polynomials, which the prover can send to the verifier and the verifier directly checks.

To check that $\mathbf{f}'(X)$ is consistent with $\mathbf{f}(X)$, the verifier samples a random challenge point $\beta \in \mathbb{F}^\times$ (where \mathbb{F}^\times denotes the multiplicative group of \mathbb{F}), and makes polynomial evaluation queries

in order to check the following equations:

$$\mathbf{f}'(\beta^2) = \mathbf{f}_e(\beta) + \rho_0 \cdot \mathbf{f}_o(\beta) = \frac{\mathbf{f}(\beta) + \mathbf{f}(-\beta)}{2} + \rho_0 \cdot \frac{\mathbf{f}(\beta) - \mathbf{f}(-\beta)}{2\beta}. \quad (4.1)$$

This is reminiscent of a reduction in [BBHR18] used to construct a low-degree test for univariate polynomials. By the Schwartz–Zippel lemma, the check passes with small probability unless $\mathbf{f}'(X)$ was computed correctly. Noting that $\hat{\mathbf{f}}(\boldsymbol{\rho}) = \langle \mathbf{f}, \otimes_{j=0}^{n-1}(1, \rho_j) \rangle$, this procedure gives a (univariate) polynomial IOP for this relation.

Definition 4. *The tensor-product relation \mathcal{R}_{TC} is the set of tuples*

$$(\mathbf{i}, \mathbf{x}, \mathbf{w}) = (\perp, (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, u), \mathbf{f})$$

where $n = \log N$, $\mathbf{f} \in \mathbb{F}^N$, $u \in \mathbb{F}$, and $\langle \mathbf{f}, \otimes_j(1, \rho_j) \rangle = u$.

We provide details of the tensor-product protocol in Section 4.5. In fact, the tensor check will be useful not only as part of our scalar-product protocol, but also more generally as part of simple checks that take place as part of our R1CS protocols (as described in Sections 4.2.5 and 4.2.6).

4.2.4.3 Elastic realization of the prover algorithm

Most complexity measures claimed in Theorem 4.2.5 follow straightforwardly from the sumcheck protocol described in Lemma 4.2.6. We are left to describe an elastic realization of the prover algorithm for the tensor-product protocol.

The prover’s task is to compute the polynomials $\mathbf{f}^{(j)}$ for each round $j \in [n]$. Given $\mathbf{f}^{(j-1)}$, which has degree $O(N/2^j)$, the prover can compute $\mathbf{f}^{(j)}$ in $O(N/2^j)$ operations via Equation (4.1). Summing up the prover costs for $j \in [n]$ gives $O(N)$ operations. Hence a linear-time prover realization for the tensor-product protocol is straightforward. Next, we give a space-efficient prover realization that uses logarithmic space.

Logarithmic space. We want the prover to run in logarithmic space, given streaming access to \mathbf{f} and \mathbf{g} . This is different from the time-efficient case, as the prover cannot store $\mathbf{f}^{(j-1)}$ to help it compute $\mathbf{f}^{(j)}$, as this requires linear space (for small j). Instead, the prover computes each $\mathbf{f}^{(j)}$ from scratch using streams of \mathbf{f} .

First we explain how the prover can *produce* a stream of $\mathbf{f}^{(j)}$ efficiently, given streaming access to \mathbf{f} , in a similar way to streaming evaluations of multivariate polynomials and low-degree extensions [CMT12; BHRRS20; BHRRS21]. Our contribution is to show that $\mathbf{f}^{(j)}$ can be evaluated in $O(N)$ time and $O(\log N)$ space, saving a logarithmic factor over prior work. Then, we explain how to perform the consistency checks.

- *Streaming $\mathbf{f}^{(j)}$.* Let $\mathbf{f} = \sum_{i=0}^{N-1} f_i X^i$. We can compute $\mathbf{f}' = \sum_{i=0}^{N/2-1} (f_{2i} + \rho f_{2i+1}) X^i$ from a stream of coefficients of \mathbf{f} by reading each pair of coefficients f_{2i}, f_{2i+1} from the stream, and computing the next coefficient as $f'_i := f_{2i} + \rho f_{2i+1}$ of \mathbf{f}' . This uses a constant amount of space: store f_{2i} and f_{2i+1} , and delete them right after computing f'_i . Each coefficient of \mathbf{f}' costs two arithmetic operations to compute.

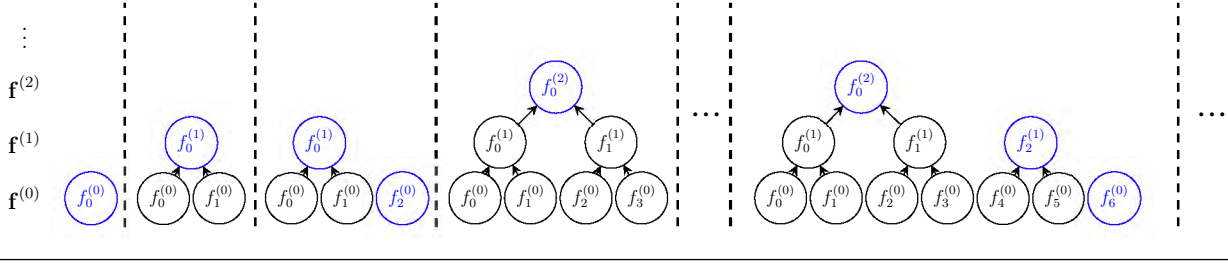


Figure 4.1: A streaming algorithm for computing the coefficients of $\mathbf{f}^{(j)}$ from $\mathbf{f}^{(0)} := \mathbf{f}$. Nodes in blue denote the coefficients that are stored in memory at any moment.

The prover can produce the stream $\mathcal{S}(\mathbf{f}^{(j)})$ for $\mathbf{f}^{(j)}$ by applying the same idea recursively as follows. Initialize a stack `Stack` consisting of pairs $(j, x) \in [\log N] \times \mathbb{F}$, and a list of challenges ρ_0, \dots, ρ_j . To generate $\mathcal{S}(\mathbf{f}^{(j)})$, the prover proceeds as follows.

- If the top element in the stack is of the form (j, y) for some $y \in \mathbb{F}$, pop it and return y .
- If the top two elements in the stack are of the form (k', x') and (k, x) with $k = k'$ (and $k < j$), then pop them and push $(k+1, x + \rho_k x')$, where $x + \rho_k x'$ is equal to $f_{k+1}^{(j)}$ (recall that the values are streamed from last to first index);
- Otherwise, query the stream $\mathcal{S}(\mathbf{f})$ for the next element $x \in \mathbb{F}$ and add $(0, x)$ to the stack.

The stack `Stack` must be initialized with some zero-entries if $N \neq 2^n$ (for instance, where N is odd) for correctness, but we avoid discussing this case here for simplicity. A visual representation of this process is displayed in Figure 4.1. This procedure produces a stream of $\mathbf{f}^{(j)}$ from a stream of \mathbf{f} in $O(N)$ and using $\log N$ memory space (since the stack `Stack` holds at most $\log N$ elements at any time).

- *Space-efficient tensor check.* The verifier must perform consistency checks to make sure that each polynomial $\mathbf{f}^{(j)}$ was correctly computed from $\mathbf{f}^{(j-1)}$, and similarly for $\mathbf{g}^{(j)}$. This check requires the computation of $\mathbf{f}^{(0)}, \dots, \mathbf{f}^{(n-1)}$. We compute them in parallel with a minor modification to the algorithm illustrated in Figure 4.1. Instead of returning only when the top of the stack has a particular index, we always output the top element in the stack. We thus construct a streaming algorithm $\mathcal{S}(\mathbf{f}^{(0)}, \dots, \mathbf{f}^{(n-1)})$ that returns elements of the form $(j, x) \in [n] \times \mathbb{F}$ where x is the next coefficient of the polynomial $\mathbf{f}^{(j)}$. With the above stream, it is possible to produce all streams $\mathcal{S}(\mathbf{f}^{(j)})$ and evaluations $\mathbf{f}^{(j)}(\beta^2), \mathbf{f}^{(j)}(+\beta), \mathbf{f}^{(j)}(-\beta)$, for each $j \in [n]$ with a single pass over $\mathcal{S}(\mathbf{f})$. In particular computing each evaluation requires storing a single \mathbb{F} -element; therefore, the total consistency check uses $n = \log N$ memory and N time. This allows to check Equation (4.1), substituting $\mathbf{f}' = \mathbf{f}^{(j)}, \mathbf{f} = \mathbf{f}^{(j-1)}$ for $j \in [n]$.

Based on the costs of maintaining the stacks for \mathbf{f} and \mathbf{g} , and computing the coefficients of $\mathbf{q}^{(j)}$ incrementally, it follows that each round takes time $O(N)$ and space $O(\log N)$. Therefore, summing over the $O(\log N)$ rounds, the protocol requires time $O(N \log N)$ and space $O(\log N)$.

Remark 4.2.8. Based on the tensor product protocol in Section 4.2.4.2, one can construct a linear-time univariate sumcheck protocol with proof length $O(N)$ and query complexity $O(\log N)$, which we believe could be of independent interest for future research. There are other univariate sumcheck protocols in the literature, however these protocols cannot be used in our setting.

- The univariate sumcheck protocol in [BCRSVW19] is a 1-message PIOP with proof length $O(N)$ and query complexity $O(1)$. That protocol does not seem useful here, because the prover requires $O(N \log N)$ time and $O(N)$ space due to the use of FFTs. In contrast, our protocol achieves elasticity, at the cost of logarithmic round complexity and logarithmic query complexity.
- Drake [Dra20] sketches a Hadamard product protocol based on univariate polynomials that does not use FFTs. That protocol, also inspired by the low-degree test in [BBHR18], may conceivably lead to a univariate sumcheck protocol that is elastic. No details (or implementations) of the protocol are available.

4.2.5 Warm-up: an elastic non-holographic PIOP for R1CS

We describe an elastic PIOP for R1CS (Definition 2) based on the elastic scalar-product protocol in Section 4.2.4. While not sublinear here, the verifier can be made elastic via similar techniques to the elastic prover. We build on this construction later in Section 4.2.6, and construct a *holographic* PIOP with logarithmic verifier time.

Theorem 4.2.9 (informal). *For every finite field \mathbb{F} , there is a PIOP for $\mathcal{R}_{\text{R1CS}}$ over \mathbb{F} with the following parameters:*

- *soundness error $O(N/|\mathbb{F}|)$;*
- *round complexity $O(\log N)$;*
- *proof length $O(N)$ and query complexity $O(\log N)$;*
- *a time-efficient prover that runs in time $O(M)$ and space $O(M)$;*
- *a space-efficient prover that runs in time $O(M \log^2 N)$ and space $O(\log N)$ (with $O(\log N)$ input passes);*
- *a time-efficient verifier that runs in time $O(M)$ and space $O(M)$; and*
- *a space-efficient verifier that runs in time $O(M \log N)$ and space $O(\log N)$ (with $O(\log N)$ input passes).*

Above, N is dimension of R1CS matrices and M the number of non-zero entries in the R1CS matrices.

The theorem holds for *any* finite field \mathbb{F} , and in particular does not require any smoothness properties for \mathbb{F} .

In order for the space-efficient realization of the prover to be well-defined, we must adopt a streaming model for R1CS instances. Below we describe a choice that: (i) suffices for the theorem; (ii) is realistic (as we elaborate shortly). After that we outline the PIOP for R1CS (and postpone details to Section 4.7).

Streaming R1CS. The R1CS problem is captured using the following indexed relation:

Definition 4.2.10. *The indexed relation $\mathcal{R}_{\text{R1CS}}$ is the set of all triples $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) = ((\mathbb{F}, N, M, A, B, C), \mathfrak{x}, \mathfrak{w})$ where \mathbb{F} is a finite field, A, B, C are matrices in $\mathbb{F}^{N \times N}$ each having at most M non-zero entries, and $\mathfrak{z} := (\mathfrak{x}, \mathfrak{w})$ is a vector in \mathbb{F}^N such that $A\mathfrak{z} \circ B\mathfrak{z} = C\mathfrak{z}$.*

We define streams for each of $\mathfrak{i}, \mathfrak{x}, \mathfrak{w}$, with A, B, C in sparse representation.

Definition 4.2.11. *The stream of U is a pair $(\mathcal{S}_{\text{rmaj}}(U), \mathcal{S}_{\text{cmaj}}(U))$, where $\mathcal{S}_{\text{rmaj}}(U)$ denotes the sequence of elements in the support (row, column, value) ordered in in row major (that is, lexicographic order with row), and $\mathcal{S}_{\text{cmaj}}(U)$ denotes the ordering of the ordering of the same sequence in column major.*

In our definition of streams for R1CS, we allow the *computation trace* $(A\mathfrak{z}, B\mathfrak{z}, C\mathfrak{z})$ of an R1CS instance to be streamed as part of the witness.

Definition 4.2.12 (streaming R1CS). *The streams associated with $((\mathbb{F}, N, M, A, B, C), \mathfrak{x}, \mathfrak{w})$ consist of:*

- **index streams:** *streams of the R1CS matrices, in row-major and column-major: $(\mathcal{S}_{\text{rmaj}}(A), \mathcal{S}_{\text{cmaj}}(A)), (\mathcal{S}_{\text{rmaj}}(B), \mathcal{S}_{\text{cmaj}}(B)), (\mathcal{S}_{\text{rmaj}}(C), \mathcal{S}_{\text{cmaj}}(C))$;*
- **instance stream:** *stream of the instance vector $\mathcal{S}(\mathfrak{x})$;*
- **witness streams:** *stream of the witness $\mathcal{S}(\mathfrak{w})$ and of the computation trace vectors $\mathcal{S}(A\mathfrak{z}), \mathcal{S}(B\mathfrak{z}), \mathcal{S}(C\mathfrak{z})$.*

The field description \mathbb{F} , instance size N , and maximum number M of non-zero entries are explicit inputs.

Including steams for the computation trace $(A\mathfrak{z}, B\mathfrak{z}, C\mathfrak{z})$ makes the PIOP for R1CS space efficient even when matrix multiplication by A, B, C requires a large amount of memory and the computation trace cannot be computed element by element on the fly given streaming access to \mathfrak{x} and \mathfrak{w} . On the other hand, for R1CS instances defined by many natural computations, such as a machine computation which repeatedly applies a transition function to a small state, the matrices A, B, C are such that their non-zero entries all lie in a thin, central diagonal band (that is, they are *banded*). In this case, one can generate a stream of $\mathcal{S}(A\mathfrak{z})$ using the streams $\mathcal{S}(\mathfrak{x}), \mathcal{S}(\mathfrak{w})$, and $\mathcal{S}_{\text{cmaj}}(A)$. (And similarly for B and C .)

The PIOP construction. We outline the PIOP construction underlying Theorem 4.2.9. The protocol adopts standard ideas from [BCRSVW19] and an optimization from [Gab20] for concrete efficiency. In the time-efficient realization, the prover receives $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ as input and the verifier receives $(\mathfrak{i}, \mathfrak{x})$ as input. In the space-efficient realization, these inputs are provided as streams according to Definition 4.4.9.

In the first step of the protocol, the prover sends \mathfrak{z} to the verifier. To check that $A\mathfrak{z} \circ B\mathfrak{z} = C\mathfrak{z}$, the verifier replies by sending a random challenge $v \in \mathbb{F}^\times$ to the prover, which the prover expands into a vector $\mathfrak{y}_C := (1, v, v^2, \dots, v^{N-1})$. Multiplying each side of the equation $A\mathfrak{z} \circ B\mathfrak{z} = C\mathfrak{z}$ on the left by \mathfrak{y}_C^\top , the prover is left to convince the verifier that

$$\langle A\mathfrak{z} \circ \mathfrak{y}_C, B\mathfrak{z} \rangle = \langle C\mathfrak{z}, \mathfrak{y}_C \rangle . \quad (4.2)$$

The prover sends the value $u_C := \langle Cz, y_C \rangle \in \mathbb{F}$ to the verifier. The prover will convince the verifier that Equation (4.2) holds by reducing the two claims $\langle Az \circ y_C, Bz \rangle = u_C$ and $\langle Cz, y_C \rangle = u_C$ to *tensor consistency checks* on z , for which we can apply the tensor-product protocol in Section 4.2.4.

As a subprotocol for the first claim, the prover and verifier run a twisted scalar product protocol, as described in Section 4.2.4. This generates two new claims, one about each of Az and Bz , leaving us with a total of three claims:

$$\begin{aligned} \langle Az, y_B \circ y_C \rangle &= u_A , \\ \langle Bz, y_B \rangle &= u_B , \\ \langle Cz, y_C \rangle &= u_C . \end{aligned} \tag{4.3}$$

Here, $y_B := \otimes_j(1, \rho_j)$, where $\rho_0, \rho_1, \dots, \rho_{n-1} \in \mathbb{F}^\times$ are the random challenges sent by the verifier during the scalar-product protocol. Setting $y_A := y_B \circ y_C$, and moving the matrices A, B, C into the right input argument of the scalar-product relation, we have

$$\begin{aligned} \langle z, \mathbf{a}^* \rangle &= u_A \quad \text{where } \mathbf{a}^* := y_A^\top A , \\ \langle z, \mathbf{b}^* \rangle &= u_B \quad \text{where } \mathbf{b}^* := y_B^\top B , \\ \langle z, \mathbf{c}^* \rangle &= u_C \quad \text{where } \mathbf{c}^* := y_C^\top C . \end{aligned} \tag{4.4}$$

Although y_B, y_C, y_A all have a tensor structure, $\mathbf{a}^*, \mathbf{b}^*, \mathbf{c}^*$ will not generally have the same structure, which means that Equation (4.4) cannot be checked directly using the tensor-product protocol. Thus, the verifier sends another random challenge $\eta \in \mathbb{F}^\times$ to the prover. Taking linear combinations of the three claims in Equation (4.4) using powers of η yields a single scalar-product claim

$$\langle z, \mathbf{a}^* + \eta \cdot \mathbf{b}^* + \eta^2 \cdot \mathbf{c}^* \rangle = u_A + \eta \cdot u_B + \eta^2 \cdot u_C . \tag{4.5}$$

The prover and verifier run a second twisted scalar-product protocol for Equation (4.5). This produces two new claims

$$\langle z, \mathbf{y} \rangle = u_D , \tag{4.6}$$

$$\langle \mathbf{a}^* + \eta \cdot \mathbf{b}^* + \eta^2 \cdot \mathbf{c}^*, \mathbf{y} \rangle = u_E , \tag{4.7}$$

where \mathbf{y} is a vector with the same tensor structure as described in Section 4.2.4, generated using random challenges produced by the verifier.

Finally, the prover and the verifier engage in a tensor-product protocol to check Equation (4.6). The verifier can check Equation (4.7) directly, since $\mathbf{a}^*, \mathbf{b}^*, \mathbf{c}^*$ can be computed directly from the RICS matrices A, B, C , along with the random challenges used throughout the RICS protocol.

Time-efficient prover. The prover runs in linear time if the prover algorithms for the underlying scalar-product and tensor-product subprotocols are realized in linear time. Note that the cost of computing $\mathbf{a}^*, \mathbf{b}^*, \mathbf{c}^*$ is linear in the number of non-zero entries in A, B, C . As a result, the verifier also runs in linear time.

Space-efficient prover. The scalar-product and tensor-product subprotocols used in the construction have a space-efficient prover that runs in time $O(N \log N)$ and space $O(\log N)$, given $O(\log N)$

passes over streams of the subprotocol inputs. Therefore, to give a space-efficient protocol for the entire R1CS protocol, it suffices to explain how to produce a stream for each subprotocol input.

The first twisted scalar-product protocol for $\langle Az \circ \mathbf{y}_C, Bz \rangle = u_C$ requires streaming access to Az, Bz, \mathbf{y}_C . The prover has streaming access to Az and Bz as part of the streams of the R1CS instance, so we explain how to generate a stream for the vector $\mathbf{y}_C = \otimes_j (1, v^{2^j}) \in \mathbb{F}^N$. This stream can be generated in $O(N)$ field operations. Let $v_j := v^{2^j}$ for $j \in [0, \dots, n-1]$. The i -th entry of \mathbf{y}_C is $\prod_j v_j^{b_j}$, where (b_0, \dots, b_{n-1}) is the binary representation of i . Consider how the binary representation of i changes when we subtract 1 from i . If $b_0 = 1$ then it simply changes to 0. If i ends with binary digits $(b_0, \dots, b_{k-1}, b_k) = (0, \dots, 0, 1)$ then these digits change to $(1, \dots, 1, 0)$. This means that we can get from the i -th entry of \mathbf{y}_C to the $(i-1)$ -th by multiplying by either v_0^{-1} or $v_k^{-1} v_{k-1} \cdots v_0$ for some $k \in [n]$. To generate the stream of \mathbf{y}_C , the prover computes $v_j := v^{2^j}$ for $j \in [0, \dots, n-1]$ via repeated squaring, which uses $O(\log N)$ operations and $O(\log N)$ space. Then, the prover can generate each element of \mathbf{y}_C in $O(N)$ operations by multiplying by the correct quotient.

The second scalar product protocol for Equation (4.5) requires streaming access to $\mathbf{z}, \mathbf{a}^* = \mathbf{y}_A^\top A, \mathbf{b}^* = \mathbf{y}_B^\top B$ and $\mathbf{c}^* = \mathbf{y}_C^\top C$. The prover has access to $\mathcal{S}(\mathbf{z})$ by concatenating the witness stream $\mathcal{S}(\mathbf{w})$ to the instance stream $\mathcal{S}(\mathbf{x})$. To generate the stream of $\mathbf{a}^* := \mathbf{y}_A^\top A$, the prover computes the i -th element of \mathbf{a}^* by multiplying each element of \mathbf{y}_A by each element of the i -th column of A , and adding the result to a running total. The stream $\mathcal{S}_{\text{cmaj}}(A)$ from the R1CS instance gives access to the non-zero entries of A , column by column. For \mathbf{y}_A , instead of generating the entire stream of \mathbf{y}_A for each i , which would cost $O(N^2)$ field operations in total, the prover generates elements of \mathbf{y}_A on the fly, at a cost $O(\log N)$ operations per element. Since A has M non-zero entries, the stream of \mathbf{a}^* costs $O(M \log N)$ operations to compute. The scalar product protocol requires $O(\log N)$ passes over the stream, and the prover runs in $O(M \log^2 N)$ time.

Combining this with the space-efficient realizations of the scalar-product and tensor-product subprotocols, which require $O(\log N)$ passes over their inputs, we obtain a space-efficient prover algorithm which runs in $O(M \log^2 N)$ time and $O(\log N)$ space.

4.2.6 Elastic holographic PIOP for R1CS

The verifier complexity in the non-holographic PIOP for R1CS described in Section 4.2.5 is linear in the size of the R1CS instance. To run a scalar-product protocol to check Equation (4.5), the verifier must compute $\mathbf{a}^*, \mathbf{b}^*, \mathbf{c}^*$ via expensive matrix-vector multiplications involving all of the non-zero entries of the matrices A, B, C .

Below we describe how to construct a *holographic* PIOP for R1CS, in which the verifier's direct access to A, B, C is replaced by query access. In this PIOP, the prover can either run in linear-time and linear-space or quasilinear-time and log-space, while the verifier runs in logarithmic time (and thus logarithmic space).

Theorem 5 (informal). *For every finite field \mathbb{F} , there exists an **holographic PIOP** for $\mathcal{R}_{\text{R1CS}}$ over \mathbb{F} with the following parameters:*

- *soundness error* $O(M/|\mathbb{F}|)$;
- *round complexity* $O(\log M)$;
- *proof length* $O(M)$ and *query complexity* $O(\log M)$;
- *an indexer that runs in time* $O(M)$ and *space* $O(M)$;
- *a time-efficient prover that runs in time* $O(M)$ and *space* $O(M)$;
- *a space-efficient prover that runs in time* $O(M \log^2 M)$ and *space* $O(\log M)$ with $O(\log M)$ input passes.
- *a verifier that runs in time* $O(|\mathbf{x}| + \log M)$ (and thus *space* $O(|\mathbf{x}| + \log M)$).

Here, M is the number of non-zero entries in the RICS matrices.

High-level overview. Our protocol follows the strategy in [BCG20]. The main difference between the holographic PIOP here and the non-holographic PIOP in Section 4.2.5 is that the prover and verifier use an alternative strategy to check Equation (4.4). The verifier does not compute \mathbf{a}^* (respectively, \mathbf{b}^* , \mathbf{c}^*) to check that $\langle \mathbf{z}, \mathbf{a}^* \rangle = u_A$ (same for B, C , cf. Equation (4.4)). Instead, the prover sends additional oracle messages to the verifier, which correspond to partial computations of the scalar product; the verifier checks these via multiple auxiliary subprotocols. The key subprotocols are a *look-up* protocol and an *entry-product* protocol.

Our main contribution is a space-efficient realization of these subprotocols, which leads to a space-efficient holographic RICS protocol. The main challenge is to show that it is possible to generate the prover's extra messages in a space-efficient manner from RICS streams (Definition 4.4.9). This places particular restrictions on the design of a space-efficient look-up protocol, which we explain how to deal with in Section 4.2.6.1. We explain how to construct a space-efficient entry-product protocol in Section 4.2.6.2.

Achieving holography. For a matrix $U \in \{A, B, C\}$, consider the vectors $\text{row}_U, \text{col}_U, \text{val}_U \in \mathbb{F}^M$ such that, for every $i \in [M]$, $\text{val}_{U,i} \in \mathbb{F}$ is the $(\text{row}_{U,i}, \text{col}_{U,i})$ -entry of U , ordered column-major. We assume that the matrices A, B, C have the same support, which means that $\text{row} := \text{row}_A = \text{row}_B = \text{row}_C$ and $\text{col} := \text{col}_A = \text{col}_B = \text{col}_C$. This can be achieved by suitably padding $\text{val}_A, \text{val}_B, \text{val}_C$ with zeroes, and increases the length of $\text{row}, \text{col}, \text{val}$ by at most a factor of 3.

The prover constructs the following vectors and sends them to the verifier as oracle messages:

$$\mathbf{r}_A^* := \mathbf{y}_A|_{\text{row}} , \quad \mathbf{r}_B^* := \mathbf{y}_B|_{\text{row}} , \quad \mathbf{r}_C^* := \mathbf{y}_C|_{\text{row}} , \quad \mathbf{z}^* := \mathbf{z}|_{\text{col}} , \quad (4.8)$$

where \mathbf{r}_A^* is the vector whose i -th element is the row_i -th element of \mathbf{a}^* , and similarly for $\mathbf{r}_B^*, \mathbf{r}_C^*, \mathbf{z}^*$. That is, $\mathbf{y}_A|_{\text{row}} := (\mathbf{y}_{A,i})_{i \in \text{row}}$. One proceeds similarly for $\mathbf{r}_B^*, \mathbf{r}_C^*$. Using Equation (4.8), Equation (4.3) can be reformulated as:

$$\begin{aligned} \langle \mathbf{r}_A^* \circ \text{val}_A, \mathbf{z}^* \rangle &= u_A , \\ \langle \mathbf{r}_B^* \circ \text{val}_B, \mathbf{z}^* \rangle &= u_B , \\ \langle \mathbf{r}_C^* \circ \text{val}_C, \mathbf{z}^* \rangle &= u_C . \end{aligned} \quad (4.9)$$

Then, the verifier must check the three claims of Equation (4.9), and that $\mathbf{r}_A^*, \mathbf{r}_B^*, \mathbf{r}_C^*, \mathbf{z}^*$ were correctly computed. The prover and verifier run a twisted scalar-product protocol for the three claims. To

check that \mathbf{r}_A^* , \mathbf{r}_B^* , \mathbf{r}_C^* and \mathbf{z}^* were correctly computed, the prover and verifier run a look-up protocol, which we describe in more detail in Section 4.2.6.1.

Elastic realization. The twisted scalar-product protocol and look-up protocol are elastic protocols with both time and space-efficient prover realization, and a succinct verifier. Our holographic PIOP for RICS inherits a time-efficient prover and succinct verifier from these subprotocols. However, to give a space-efficient prover realization, we must show that the prover can produce streams of \mathbf{r}_A^* , \mathbf{r}_B^* , \mathbf{r}_C^* , using input RICS streams and the verifier challenges. The RICS streams $\mathcal{S}_{\text{cmaj}}(A)$, $\mathcal{S}_{\text{cmaj}}(B)$ and $\mathcal{S}_{\text{cmaj}}(C)$ of the matrices A , B and C produce elements of the form $(i, j, e) \in [N] \times [N] \times \mathbb{F}$. Streaming only the first element of the triple produces the stream $\mathcal{S}_{\text{cmaj.row}}(A) = \mathcal{S}_{\text{cmaj.row}}(B) = \mathcal{S}_{\text{cmaj.row}}(C)$ of the vector row (we recall that we assumed the support of A, B, C to be the same, and that row is ordered column-major).

Similarly, the second element of the triple induces a stream $\mathcal{S}_{\text{cmaj.col}}(A)$ of the vector col, which is also equal to $\mathcal{S}_{\text{cmaj.col}}(B)$ and $\mathcal{S}_{\text{cmaj.col}}(C)$, again since the support is the same. Additionally, $\mathcal{S}_{\text{cmaj.col}}(A)$ is non-increasing: the column indices, in the dense representation of the matrix, are sorted in decreasing order when streamed column-major. As a result, the entries of \mathbf{z}^* can be produced one by one in $O(1)$ space from streams $\mathcal{S}(\mathbf{z})$ and $\mathcal{S}_{\text{cmaj}}(A)$: examine each entry of $\mathcal{S}_{\text{cmaj.col}}(A)$, advance forwards \mathbf{z} if the column changed, and output that same entry as long as the next element of $\mathcal{S}_{\text{cmaj.col}}(A)$ stays unchanged.

The streams $\mathcal{S}_{\text{cmaj.val}}(A)$ (respectively, $\mathcal{S}_{\text{cmaj.val}}(B)$ and $\mathcal{S}_{\text{cmaj.val}}(C)$) are defined by projecting onto the third element of the streams $\mathcal{S}_{\text{cmaj}}(A)$ (respectively, $\mathcal{S}_{\text{cmaj}}(B)$ and $\mathcal{S}_{\text{cmaj}}(C)$), and produce the streams for the vectors val_A , val_B , and val_C in column-major order.

For \mathbf{r}_A^* , \mathbf{r}_B^* and \mathbf{r}_C^* , recall that $\mathbf{y}_B = \otimes_j(1, \rho_j)$, $\mathbf{y}_C = \otimes_j(1, v^{2^j})$, and $\mathbf{y}_A = \mathbf{y}_B \circ \mathbf{y}_C = \otimes_j(1, \rho_j v^{2^j})$. Thus, any entry of \mathbf{r}_B^* or \mathbf{r}_C^* (and hence \mathbf{r}_A^*) can be computed in $O(\log N)$ operations from $v \in \mathbb{F}^\times$ and $\rho_0, \dots, \rho_{n-1} \in \mathbb{F}^\times$.

4.2.6.1 Lookup protocol

Lookup protocols enable the prover to convince the verifier that all of the entries in a vector $\mathbf{g}^* \in \mathbb{F}^M$ appear as entries of another vector $\mathbf{g} \in \mathbb{F}^N$ according to the data stored in the *address vector* $\text{addr} \in [N]^M$, i.e.:

$$\{(\mathbf{g}_i^*, \text{addr}_i)\}_{i \in [M]} \subseteq \{(\mathbf{g}_j, j)\}_{j \in [N]} .$$

We denote this condition by “ $(\mathbf{g}^*, \text{addr}) \subseteq (\mathbf{g}, [N])$ ”. In order to verify that \mathbf{r}_U^* and \mathbf{z}^* were correctly computed, the verifier must check four lookup relations:

$$\begin{aligned} (\mathbf{r}_A^*, \text{row}) &\subseteq (\mathbf{y}_A, [N]) , \\ (\mathbf{r}_B^*, \text{row}) &\subseteq (\mathbf{y}_B, [N]) , \\ (\mathbf{r}_C^*, \text{row}) &\subseteq (\mathbf{y}_C, [N]) , \\ (\mathbf{z}^*, \text{col}) &\subseteq (\mathbf{z}, [N]) . \end{aligned} \tag{4.10}$$

Note that $\mathbf{y}_A = \mathbf{y}_B \circ \mathbf{y}_C$, and \mathbf{r}_A^* , \mathbf{r}_B^* , \mathbf{r}_C^* come from looking up the entries of \mathbf{y}_A , \mathbf{y}_B and \mathbf{y}_C at the indices specified by row. Therefore, instead of checking that $(\mathbf{r}_A^*, \text{row}) \subseteq (\mathbf{y}_A, [N])$, it suffices

to check the Hadamard product relation $\mathbf{y}_A = \mathbf{y}_B \circ \mathbf{y}_C$. This can be done using an extension of the twisted scalar product protocol. This leaves four look-up relations to check.

Polynomial identities for look-up relations. To verify look-up relations, we use the polynomial identity from [GW20] and construct a PIOP to verify it via an approach similar to [BCG20].

We reduce the lookup conditions $(\mathbf{r}_U^*, \text{row}) \subseteq (\mathbf{y}_U, [N])$ and $(\mathbf{z}^*, \text{col}) \subseteq (\mathbf{z}, [N])$ to simpler inclusion conditions such as $\mathbf{f}^* \subseteq \mathbf{f}$, where each entry in the vector \mathbf{f}^* equals some entry in the vector \mathbf{f} . To do so, for each matrix $U \in \{A, B, C\}$, we *algebraically hash* the pairs $(\mathbf{z}^*, \text{col}), (\mathbf{z}, [N]), (\mathbf{r}_U^*, \text{row}), (\mathbf{y}_U, [N])$ into vectors $\mathbf{z}^* + \eta \cdot \text{col}$ (and similarly for the other pairs) in parallel, by taking a random linear combination of each pair using the same random challenge $\eta \in \mathbb{F}^\times$ from the verifier. Let $\text{sort}(\mathbf{g}, \mathbf{f})$ denote the function that sorts the entries of $\mathbf{g} \parallel \mathbf{f}$ according to order of appearance in \mathbf{f} .

Lemma 4.2.13 ([GW20, Claim 3.1]). *Let $\mathbf{f}^* \in \mathbb{F}^M$ and $\mathbf{f} \in \mathbb{F}^N$. Then $\mathbf{f}^* \subseteq \mathbf{f}$ if and only if there exists a witness $\mathbf{w} \in \mathbb{F}^{M+N}$ such that the equation below in $\mathbb{F}[Y, Z]$ is satisfied:*

$$\prod_{j=0}^{M+N-1} (Y(1+Z) + w_{j+1} + w_j \cdot Z) = (1+Z)^M \prod_{j=0}^{M-1} (Y + f_j) \prod_{j=0}^{N-1} (Y(1+Z) + f_{j+1} + f_j \cdot Z) \quad (4.11)$$

where indices are taken (respectively) modulo $M + N, N$. If $\mathbf{f}^* \subseteq \mathbf{f}$ then $\mathbf{w} := \text{sort}(\mathbf{f}^*, \mathbf{f})$ is a valid witness.

The strategy in the look-up protocol is for the prover to compute \mathbf{w} and prove that Equation (4.11) is satisfied, for every look-up relation that needs to be checked. The prover computes \mathbf{w} and sends it to the verifier. Then, the verifier sends random challenges $v, \zeta \in \mathbb{F}^\times$ to the prover, who computes each of the three product expressions in Equation (4.11), evaluated at v and ζ :

$$\begin{aligned} e_0 &= \prod_{i=0}^{M+N-1} (v(1 + \zeta) + w_{i+1 \bmod M+N} + w_i \cdot \zeta), \\ e_1 &= \prod_{i=0}^{M-1} (v + f_i^*), \\ e_2 &= \prod_{i=0}^{N-1} (v(1 + \zeta) + f_{i+1 \bmod N} + f_i \cdot \zeta). \end{aligned} \quad (4.12)$$

The prover then sends the three product values e_0, e_1, e_2 to the verifier. The verifier checks that Equation (4.11) holds at v and ζ by checking that $e_0 = (1 + \zeta)^M e_1 e_2$, and uses three *entry-product* subprotocols, which we describe in Section 4.2.6.2, to check that e_0, e_1, e_2 were correctly computed from \mathbf{f}^*, \mathbf{f} , and \mathbf{w} .

This approach requires polynomial query access to \mathbf{f}_\circ^* , the cyclic right-shift of \mathbf{f}^* , since the inputs to the entry product protocols depend on \mathbf{f}_\circ^* . The look-up protocol in [BCG20] uses a *shift* subprotocol to check this condition. By contrast, we avoid this additional step by considering instead the look-up protocol over vectors with a leading zero coefficient. Queries on the right-shift \mathbf{f}_\circ^* can be

related to queries on \mathbf{f}^* with a single evaluation query, since the leading coefficient is known in advance. We explain this optimization in Section 4.8.

Elastic realization. As shown in prior work [BCG20], if the underlying entry product protocols have a linear-time prover realization and succinct verifier, then the same is true for the look-up protocol. We focus on explaining a space-efficient prover realization of the look-up protocol. Assuming that the entry-product protocol has a suitable space-efficient realization, it suffices to explain how to realize streaming access to look-up protocol vectors \mathbf{f}^* , \mathbf{f} , \mathbf{w} using previously derived streams.

First we consider $(\mathbf{z}^*, \text{col})$ and $(\mathbf{z}, [N])$. Each pair is algebraically hashed into vectors \mathbf{f}^* and \mathbf{f} . One can produce the streams $\mathcal{S}(\mathbf{f}^*)$ and $\mathcal{S}(\mathbf{f})$ from the streams $\mathcal{S}(\mathbf{z}^*)$, $\mathcal{S}_{\text{cmaj.col}}(A)$, $\mathcal{S}(\mathbf{z})$, $\mathcal{S}([N])$, by applying the same algebraic hash function to pairs of entries on-the-fly. The same applies to input pairs $(\mathbf{r}_U^*, \text{row})$ and $(\mathbf{y}_U, [N])$.

Next we explain how to generate a stream of $\mathbf{w} = \text{sort}(\mathbf{f}^*, \mathbf{f})$ using small space. This is more challenging because storing the entire vectors \mathbf{f}^* and \mathbf{f} and sorting them requires space $O(M + N)$. In the case of inputs $(\mathbf{z}^*, \text{col})$ and $(\mathbf{z}, [N])$, as col is a non-decreasing sequence, it turns out that $\mathcal{S}_{\text{cmaj.col}}(A)$ is already sorted into a suitable order, and it suffices to *merge* the streams of \mathbf{f}^* and \mathbf{f} together to produce a stream for \mathbf{w} . The same is not true for row , which is not necessarily ordered. However, the vector row in non-decreasing form is already available from the inputs: it can be constructed from $\mathcal{S}_{\text{rmaj}}(A)$, the sparse representation of the matrix in row-major order. To apply the same idea to input pairs \mathbf{r}_U^* and row , we build $\mathcal{S}_{\text{rmaj.row}}(A)$, which is non-decreasing, and use it to produce the stream of the sorted vector for the lookup protocol. We describe our look-up protocol in more detail in Section 4.8.2.1.

On alternative proof techniques for look-up relations. Prior work such as [Set20] checks look-up relations using an *offline memory-checking* [BEGKN91; CDDGS03] abstraction in which the prover shows that \mathbf{g}^* was correctly constructed entry by entry from \mathbf{g} using read and write operations. This leads to an alternative polynomial identity replacing Equation (4.25), which uses a list of *timestamps* recording when a particular element of \mathbf{g}^* is read from \mathbf{g} . In this case though, it is unclear how to generate the timestamps required by this method without storing linear memory. While in [GW20] the polynomial relation is independent from the ordering of the matrix (row-major or column-major), the memory-checking approach requires random access to the vector row in order to access the last visited timestamps, which cannot be done in small space.

4.2.6.2 Entry product protocol

Let $\mathbf{f} = (f_0, \dots, f_{N-1}) \in \mathbb{F}^N$ such that $e = f_0 \cdots f_{N-1}$. We describe an entry-product protocol, building on [BCG20, Sec. 6.4], that reduces an entry product statement $\prod_i f_i = e$ to a single scalar-product relation, using polynomial evaluation query access to \mathbf{f} .

Compared with the prior work, our work exploits the structure of univariate polynomials to simplify the scheme and remove the need for *cyclic-shift tests* [BCG20, Sec. 6.3]. We propose additional optimizations in Section 4.2.7 which improve the concrete efficiency of our protocol.

High-level overview. Let \mathbf{f} be as above, with $f_{N-1} = 1$.⁶ Let $\psi \in \mathbb{F}^\times$ and let $\mathbf{y}' = (1, \psi, \dots, \psi^{N-1})$. Let \mathbf{g} be the vector of partial products of the entries of \mathbf{f} , that is:

$$\mathbf{g} := (\prod_{i \geq 0} f_i, \prod_{i \geq 1} f_i, \dots, f_{N-2}f_{N-1}, f_{N-1}) \quad (4.13)$$

Then, observe that:

$$\begin{aligned} \langle \mathbf{g} \circ \mathbf{y}', \mathbf{f}_\circ \rangle &= \sum_{i=1}^{N-1} g_i f_{i-1} \psi^i + g_0 f_{N-1} \\ &= \sum_{i=1}^{N-1} g_{i-1} \psi^i + e + g_{N-1} \psi^N - g_{N-1} \psi^N \\ &= \psi \mathbf{g}(\psi) + e - \psi^N . \end{aligned} \quad (4.14)$$

In the entry product protocol, the prover sends the oracle \mathbf{g} to the verifier, and the verifier replies with the random challenge $\psi \in \mathbb{F}^\times$, and makes a polynomial evaluation query $\mathbf{g}(\psi) = v$. Then, both parties engage in a twisted scalar product protocol to verify Equation (4.14). Polynomial evaluation queries $\mathbf{f}_\circ(x)$ for $x \in \mathbb{F}$ made as part of the twisted scalar-product protocol can be computed using evaluation queries $\mathbf{f}(x)$. To do this, note that $\mathbf{f}_\circ(x) = x\mathbf{f}(x) - x^N + 1$ since $f_{N-1} = 1$; thus the verifier can compute $\mathbf{f}_\circ(x)$ from $\mathbf{f}(x)$ in $O(\log N)$ operations. We give a formal description of the entry product protocol in Section 4.8.3.

Elastic realization. As with other subprotocols, the entry-product protocol inherits a linear-time prover realization and succinct verifier from the underlying twisted scalar-product protocol.

To give a space-efficient realization, it suffices to show that \mathbf{g} can be generated element-by-element given access to the stream $\mathcal{S}(\mathbf{f})$: the partial products of elements of \mathbf{f} can be produced by streaming each successive element of $\mathcal{S}(\mathbf{f})$ and multiplying it into a running product. Note that the partial products in \mathbf{g} are computed from the last entry to the first, starting with f_{N-1} . This is because streams of polynomials move from the highest-order coefficient to the lowest to be compatible with space-efficient commitment algorithms, as explained in Section 4.2.3.

4.2.7 Implementation and optimizations

We implemented the elastic SNARKs from Sections 4.2.5 and 4.2.6 by leveraging and extending arkworks [ark], a Rust ecosystem for developing and programming with zk-SNARKs. Our implementation is called ark-gemini, and is open-sourced as part of arkworks. The code structure follows the modular design of our construction, which involves combining an elastic polynomial commitment scheme and an elastic PIOP. We deem each of the components of our implementation (the streaming infrastructure, the commitment scheme, and the subprotocols for sumcheck, tensor check, entry product, and lookup) to be of independent interest for future work on space-efficient SNARKs. Below, we provide an overview of the streaming infrastructure and the algorithmic optimizations that we adopted in the implementation.

⁶ This restriction is merely didactical. Given any $\mathbf{f} \in \mathbb{F}^N$, representing the coefficients of a degree $N-1$ polynomial, it is easy to simulate polynomial-evaluation query access to $(\mathbf{f}, 1)$ using the polynomial $\mathbf{f}(X) + X^{N+1}$. For any evaluation query in $x \in \mathbb{F}$, forward evaluation queries to \mathbf{f} and add x^{N+1} before returning. This costs $O(\log N)$ \mathbb{F} -ops.

4.2.7.1 Streaming infrastructure

We extend the arkworks framework with support for streams in order to express space-efficient protocols. A stream is a wrapper over `iter::Iterator`, the Rust interface for iterators. Streams can be restarted and iterated over multiple times. We use Rust’s borrow abstractions to produce streams that avoid copying elements whenever possible: a stream either returns a field element, or a reference to a field element. In other words, we have a zero-copy interface where data structures do not require to be copied from memory, unless really needed. In practice, input streams can be instantiated with arrays (e.g., memory-mapped files) or a concurrent data stream downloaded from the web, but could be potentially extended to new inputs. Streams can be composed.

Baum, Malozemoff, Rosen, and Scholl [BMRS21] also study streaming provers, and provide a Rust implementation relying on asynchronous programming features of Rust. Rust’s asynchronous streams are also iterators, and thus our approach can be seen as a generalization of their framework.

4.2.7.2 Optimizations

We leverage several algorithmic optimizations that improve the concrete performance of our scheme. **Elastic provers.** Our elastic SNARK allows switching from the space-efficient implementation to the time-efficient implementation with specified memory threshold. For example, in the scalar product, if the prover has enough memory, then it can transcribe the intermediate prover state and proceed with the time-efficient implementation of the prover function. This allows for a more fine-grained control of the space complexity of the prover, and to benefit from the speed-up of the time-efficient prover for the last few rounds of the protocol. Since the prover’s messages are the same in both modes, this switch does not affect the end result.

Batch tensor-product protocols. As discussed in Section 4.2.4.2, we use a tensor product protocol to check the multivariate evaluation claims resulting from the sumcheck protocol. In our holographic PIOP, we batch multiple tensor product claims in parallel using the same randomness. Moreover, the polynomials in each round can be batched into a single polynomial commitment per round of the tensor product protocol.

Batch [KZG10] for multiple points and polynomials. Boneh et al. [BDFG20] proposed an optimization of [KZG10] to batch evaluation proofs for a set of evaluation points over different polynomials, exploiting the structure of univariate polynomials. We adapt and implement these optimizations to our elastic polynomial commitment scheme. In particular, while our tensor product protocol requires the verifier to query different polynomials at different evaluation points, evaluation proofs are batched into a single group element. This makes the concrete size of the evaluation proof smaller than for multi-linear approaches such as [ZGKPP17b; ZGKPP18], which require a logarithmic-size evaluation proof. We elaborate on this in Section 4.9.

Offline memory-checking. As discussed in Section 4.2.6.1, the offline memory-checking protocol is not compatible with space efficiency, because the computation of timestamps (in general) requires random-access over the sparse representation of the RICS matrices A, B, C . Nevertheless, we observe that in the particular implementation of our protocol, the offline memory checking can be used to prove the lookup for $(\mathbf{z}_U^*, \text{col}_U) \subseteq (\mathbf{z}, [N])$. We view the offline memory-checking as

an optimization because it is concretely more efficient than the plookup protocol. That is because the sender in the plookup protocol must send additional commitments to the verifier; whereas, the commitments in the offline memory-checking can be precomputed by the indexer. We elaborate on this in Section 4.8.2.2.

4.2.8 Evaluation

We run extensive benchmarks for Gemini (both preprocessing and non-preprocessing SNARKs), over an Amazon AWS EC2 c5.9xlarge instance, with 36 cores. We use the Rust library `rayon` for multi-threading, and use parallelism for multi-scalar multiplications and for the batched sumcheck in the preprocessing SNARK (multiple sumcheck instances are run in parallel). We select BLS12-381 as the underlying pairing-friendly elliptic curve, but note that we do not rely on the smoothness of this curve’s prime field (see Remark 4.2.7).

We benchmark instance sizes N from 2^{18} to 2^{35} (with $M = N$). These sizes are much larger than what is commonly reported in the literature, and showcase the behavior of our SNARK over very large instances.

Proving space. Gemini supports proving instances of arbitrary sizes. Figure 4.2 shows that memory usage remains constant as instance size increases: it is below 1GB memory for the non-preprocessing SNARK, and slightly more than 1GB for the preprocessing SNARK. Two main parameters affect memory usage.

- The memory budget allocated for multi-scalar multiplication (MSM). Algorithms for MSM (e.g., Pippenger’s algorithm) achieve improved time efficiency at the expense of large space usage (linear in the number of scalar multiplications), which precludes an elastic implementation. In practice, to maintain space efficiency, it is useful to allocate a constant-size buffer, and apply the MSM algorithm over chunks of the inputs, accumulating the final result. We set the buffer to be of size 2^{20} .
- The sumcheck round threshold, after which the elastic prover transcribes the sumcheck intermediate state and proceeds with the time-efficient algorithm. We set the threshold to 22: the last 22 rounds of the sumcheck protocol are performed with the time-efficient prover.

The memory usage is obtained reading the value of resident data and stack memory at regular intervals of 10 seconds on `/proc/[pid]/statm`.

The overall memory consumption appears constant, suggesting that the above parameters dominate the logarithmic factor in space complexity. The difference in memory consumption between the two SNARKs is explained by the batch sumcheck (used solely in the preprocessing SNARK), where multiple instances are transcribed in memory at the same time.

Our benchmarks stop at 2^{35} for the non-preprocessing SNARK and the 2^{32} for the preprocessing one, but the upper limit in our benchmarks is arbitrary: as long as it is possible to generate the input streams for the time prover, then prover can terminate while keeping memory usage small. Prior work on preprocessing SNARKs for R1CS provide benchmarks for sizes up to 2^{20} . When running

benchmarks ourselves to compare our work with previous literature such as Marlin⁷, we were unable to proceed beyond size 2^{24} due to out of memory crashes. This is due to the kernel’s OOM (Out Of Memory) Killer process that intervenes forbidding new allocations and terminating the prover before the end of its execution.⁸

Proving time. The elastic prover switches to the time-efficient mode if the intermediate state fits within the memory budget. In particular, when the instance size is small enough, the elastic prover runs in the time-efficient mode only. The most time expensive operations in the protocol are the cryptographic operations, namely the multi-scalar multiplications. For this reason, in Figure 4.2, where we show the running times for for different values of N , with $M = N$, it is possible to observe a graph that evolves almost linearly. The squared logarithmic factor does not influence noticeably the overall runtime (as far as we were able to measure within the window of instance sizes of our benchmarks).

Proving cost in dollars. The preprocessing SNARK prover spends about 7.6×10^{-5} seconds per gate. Using the AWS estimator (on-demand hourly cost 1.836 USD obtained from <https://calculator.aws>), we conclude that the cost for the preprocessing SNARK is about 2.30×10^{-5} USD per gate. In particular, the estimated cost for instance size 2^{31} is 89 USD. In contrast, for this instance size, DIZK [WZCPS18] incurs a much higher cost at around 500 USD; this is because DIZK runs the prover on 20 more powerful and expensive machines (r3.8xlarge EC2 instances with on-demand hourly cost 2.656 USD) for about 10 hours. In the case of non-preprocessing SNARK, the cost is lower and around 40 USD for the size 2^{35} .

Proof size and verification time. We measure the proof size and verification time for the preprocessing SNARK. The verifier can cheaply verify proofs for large instances since it does not read the instance (instead, it uses a short verification key derived from the instance). For instance sizes ranging from 2^{12} to 2^{35} , the proof size is about 13 – 27 KB and the verification time is about 16 – 30 ms.

⁷cf. <https://github.com/arkworks-rs/marlin>.

⁸We observe that the program will still crash if we instruct the kernel to allow for memory over commitment. See `vm.overcommit=2` at <https://www.kernel.org/doc/Documentation/vm/overcommit-accounting>.

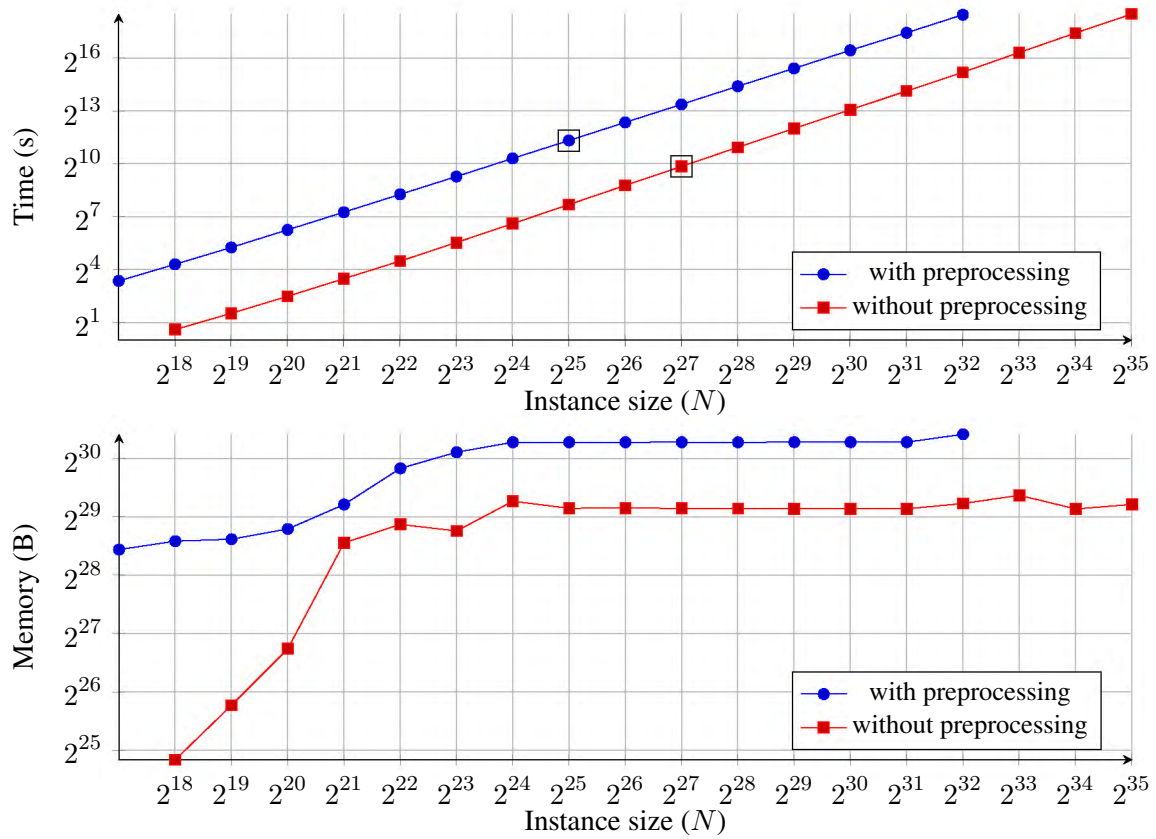


Figure 4.2: Running time (above) and memory usage (below) for the elastic prover in the *preprocessing* protocol (blue) and the *non-preprocessing* protocol (red), for different RICS sizes with $N = M$. The black squares indicate the size for which the time-efficient prover triggers an out-of-memory crash (it uses too much memory).

4.3 Preliminaries

4.3.1 Notation

For integers a, b with $a < b$ integers, let $[a, b]$ denote the set $\{a, \dots, b - 1\}$; $[b]$ will be used as a shorthand for $[0, b]$. Let $\log: \mathbb{N} \rightarrow \mathbb{Z}$ denote the base-2 logarithm function, rounded up to the nearest integer. Let \mathbb{F} be a field; and let \mathbb{F}^\times denote its invertible elements. For a vector \mathbf{a} over \mathbb{F} of length N , let a_i denote its i -th entry. Let $\mathbf{a} \circ \mathbf{b}$ denote the Hadamard product of vectors \mathbf{a} and \mathbf{b} ; $\mathbf{a} \cdot \mathbf{b}$ their dot product and $\mathbf{a} \otimes \mathbf{b}$ their tensor product. We view vectors both as an element of \mathbb{F}^N as well as a polynomial in $\mathbb{F}[X]$ of degree at most N . Given a vector $\mathbf{a} \in \mathbb{F}^N$ and an indexing vector $\text{idx} \in [N]^M$, let $\mathbf{a}|_{\text{idx}} \in \mathbb{F}^M$ be the vector whose i -th element is a_{idx_i} . We will occasionally explicitly refer to the polynomial associated with a vector $\mathbf{a} \in \mathbb{F}^N$ using the notation $\mathbf{a}(X) \in \mathbb{F}[X]$.

Algorithms are written in calligraphic math font. We use standard big O notation for asymptotic operations over field elements. We use $\text{negl}(\lambda)$ as a shorthand for $\text{negl}(\lambda) = \lambda^{-\omega(1)}$. We use O_λ to describe big O notation in which λ is considered a constant. This will be useful when describing the cost of cryptographic operations, which depend on a security parameter λ .

We use $\mathcal{S}(a)$ to denote the stream of a . This is defined in Page 110 for vectors and matrices, and in Definition 4.4.9 for RICS instances.

Definition 4.3.1. Let $\mathbf{v} \in \mathbb{F}^N$. We denote with $\mathbf{v}_\circ \in \mathbb{F}^N$ the right rotation, i.e. $\mathbf{v}_\circ := (v_N, v_1, \dots, v_{N-1})$.

Definition 4.3.2 (multilinear polynomials). Let $\mathbf{f} \in \mathbb{F}^N$, with $N = 2^n$. Write $\hat{\mathbf{f}}(X_0, \dots, X_{n-1})$ for the multilinear polynomial whose coefficients are the entries of \mathbf{f} , ordered so that $\mathbf{f}_{i_0 \dots i_{n-1}}$ is the coefficient of $X_0^{i_0} \dots X_{n-1}^{i_{n-1}}$.

Definition 4.3.3 (odd and even parts). Let $\mathbf{f}(X) \in \mathbb{F}[X]$. Let $\mathbf{f}_e(X), \mathbf{f}_o(X) \in \mathbb{F}[X]$ be the even and odd parts of $\mathbf{f}(X)$ i.e. the unique polynomials such that $\mathbf{f}(X) = \mathbf{f}_e(X^2) + X\mathbf{f}_o(X^2)$.

4.3.2 Polynomial IOPs

A polynomial IOP [CHMMVW20; BFS20] over a field family \mathcal{F} for an indexed relation \mathcal{R} is specified by a tuple

$$\text{IOP} = (\mathbf{k}, o, \mathbf{d}, \mathcal{I}, \mathcal{P}, \mathcal{V})$$

where $\mathbf{k}, o, \mathbf{d}: \{0, 1\}^* \rightarrow \mathbb{N}$ are polynomial-time computable functions and $\mathcal{I}, \mathcal{P}, \mathcal{V}$ are three algorithms known as the *indexer*, *prover*, and *verifier*. The parameter \mathbf{k} specifies the number of interaction rounds, o specifies the number of polynomials in each round, and \mathbf{d} specifies degree bounds on these polynomials.

In the offline phase (“0-th round”), the indexer \mathcal{I} receives as input a field $\mathbb{F} \in \mathcal{F}$ and an index \mathbf{i} for \mathcal{R} , and outputs $o(0)$ polynomials $\mathbf{p}_0^{(0)}, \dots, \mathbf{p}_{o(0)-1}^{(0)} \in \mathbb{F}[X]$ of degrees at most $\mathbf{d}(|\mathbf{i}|, 0, 1), \dots, \mathbf{d}(|\mathbf{i}|, 0, o(0))$ respectively. Note that the offline phase does not depend on any particular instance or witness, and merely considers the task of encoding the given index \mathbf{i} .

In the online phase, given an instance \mathbb{x} and witness \mathbb{w} such that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$, the prover \mathcal{P} receives $(\mathbb{F}, \mathbb{i}, \mathbb{x}, \mathbb{w})$ and the verifier \mathcal{V} receives (\mathbb{F}, \mathbb{x}) and oracle access to the polynomials output by $\mathcal{I}(\mathbb{F}, \mathbb{i})$. The prover \mathcal{P} and the verifier \mathcal{V} interact over $k = k(|\mathbb{i}|)$ rounds.

For $j \in [k]$, in the j -th round of interaction, the verifier \mathcal{V} sends a message $\rho_j \in \mathbb{F}^\times$ to the prover \mathcal{P} ; then the prover \mathcal{P} replies with $o(j)$ oracle polynomials $\mathbf{p}_0^{(j)}, \dots, \mathbf{p}_{o(j)-1}^{(j)} \in \mathbb{F}[X]$. The verifier may query any of the polynomials it has received any number of times. A query consists of a location $z \in \mathbb{F}$ for an oracle $\mathbf{p}_i^{(j)}$, and its corresponding answer is $\mathbf{p}_i^{(j)}(z) \in \mathbb{F}$. After the interaction, the verifier accepts or rejects. The function d determines which provers to consider for the completeness and soundness properties of the proof system. In more detail, we say that a (possibly malicious) prover $\tilde{\mathcal{P}}$ is **admissible** for IOP if, on every interaction with the verifier \mathcal{V} , it holds that for every round $j \in [k]$ and oracle index $i \in [o(j)]$ we have $\deg(\mathbf{p}_i^{(j)}) \leq d(|\mathbb{i}|, j, i)$. The honest prover \mathcal{P} is required to be admissible under this definition.

Remark 4.3.4 (non-oracle messages). We also allow the prover in an IOP to arbitrary messages that the verifier will simply read in full (without making any queries), at any point in the interaction, as in a typical interactive proof. We refer to such messages as *non-oracle messages*, to differentiate them from the *oracle messages* to which the verifier has query access. These non-oracle messages can typically be viewed as degenerate cases of oracle messages, and we use them in protocol descriptions for ease of exposition.

We say that IOP has perfect completeness and soundness error ϵ if the following holds.

- **Completeness.** For every field $\mathbb{F} \in \mathcal{F}$ and index-instance-witness tuple $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$, the probability that $\mathcal{P}(\mathbb{F}, \mathbb{i}, \mathbb{x}, \mathbb{w})$ convinces $\mathcal{V}^{\mathcal{I}(\mathbb{F}, \mathbb{i})}(\mathbb{F}, \mathbb{x})$ to accept in the interactive oracle protocol is 1.
- **Soundness.** For every field $\mathbb{F} \in \mathcal{F}$, index-instance pair $(\mathbb{i}, \mathbb{x}) \notin \mathcal{L}(\mathcal{R})$, and admissible prover $\tilde{\mathcal{P}}$, the probability that $\tilde{\mathcal{P}}$ convinces $\mathcal{V}^{\mathcal{I}(\mathbb{F}, \mathbb{i})}(\mathbb{F}, \mathbb{x})$ to accept in the interactive oracle protocol is at most ϵ .

The *proof length* l is the sum of all degree bounds in the offline and online phases, $l(|\mathbb{i}|) := \sum_{j=0}^{k(|\mathbb{i}|)-1} \sum_{i=0}^{o(j)-1} d(|\mathbb{i}|, j, i)$.

The *query complexity* q is the total number of queries made by the verifier to the polynomials. This includes queries to the polynomials output by the indexer and those sent by the prover.

All PIOPs that we construct achieve the stronger property of *knowledge soundness* (against admissible provers). We define both of these properties below.

Knowledge soundness. We say that IOP has knowledge error ϵ if there exists a probabilistic polynomial-time extractor \mathbf{E} for which the following holds. For every field $\mathbb{F} \in \mathcal{F}$, index \mathbb{i} , instance \mathbb{x} , and admissible prover $\tilde{\mathcal{P}}$, the probability that $\mathbf{E}^{\tilde{\mathcal{P}}}(\mathbb{F}, \mathbb{i}, \mathbb{x}, 1^{l(|\mathbb{i}|)})$ outputs \mathbb{w} such that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$ is at least the probability that $\tilde{\mathcal{P}}$ convinces $\mathcal{V}^{\mathcal{I}(\mathbb{F}, \mathbb{i})}(\mathbb{F}, \mathbb{x})$ to accept minus ϵ . Here the notation $\mathbf{E}^{\tilde{\mathcal{P}}}$ means that the extractor \mathbf{E} has black-box access to each of the next-message functions that define the interactive algorithm $\tilde{\mathcal{P}}$. (In particular, the extractor \mathbf{E} can “rewind” the prover $\tilde{\mathcal{P}}$.) Note that

since \mathbf{E} receives the proof length $l(|\mathfrak{i}|)$ in unary, \mathbf{E} has enough time to receive, and perform efficient computations on, polynomials output by $\tilde{\mathcal{P}}$.

Additional properties. All of our PIOP protocols will satisfy the following additional properties:

- *Public coins:* IOP is *public-coin* if each verifier message to the prover is a uniformly random string of some prescribed length (or an empty string). Hence the verifier’s randomness is its messages $\rho_0, \dots, \rho_{k-1} \in \mathbb{F}^\times$ and possibly additional randomness $\rho_k \in \mathbb{F}^\times$ used after the interaction. All verifier queries can be postponed, without loss of generality, to a query phase that occurs after the interactive phase with the prover.
- *Non-adaptive queries:* IOP is *non-adaptive* if all of the verifier’s query locations are solely determined by the verifier’s randomness and inputs (the field \mathbb{F} and the instance \mathfrak{x}).

Polynomial IOPs of proximity.

An *polynomial IOP of proximity* is similar to a PIOP, with the difference that the verifier \mathcal{V} has query access to the candidate witness \mathfrak{w} (we assume that \mathfrak{w} can be parsed as a polynomial or polynomials) as well as $(\mathcal{I}(\mathbb{F}, \mathfrak{i}), \mathbf{p}_0^{(0)}, \dots, \mathbf{p}_{o(k)-1}^{(k-1)})$. Completeness and soundness properties of PIOPPs are defined similarly to those of PIOPs, except that \mathcal{V} has query access to the polynomials in the witness \mathfrak{w} .

4.4 Streaming model

We provide a formal model of streams and streaming algorithms.

Definition 4.4.1. A stream is a sequence $K \in \Sigma^I$, where Σ is a finite alphabet, and I is a well-ordered set.

Definition 4.4.2 (streaming oracle). Let K be a stream over the alphabet Σ and index I . The streaming oracle $\mathcal{S}(K)$ of K behaves as follows:

- On inputs *start* and a session number $L \in \mathbb{N}$, the oracle creates or resets a counter $i_L \in I \cup \{\perp\}$ which is initially set to the first element of I .
- On inputs *next* and a session number $L \in \mathbb{N}$, if $i_L \in I$, then the oracle returns $k_{i_L} \in U$, and updates i_L to the next element of I (or to \perp if i_L is equal to the last element of I). If $i_L = \perp$ then the oracle returns \perp .

Remark 4.4.3. The use of session numbers L allows the same stream to be accessed in different positions by multiple algorithms simultaneously. However, to avoid an unrealistic streaming model in which algorithms have arbitrary random access to streamed data, in this work, no stream will be accessed through more than a logarithmic number of sessions simultaneously.

If v is an array whose elements have a clear ordering in context, then we simply say that A has streaming oracle access to v . We now introduce the streaming algorithm, which has access to streaming oracles in a specific order and produce an output stream.

4.4.1 Streaming algorithms

Definition 4.4.4 (streaming algorithm). We say that A is a **streaming algorithm** over Σ if A receives no inputs, but has access to various streaming oracles $\mathcal{S}(K_1), \dots, \mathcal{S}(K_l)$ over some alphabet Σ through *start* and *next* commands. In addition, A produces output upon receiving the command *next*, which takes one element of Σ as input. We write $O = A(\mathcal{S}(K_1), \dots, \mathcal{S}(K_l))$ to show that O is the entire output stream of A .

It is possible to compose streaming algorithms so that one stream algorithm has streaming access to the output of another stream algorithm.

Definition 4.4.5 (composing streaming algorithms). Let A and B be streaming algorithms. Suppose that B takes inputs $\mathcal{S}(K_1), \dots, \mathcal{S}(K_l)$. We write $A(B)$ when A interacts with B as follows:

- when A sends a *start* command to B , the execution of B is reset, and A also sends *start* commands to each of $\mathcal{S}(K_1), \dots, \mathcal{S}(K_l)$;
- A forwards any *start* or *next* command from B to the correct streaming oracle, and returns the output to B ;

- on input next, the execution of B yields the next outputs and returns it to A .

Lemma 4.4.6. *If A is a streaming algorithm with time complexity t_A , space complexity s_A , and k_A input passes, and B is a streaming algorithm with time complexity t_B , space complexity s_B , and k_B input passes, then A composed with B has time complexity $t_A + k_A t_B$, space complexity $s_A + s_B$, and $k_A k_B$ input passes.*

4.4.2 Streaming R1CS

We introduce the streaming R1CS model. We begin by recalling the indexed R1CS relation.

Definition 4.4.7. *The indexed relation $\mathcal{R}_{\text{R1CS}}$ is the set of all triples $(\mathbb{F}, \mathbf{x}, \mathbf{w}) = ((\mathbb{F}, N, M, A, B, C), \mathbf{x}, \mathbf{w})$ where \mathbb{F} is a finite field, A, B, C are matrices in $\mathbb{F}^{N \times N}$ each having at most M non-zero entries, and $\mathbf{z} := (\mathbf{x}, \mathbf{w})$ is a vector in \mathbb{F}^N such that $A\mathbf{z} \circ B\mathbf{z} = C\mathbf{z}$.*

We define a streaming R1CS instance in the terms of the sparse representation of the R1CS matrices A, B and C .

Definition 4.4.8. *The stream of U is a pair $(\mathcal{S}_{\text{rmaj}}(U), \mathcal{S}_{\text{cmaj}}(U))$, where $\mathcal{S}_{\text{rmaj}}(U)$ denotes the sequence of elements in the support (row, column, value) ordered in row major (that is, lexicographic order with row), and $\mathcal{S}_{\text{cmaj}}(U)$ denotes the ordering of the same sequence in column major.*

Definition 4.4.9 (streaming R1CS). *The streams associated with $((\mathbb{F}, N, M, A, B, C), \mathbf{x}, \mathbf{w})$ consist of:*

- **index streams:** streams of the R1CS matrices, in row-major and column-major: $(\mathcal{S}_{\text{rmaj}}(A), \mathcal{S}_{\text{cmaj}}(A)), (\mathcal{S}_{\text{rmaj}}(B), \mathcal{S}_{\text{cmaj}}(B)), (\mathcal{S}_{\text{rmaj}}(C), \mathcal{S}_{\text{cmaj}}(C))$;
- **instance stream:** stream of the instance vector $\mathcal{S}(\mathbf{x})$;
- **witness streams:** stream of the witness $\mathcal{S}(\mathbf{w})$ and of the computation trace vectors $\mathcal{S}(A\mathbf{z}), \mathcal{S}(B\mathbf{z}), \mathcal{S}(C\mathbf{z})$.

The field description \mathbb{F} , instance size N , and maximum number M of non-zero entries are explicit inputs.

The streaming R1CS relation naturally captures other models of computation, such as *R1CS automata*.

Definition 4.4.10 (R1CS automata). *Let \mathbb{F} be a finite field, $T \in \mathbb{N}$ be a computation time, and $k \in \mathbb{N}$ a computation width. We consider execution traces $f \in (\mathbb{F}^k)^T$. Each state $f[t] \in \mathbb{F}^k$ represents the state of the computation at time $t \in [T]$.*

An R1CS automaton is specified by matrices $A, B, C \in \mathbb{F}^{k \times 2k}$ that define constraints between different time steps, and a set of boundary constraints $\mathcal{B} \subseteq [T] \times [k] \times \mathbb{F}$.

An execution trace f is accepted by the automaton if:

- f satisfies the RICS time constraints i.e. $\forall t \in [T]$, it holds that $Af[t, t + 1] \circ Bf[t, t + 1] = Af[t, t + 1]$;
- f satisfies the RICS boundary conditions i.e. $\forall (t, j, \alpha) \in \mathcal{B}$ it holds that $f[t]_j = \alpha$.

Theorem 4.4.11 (automata to streaming RICS). *Let $(\mathbb{F}, k, T, A, B, C, \mathcal{B})$ be an RICS automata instance. Then there is an RICS instance which verifies the same computation as $(\mathbb{F}, k, T, A, B, C, \mathcal{B})$, and whose streams, for the streaming RICS model, can be produced in time $O(k^2 \log k)$ and space $O(k^2)$.*

Further, the witness streams can be produced from f in $O(k^2)$ operations per element of \mathbb{F}^k , and using space $O(k^2)$.

4.5 Tensor-product protocol

We introduce a tensor-product checking protocol which is used in our constructions. Then we show how to batch multiple tensor-product checks, which leads to better performance than running the basic protocol multiple times.

4.5.1 Basic tensor-product protocol

The tensor-product protocol checks the following relation.

Definition 4.5.1. *The tensor-product relation \mathcal{R}_{TC} is the set of tuples*

$$(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) = (\perp, (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, u), \mathbf{f})$$

where $n = \log N$, $\mathbf{f} \in \mathbb{F}^N$, $u \in \mathbb{F}$, and $\langle \mathbf{f}, \otimes_j(1, \rho_j) \rangle = u$.

Theorem 4.5.2. *For every finite field \mathbb{F} and positive integer N , there is a PIOP for the indexed relation \mathcal{R}_{TC} that supports instances over \mathbb{F} , with:*

time-efficient prover	space-efficient prover	verifier time	soundness error	round complexity	message complexity	query complexity
$O(N)$ \mathbb{F} -ops	$O(N \log N)$ \mathbb{F} -ops	$O(\log N)$ \mathbb{F} -ops	$O\left(\frac{N}{ \mathbb{F} }\right)$	$O(\log N)$	$O(\log N)$	$O(\log N)$
$O(N)$ memory	$O(\log N)$ memory $O(1)$ passes					

We prove Theorem 4.5.2 with the following construction.

Construction 1. *We construct a PIOP for the indexed relation \mathcal{R}_{TC} . The prover \mathcal{P} takes as input an index $\mathfrak{i} = \perp$, instance $\mathfrak{x} = (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, u)$, and witness $\mathfrak{w} = \mathbf{f}$; the verifier \mathcal{V} takes as input the index \mathfrak{i} and the instance \mathfrak{x} .*

- Write $\mathbf{f}^{(0)}(X) = \mathbf{f}(X)$.
- For $j \in [n]$, the prover \mathcal{P} computes

$$\mathbf{f}^{(j)}(X) := \mathbf{f}_e^{(j-1)}(X) + \rho_{j-1} \cdot \mathbf{f}_e^{(j-1)}(X) .$$

where $\mathbf{f}^{(j)}(X) = \mathbf{f}_e^{(j-1)}(X^2) + X \mathbf{f}^{(j)}(X^2)_o$. The prover sends the oracle messages $\mathbf{f}^{(1)}, \dots, \mathbf{f}^{(n-1)}$ to the verifier.

- The verifier \mathcal{V} samples a challenge $\beta \leftarrow \mathbb{F}^\times$ uniformly at random and makes the following evaluation queries for $j \in [n]$:

$$e^{(j)} := \mathbf{f}^{(j)}(\beta), \quad \bar{e}^{(j)} := \mathbf{f}^{(j)}(-\beta), \quad \hat{e}^{(j)} := \mathbf{f}^{(j+1)}(\beta^2), \quad (4.15)$$

Skip $\hat{e}^{(n)}$ and artificially set $\hat{e}^{(n)} := u$.

- check that, for all $j \in [n]$:

$$\hat{e}^{(j)} = \frac{e^{(j)} + \bar{e}^{(j)}}{2} + \rho_j \cdot \frac{e^{(j)} - \bar{e}^{(j)}}{2\beta}, \quad (4.16)$$

Lemma 4.5.3. *Construction 1 has perfect completeness.*

Proof. Suppose that $\langle \mathbf{f}, \otimes_j(1, \rho_j) \rangle = u$. We argue that the verifier's consistency checks of Equation (4.16) are satisfied. Firstly, indexing $\mathbf{f} \in \mathbb{F}^N$ by $i_0, \dots, i_{n-1} \in \{0, 1\}$, one can prove by induction on j that:

$$\mathbf{f}^{(j)}(X) = \sum_{i_0, \dots, i_{n-1} \in \{0, 1\}} \mathbf{f}^{(i_0, \dots, i_{n-1})_2} \rho_0^{i_0} \dots \rho_{j-1}^{i_{j-1}} X^{i_j + 2i_{j+1} + \dots + 2^{n-1-j} i_{n-1}}$$

for $j \in [n]$. This shows that $\mathbf{f}^{(n)}(\beta) = \mathbf{f}^{(n)}(-\beta) = u$. By definition of $\mathbf{f}_e^{(j-1)}$ and $\mathbf{f}_o^{(j-1)}$ (Definition 4.3.3), we have $\mathbf{f}^{(j-1)}(X) = \mathbf{f}_e^{(j-1)}(X^2) + X \cdot \mathbf{f}_o^{(j-1)}(X^2)$ and $\mathbf{f}^{(j)}(X) := \mathbf{f}_o^{(j-1)}(X) + \rho_{j-1} \cdot \mathbf{f}_e^{(j-1)}(X)$. Thus $\mathbf{f}^{(j-1)}(X) + \mathbf{f}^{(j-1)}(-X) = 2\mathbf{f}_e^{(j-1)}(X^2)$ and $\mathbf{f}^{(j-1)}(X) - \mathbf{f}^{(j-1)}(-X) = 2X\mathbf{f}_o^{(j-1)}(X^2)$. Evaluation at β and taking linear combinations of the last two equations shows that the consistency checks are satisfied. \square

Lemma 4.5.4. *Construction 1 has soundness error $\frac{N-1}{|\mathbb{F}^\times|}$.*

Proof. Suppose that $\langle \mathbf{f}, \otimes_j(1, \rho_j) \rangle \neq u$. Fix a malicious prover, which determines certain next-message functions $\tilde{\mathbf{f}}^{(1)}(X), \dots, \tilde{\mathbf{f}}^{(n-1)}(X)$, $e^{(0)}$ and $\bar{e}^{(0)}$. Set $\tilde{\mathbf{f}}^{(n)} = e^{(n)} = \bar{e}^{(n)} = u$, noting that $\tilde{\mathbf{f}}^{(j)}$ has degree at most $N/2^j$.

Since $\langle \mathbf{f}, \otimes_j(1, \rho_j) \rangle \neq u$ we must have $\mathbf{f}^{(j)}(X) \neq \mathbf{f}_e^{(j-1)}(X) + \rho_{j-1} \cdot \mathbf{f}_o^{(j-1)}(X)$, for some $j \in [n]$. Let j^* be the largest value of j for which this happens. This implies that:

$$\mathbf{p}(X) := \mathbf{f}^{(j^*)}(X^2) - \frac{\mathbf{f}^{(j^*-1)}(X) + \mathbf{f}^{(j^*-1)}(-X)}{2} - \rho_{j^*-1} \frac{\mathbf{f}^{(j-1)}(X) - \mathbf{f}^{(j^*-1)}(-X)}{2\beta}$$

is a non-zero polynomial of degree at most $2^{n-(j^*-1)} - 1$. Setting $X = Y^{2^{j^*-1}}$, and evaluating at $Y = \beta$, the probability that \mathbf{p} evaluates to 0 is at most $\frac{2^n - 1}{|\mathbb{F}^\times|}$. This implies that the verifier check in Equation (4.16) is satisfied with probability at most $\frac{N-1}{|\mathbb{F}^\times|}$. \square

Lemma 4.5.5. *The prover for Construction 1 can be implemented in $O(N)$ operations in \mathbb{F} , and space complexity $O(N)$.*

Proof. The prover requires $O(N/2^j)$ space to store $\mathbf{f}^{(j)}(X)$, and $N/2^j$ operations to compute $\mathbf{f}^{(j+1)}(X)$ from $\mathbf{f}^{(j)}(X)$. Summing up the prover's time complexity at each step gives $O(N)$ operations. \square

Lemma 4.5.6. *The prover for Construction 1 can be implemented in $O(N \log N)$ operations in \mathbb{F} , and space complexity $O(\log N)$ with $O(1)$ passes over \mathbf{f} .*

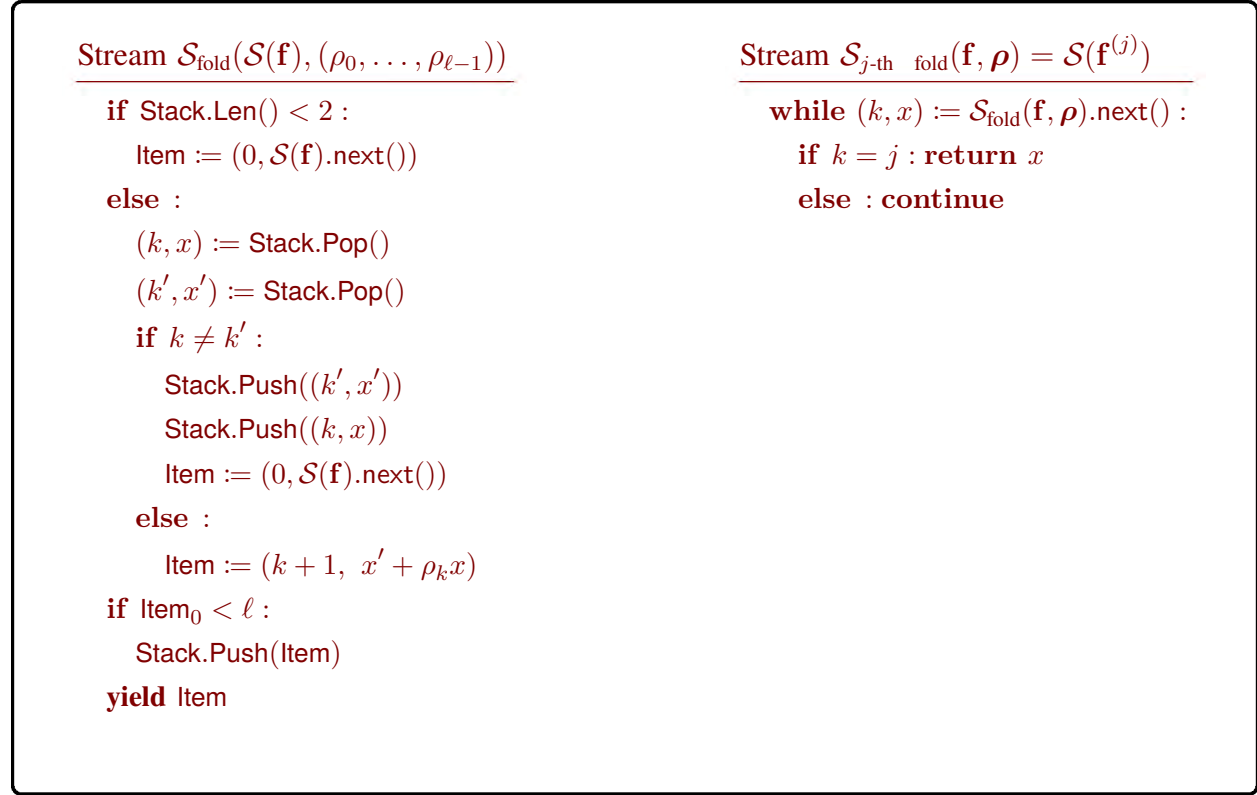


Figure 4.3: On the left-hand side, the stream for generating the coefficients of all *all* folded polynomials $\mathbf{f}^{(0)}, \dots, \mathbf{f}^{(n)}$, as a pair composed of the current round number, and the next coefficient. On the right, the stream for generating coefficients of the vector $\mathbf{f}^{(j)}$, given $\mathcal{S}(\mathbf{f})$ and $\boldsymbol{\rho} = (\rho_0, \dots, \rho_{n-1})$ with $n \geq j$.

Proof. Figure 4.3 gives an algorithm $\mathcal{S}_{\text{fold}}(\mathcal{S}(\mathbf{f}), j, \rho_0, \dots, \rho_{j-1})$ for producing streams of $\mathcal{S}(\mathbf{f}^{(0)}), \dots, \mathcal{S}(\mathbf{f}^{(j)})$ simultaneously. The indices k of items (k, x) in Stack form an ascending sequence with $k \leq j$. Whenever two items (k, x) and (k, x') are next to each other at the beginning of the sequence, they are merged using $O(1)$ field operations. The algorithm never adds an item with $k = j$ to the stack, so the stack never contains more than $j \leq \log N$ elements. To produce an item with $k = j$ requires passing through and merging together exactly 2^j elements of the stream of \mathbf{f} , and uses 2^j operations. Producing all $N/2^j$ elements of $\mathcal{S}(\mathbf{f}^{(j)})$ costs $N/2^j \cdot 2^j = N$ operations.

To produce the streams $\mathcal{S}(\mathbf{f}^{(0)}), \dots, \mathcal{S}(\mathbf{f}^{(n)})$ requires a single pass over $\mathcal{S}(\mathbf{f})$ and uses $O(N)$ operations. □

4.5.2 Batched tensor-product protocol

We present a protocol for checking m instances of \mathcal{R}_{TC} at the same time. The new protocol gives a query complexity which depends additively on m instead of multiplicatively.

Definition 4.5.7. The **batched tensor-product relation** \mathcal{R}_{BTC} is the set of tuples

$$(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) = \left(\perp, (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, \{u_i\}_{i=0}^{m-1}), (\mathbf{f}_0, \dots, \mathbf{f}_{m-1}) \right)$$

where $N = 2^n$, $\mathbf{f}_0, \dots, \mathbf{f}_{m-1} \in \mathbb{F}^N$, $u_0, \dots, u_{m-1} \in \mathbb{F}$, and $\langle \mathbf{f}_i, \otimes_j(1, \rho_j) \rangle = u_i$ for each $i \in [m]$.

Theorem 4.5.8. For every finite field \mathbb{F} and positive integer N , there is a holographic PIOP for the indexed relation \mathcal{R}_{BTC} that supports instances over \mathbb{F} , with:

time-efficient prover	space-efficient prover	verifier time	soundness error	round complexity	message complexity	query complexity
$O(mN)$ \mathbb{F} -ops	$O(N(\log N + m))$ \mathbb{F} -ops					
$O(mN)$ memory	$O(\log N)$ memory	$O(m + \log N)$ \mathbb{F} -ops	$O\left(\frac{m+N}{ \mathbb{F} }\right)$	$O(\log N)$	$O(m + \log N)$	$O(m + \log N)$
	$O(\log N)$ passes					

Construction 2. We construct a PIOP for the indexed relation \mathcal{R}_{BTC} . The prover \mathcal{P} takes as input an index $\mathfrak{i} = \perp$, instance $\mathfrak{x} = (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, \{u_i\}_{i=0}^{m-1})$, and witness $\mathfrak{w} = (\mathbf{f}_0, \dots, \mathbf{f}_{m-1})$; the verifier \mathcal{V} takes as input the index \mathfrak{i} and the instance \mathfrak{x} .

- The verifier samples random challenge $\zeta \in \mathbb{F}^\times$.
- The prover \mathcal{P} computes the polynomial $\mathbf{f}(X) = \sum_{i=0}^{m-1} \zeta^i \mathbf{f}_i(X)$ and sends it to the verifier.
- The prover and verifier run the tensor-product check with $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) = \left(\perp, (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, \sum_i u_i \zeta^i), \mathbf{f} \right)$ and randomness β_0 to check that $\langle \mathbf{f}, \otimes_j(1, \rho_j) \rangle = \sum_i u_i \zeta^i$.
- The verifier samples randomness β and makes an oracle query to learn $\mathbf{f}(\beta)$. For $i = 0, \dots, m-1$, the verifier makes oracle queries to learn $\mathbf{f}_i(\beta)$. The verifier checks whether $\mathbf{f}(\beta) = \sum_{i=0}^{m-1} \mathbf{f}_i(\beta) \zeta^i$.

Remark 4.5.9. Since the simple tensor-product check requires an evaluation of \mathbf{f} at a random point, Construction 2 can be optimized so that Construction 1 uses the same randomness β as Construction 2, which saves one evaluation query.

Lemma 4.5.10. Construction 2 has perfect completeness.

Proof. Suppose that $\langle \mathbf{f}_i, \otimes_j(1, \rho_j) \rangle = u_i$ for each i . By definition, $\mathbf{f} = \sum_{i=0}^{m-1} \zeta^i \mathbf{f}_i$. Querying each polynomial in this expression at β , it is clear that the verifier's check is satisfied.

Next, since $\langle \mathbf{f}_i, \otimes_j(1, \rho_j) \rangle = u_i$ for each i , taking a linear combination of these expressions using powers of ζ , it is clear that $\langle \mathbf{f}, \otimes_j(1, \rho_j) \rangle = \sum_i u_i \zeta^i$, so the basic tensor-product protocol accepts. \square

Lemma 4.5.11. Construction 2 has soundness error $\frac{m+N-2}{|\mathbb{F}^\times|}$.

Proof. Suppose that there is some i for which $\langle \mathbf{f}_i, \otimes_j(1, \rho_j) \rangle \neq u_i$. Fix a malicious prover, which determines next-message function $\tilde{\mathbf{f}}(X)$. Let $\mathbf{f}(X) = \sum_{i=0}^{m-1} \zeta^i \mathbf{f}_i(X)$. By the Schwartz–Zippel lemma, $\langle \mathbf{f}, \otimes_j(1, \rho_j) \rangle \neq \sum_{i=0}^{m-1} \zeta^i u_i$, except with probability at most $(m-1)/|\mathbb{F}^\times|$ over the random choice of ζ . If $\tilde{\mathbf{f}}(X) = \mathbf{f}(X)$, then by the soundness of the basic tensor-product protocol, the verifier accepts with probability at most $(N-1)/|\mathbb{F}^\times|$. If $\tilde{\mathbf{f}}(X) \neq \mathbf{f}(X)$, then by the Schwartz–Zippel lemma, $\mathbf{f}(\beta) \neq \sum_{i=0}^{m-1} \zeta^i \mathbf{f}_i(\beta)$, except with probability at most $(N-1)/|\mathbb{F}^\times|$ over the random choice of β . \square

Lemma 4.5.12. *Construction 2 can be implemented in $O(N(\log N + m))$ operations in \mathbb{F} , and space complexity $O(m + \log N)$ with $O(\log N)$ passes over each \mathbf{f}_i for $i \in [m]$.*

Proof. By Lemma 4.5.6, the basic tensor-product check requires $O(\log N)$ passes over $\mathcal{S}(\mathbf{f})$ and uses $O(\log N)$ memory. Since $\mathbf{f}(X) = \sum_{i=0}^{m-1} \zeta^i \mathbf{f}_i(X)$, the stream $\mathcal{S}(\mathbf{f})$ can be computed by a streaming algorithm that uses $O(mN)$ operations, $O(1)$ space (never storing more than a single coefficient of one \mathbf{f}_i polynomial and the partial computation of a coefficient of \mathbf{f}), and a single pass over each of its m inputs. Composing the two streaming algorithms and using Lemma 4.4.6 gives the result. \square

4.6 Elastic protocols for scalar products

We describe elastic PIOP protocols which reduce checking twisted scalar product relations to consistency checks of the type described in Section 4.5.

Definition 4.6.1. *The twisted scalar product relation \mathcal{R}_{TSP} is the set of tuples*

$$(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) = (\perp, (\mathbb{F}, N, u), (\mathbf{f}, \mathbf{y}, \mathbf{g}))$$

where $\mathbf{f}, \mathbf{y}, \mathbf{g} \in \mathbb{F}^N$, $u \in \mathbb{F}$, and $\langle \mathbf{f} \circ \mathbf{y}, \mathbf{g} \rangle = u$.

Standard scalar products are the special case where every entry of \mathbf{y} is equal to 1.

We give two elastic PIOPs for \mathcal{R}_{TSP} . The first, in Section 4.6.1, is a protocol for the special case where $\mathbf{y} := \otimes_{j=0}^{n-1} (1, v_j)$ for public v_0, \dots, v_{n-1} .

Theorem 4.6.2. *For every finite field \mathbb{F} , every $N \in \mathbb{N}$ with $n = \log N$, there is a holographic PIOP for the indexed relation \mathcal{R}_{TSP} with $\mathbf{y} := \otimes_{j=0}^{n-1} (1, v_j)$ for public $v_0, \dots, v_{n-1} \in \mathbb{F}$, with:*

time-efficient prover	space-efficient prover	verifier time	soundness error	round complexity	message complexity	query complexity
$O(N)$ \mathbb{F} -ops	$O(N \log N)$ \mathbb{F} -ops					
$O(N)$ memory	$O(\log N)$ memory $O(\log N)$ passes	$O(\log N)$ \mathbb{F} -ops	$O\left(\frac{N}{ \mathbb{F} }\right)$	$O(\log N)$	$O(\log N)$	$O(\log N)$

The second protocol, in Section 4.6.4, reduces the general case to the special case. Finally, in Section 4.6.5, we give a PIOP for Hadamard products of vectors, which also follows from the special case of twisted scalar products.

Definition 4.6.3. *The Hadamard product relation \mathcal{R}_{HP} is the set of tuples $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) = (\perp, (\mathbb{F}, N), (\mathbf{f}, \mathbf{g}, \mathbf{h}))$ where $\mathbf{f}, \mathbf{g}, \mathbf{h} \in \mathbb{F}^N$ and $\mathbf{f} \circ \mathbf{g} = \mathbf{h}$.*

Theorem 4.6.4. *For every finite field \mathbb{F} and positive integer N , there is a holographic PIOP for the indexed relation \mathcal{R}_{HP} that supports instances over \mathbb{F} , with:*

time-efficient prover	space-efficient prover	verifier time	soundness error	round complexity	message complexity	query complexity
$O(N)$ \mathbb{F} -ops	$O(N \log N)$ \mathbb{F} -ops					
$O(N)$ memory	$O(\log N)$ memory $O(\log N)$ passes	$O(\log N)$ \mathbb{F} -ops	$O\left(\frac{N}{ \mathbb{F} }\right)$	$O(\log N)$	$O(\log N)$	$O(\log N)$

4.6.1 Elastic scalar-product protocol (special case)

Construction 3. *We construct a PIOP for the indexed relation \mathcal{R}_{TSP} . The prover \mathcal{P} takes as input an index $\mathfrak{i} = \perp$, instance $\mathfrak{x} = (\mathbb{F}, v_0, v_1, \dots, v_{n-1}, u)$, and witness $\mathfrak{w} = (\mathbf{f}, \mathbf{g})$; the verifier \mathcal{V} takes as input the index \mathfrak{i} and the instance \mathfrak{x} .*

Interactive phase. Let $\mathcal{H} := \{-1, 1\}$. The prover and verifier run a **multivariate sumcheck** protocol on the multivariate polynomials $\widehat{\mathbf{f}}, \widehat{\mathbf{g}}$ associated with $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$ to show that

$$\frac{1}{2^n} \sum_{\omega \in \mathcal{H}^n} (\widehat{\mathbf{f}} \circ \widehat{\mathbf{y}} \cdot \widehat{\mathbf{g}})(\omega) = u . \quad (4.17)$$

The verifier \mathcal{V} for the multivariate sumcheck protocol outputs a claim that $(\widehat{\mathbf{f}} \circ \widehat{\mathbf{y}} \cdot \widehat{\mathbf{g}})(\rho_0, \dots, \rho_{n-1}) = u$, where $\rho_0, \dots, \rho_{n-1}$ is the verifier randomness used in the sumcheck protocol.

The prover sends claimed evaluations $u_A, u_B \in \mathbb{F}$ to the verifier, corresponding to the claims that $\widehat{\mathbf{f}} \circ \widehat{\mathbf{y}}(\rho_0, \dots, \rho_{n-1}) = u_A$ and $\widehat{\mathbf{g}}(\rho_0, \dots, \rho_{n-1}) = u_B$. The verifier checks whether $u_A \cdot u_B = u_C$.

Rewriting the first evaluation claim, the output of the interactive phase consists of the two claims:

$$\widehat{\mathbf{f}}(v_0 \rho_0, \dots, v_{n-1} \rho_{n-1}) = u_A , \quad \widehat{\mathbf{g}}(\rho_0, \dots, \rho_{n-1}) = u_B . \quad (4.18)$$

Query phase. The prover and the verifier run the **univariate PIOP** for tensor products from Construction 1 to check that the two claims from Equation (4.18) are true:

- one execution with $\mathbb{x} = (\mathbb{F}, N, v_0 \rho_0, \dots, v_{n-1} \rho_{n-1}, u_A)$ and $\mathbb{w} = \mathbf{f}$; and
- one execution with $\mathbb{x} = (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, u_B)$ and $\mathbb{w} = \mathbf{g}$.

Remark 4.6.5. The message complexity of the sumcheck protocol can be reduced from $3 \log N$ to $2 \log N$ field elements. In each round of the sumcheck protocol, the prover sends the coefficients of the polynomial $\mathbf{q}^{(j)}$, and the verifier checks whether $\mathbf{q}^{(j)}(1) + \mathbf{q}^{(j)}(-1) = 2 \cdot \mathbf{q}^{(j-1)}(\rho_{j-1})$. Since $\mathbf{q}^{(j)}(X) = q_{j,0} + q_{j,1}X + q_{j,2}X^2$ is quadratic, $\mathbf{q}^{(j)}(1) + \mathbf{q}^{(j)}(-1) = 2q_{j,0} + 2q_{j,2}$, and the verification checks amount to checking whether $2q_{j,0} + 2q_{j,2} = \mathbf{q}^{(j-1)}(\rho_{j-1})$. Thus, instead of having the prover send $q_{j,1}$ in each round, and asking the verifier to perform the aforementioned checks, the verifier can simply use $\mathbf{q}^{(j-1)}(\rho_{j-1})$ (which is known from the previous round, or equal to u when $j = 0$) as the *definition* of the value of $q_{j,1}$.

Remark 4.6.6. Construction 3 is a univariate PIOP and uses Construction 1 to reduce claims about multivariate polynomial evaluations, which can be rewritten as tensor products, to claims about univariate polynomial evaluations. However, one can convert Construction 3 into a multivariate PIOP by concluding the protocol using a multivariate evaluation query rather than invoking Construction 1. This means that one can compile our PIOPs into succinct arguments using either multivariate or univariate polynomial commitment schemes.

4.6.2 Proof of Theorem 4.6.2

We prove completeness in Lemma 4.6.9. We prove soundness in Lemma 4.6.10. We analyse the complexity of the time-efficient prover and the verifier in Lemma 4.6.11. We analyse the complexity of the space-efficient prover in Lemma 4.6.12.

In our analysis, we rely on results from prior work, stated in two lemmas. The first relates polynomial coefficients to sums of evaluations.

Lemma 4.6.7 ([BCG20, Lemma 5.7]). *Let \mathcal{H} be a multiplicative subgroup of a finite field \mathbb{F} and let $\mathbf{p}(X_0, \dots, X_{n-1}) \in \mathbb{F}[X_0, \dots, X_{n-1}]$. If we denote by $p_{i_0, \dots, i_{n-1}} \in \mathbb{F}$ the coefficient of $X_0^{i_0} \cdots X_{n-1}^{i_{n-1}}$ in the polynomial $\mathbf{p}(X_0, \dots, X_{n-1})$, then*

$$\sum_{\omega \in \mathcal{H}^n} \mathbf{p}(\omega) = \left(\sum_{\mathbf{i} \equiv \mathbf{0} \pmod{|\mathcal{H}|}} p_{\mathbf{i}} \right) \cdot |\mathcal{H}|^n . \quad (4.19)$$

The second describes the properties of sumcheck protocol for products of multilinear polynomials, as studied in [Tha13; XZZPS19; BCG20].

Lemma 4.6.8. *For every finite field \mathbb{F} and every $N \in \mathbb{N}$ with $n = \log N$, the sumcheck protocol for*

$$\frac{1}{2^n} \sum_{\omega \in \mathcal{H}^n} (\widehat{\mathbf{f}} \cdot \widehat{\mathbf{g}})(\omega) = u . \quad (4.20)$$

for $\mathcal{H} = \{-1, 1\}$ and the multilinear polynomials $\widehat{\mathbf{f}}(X_0, \dots, X_{n-1})$ and $\widehat{\mathbf{g}}(X_0, \dots, X_{n-1})$ associated with $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$ has the following properties: soundness error is $O(n/|\mathbb{F}|)$ (as a reduction to claims about polynomial evaluations); round complexity is $O(n)$; the prover uses $O(N)$ field operations; and the verifier uses $O(n)$ field operations.

Lemma 4.6.9. *Construction 3 has perfect completeness.*

Proof. Suppose that $\langle \mathbf{f} \circ \mathbf{y}, \mathbf{g} \rangle = u$. By Lemma 4.6.7, $\frac{1}{2^n} \sum_{\omega \in \mathcal{H}^n} (\widehat{\mathbf{f}} \circ \widehat{\mathbf{y}} \cdot \widehat{\mathbf{g}})(\omega) = \left(\sum_{\mathbf{i} \equiv \mathbf{0} \pmod{2}} (\widehat{\mathbf{f}} \circ \widehat{\mathbf{y}} \cdot \widehat{\mathbf{g}})_{\mathbf{i}} \right)$.

Since $\widehat{\mathbf{f}} \circ \widehat{\mathbf{y}}$ and $\widehat{\mathbf{g}}$ are multilinear polynomials, the contributions to the coefficients of $\widehat{\mathbf{f}} \circ \widehat{\mathbf{y}} \cdot \widehat{\mathbf{g}}$ with $\mathbf{i} \equiv \mathbf{0} \pmod{2}$ are exactly the terms $\widehat{\mathbf{f}} \circ \widehat{\mathbf{y}}_{\mathbf{j}} \cdot \widehat{\mathbf{g}}_{\mathbf{j}}$ where $\mathbf{j} \in \{0, 1\}^n$ is the unique vector such that $\mathbf{i} = 2\mathbf{j}$. Hence, $\frac{1}{2^n} \sum_{\omega \in \mathcal{H}^n} (\widehat{\mathbf{f}} \circ \widehat{\mathbf{y}} \cdot \widehat{\mathbf{g}})(\omega) = \langle \mathbf{f} \circ \mathbf{y}, \mathbf{g} \rangle = u$.

By the completeness property of the sumcheck protocol (Lemma 4.6.8), the claims $\widehat{\mathbf{f}}(v_0 \rho_0, \dots, v_{n-1} \rho_{n-1}) = u_A$ and $\widehat{\mathbf{g}}(\rho_0, \dots, \rho_{n-1}) = u_B$ are true.

By completeness of the PIOP for tensor products, Construction 1, the verifier accepts. \square

Lemma 4.6.10. *Construction 3 has soundness error $\epsilon_{SP} := O(N/|\mathbb{F}|)$.*

Proof. Suppose that $\langle \mathbf{f} \circ \mathbf{y}, \mathbf{g} \rangle \neq u$. By the soundness of the sumcheck protocol, the probability that the sumcheck verifier accepts and the output claims are both true is at most $\frac{2n}{|\mathbb{F}|}$. If the claims produced by the sumcheck protocol are not true, then by the soundness of the tensor-product protocol (Construction 1), the probability that the verifier for the tensor-product protocol accepts is at most $\frac{N}{|\mathbb{F}|}$. The result follows by a union bound. \square

Lemma 4.6.11. *The prover in Construction 3 has arithmetic complexity $O(N)$. The verifier in Construction 3 has arithmetic complexity $O(\log N)$.*

Proof. This follows from the arithmetic complexities of the prover and the verifier in the sumcheck protocol given in Lemma 4.6.8, and those of the tensor-product protocol given in Theorem 4.5.2. \square

4.6.3 Space efficient realization of Construction 3

Lemma 4.6.12. *The prover for Construction 3 can be implemented using a streaming algorithm with arithmetic complexity $O(N \log N)$ that makes $O(\log N)$ passes over the streams $\mathcal{S}(\mathbf{f})$ and $\mathcal{S}(\mathbf{g})$.*

Remark 4.6.13. Lemma 4.6.12 follows directly from [CTY11; CMT12], which proves results about streaming provers for interactive proofs for uniform circuits. We provide a direct proof below for the special case of twisted scalar products.

The techniques used to prove Lemma 4.6.12 are related to those in [BHRRS20], used for messages for a split-and-fold protocol that was shown to be closely related to the sumcheck protocol in [BCS21]. Our algorithm has a prover arithmetic complexity of $O(N \log N)$, which is more efficient than the $O(N \log^2 N)$ algorithm given in [BHRRS20] and has a significant impact on concrete efficiency. We must also carefully modify our protocol to account for the vector \mathbf{y} to reason about a twisted scalar-product relation, which requires extra thought.

Proof. We show how to implement the space-efficient prover for Construction 3 using streaming access to the inputs \mathbf{f} and \mathbf{g} . By Lemma 4.5.6, the tensor product protocol can be implemented in space $O(\log N)$ using $O(1)$ passes. It remains to show how to compute the next-message function for the sumcheck protocol in small space. Define partially evaluated polynomials in $\mathbb{F}[X_j, X_{j+1}, \dots, X_n]$

$$\begin{aligned} \mathbf{f}^{(j)}(X_j, \dots, X_{n-1}) &:= \widehat{\mathbf{f} \circ \mathbf{y}}(\rho_0, \dots, \rho_{j-1}, X_j, \dots, X_{n-1}) = \sum_{i_j, \dots, i_{n-1} \in \{0,1\}} \mathbf{f}_{i_j, \dots, i_{n-1}}^{(j)} X_j^{i_j} \cdots X_{n-1}^{i_{n-1}}, \\ \mathbf{g}^{(j)}(X_j, \dots, X_{n-1}) &:= \widehat{\mathbf{g}}(\rho_0, \dots, \rho_{j-1}, X_j, \dots, X_{n-1}) = \sum_{i_j, \dots, i_{n-1} \in \{0,1\}} \mathbf{g}_{i_j, \dots, i_{n-1}}^{(j)} X_j^{i_j} \cdots X_{n-1}^{i_{n-1}}. \end{aligned}$$

In the $(j+1)$ -th round of the sumcheck protocol, the prover \mathcal{P} sends the polynomial

$$\begin{aligned} \mathbf{q}^{(j)}(X) &:= \frac{1}{2^{n-(j+1)}} \sum_{\omega_{j+1}, \dots, \omega_{n-1} \in \mathcal{H}} (\widehat{\mathbf{f} \circ \mathbf{y}} \cdot \widehat{\mathbf{g}})(\rho_0, \dots, \rho_{j-1}, X_j, \omega_{j+1}, \dots, \omega_{n-1}) \\ &= \frac{1}{2^{n-(j+1)}} \sum_{\omega_{j+1}, \dots, \omega_{n-1} \in \mathcal{H}} (\mathbf{f}^{(j)} \cdot \mathbf{g}^{(j)})(X_j, \omega_{j+1}, \dots, \omega_{n-1}). \end{aligned}$$

By Lemma 4.6.7, we see that

$$\mathbf{q}^{(j)}(X) = \sum_{i_j, k_j \in \{0,1\}} \sum_{i_{j+1} \in \{0,1\}} \cdots \sum_{i_{n-1} \in \{0,1\}} \mathbf{f}_{i_j, i_{j+1}, \dots, i_{n-1}}^{(j)} \cdot \mathbf{g}_{k_j, i_{j+1}, \dots, i_{n-1}}^{(j)} X_j^{i_j + k_j}.$$

This implies that

$$\begin{aligned} q_{j,0} &= \left(\sum_{i_{j+1} \in \{0,1\}} \cdots \sum_{i_{n-1} \in \{0,1\}} \mathbf{f}_{0,i_{j+1},\dots,i_{n-1}}^{(j)} \cdot \mathbf{g}_{0,i_{j+1},\dots,i_{n-1}}^{(j)} \right), \\ q_{j,1} &= \left(\sum_{i_j \in \{0,1\}} \cdots \sum_{i_{n-1} \in \{0,1\}} \mathbf{f}_{i_j,i_{j+1},\dots,i_{n-1}}^{(j)} \cdot \mathbf{g}_{1-i_j,i_{j+1},\dots,i_{n-1}}^{(j)} \right), \text{ and} \\ q_{j,2} &= \left(\sum_{i_{j+1} \in \{0,1\}} \cdots \sum_{i_{n-1} \in \{0,1\}} \mathbf{f}_{1,i_{j+1},\dots,i_{n-1}}^{(j)} \cdot \mathbf{g}_{1,i_{j+1},\dots,i_{n-1}}^{(j)} \right). \end{aligned}$$

From these formulae, it is clear that $q_{j,0}$, $q_{j,1}$ and $q_{j,2}$, can be computed simultaneously in $O(1)$ memory using streams for $\mathbf{f}^{(j)}$ and $\mathbf{g}^{(j)}$.

Next, we show that streams of $\mathbf{f}^{(j)}$ and $\mathbf{g}^{(j)}$ can be generated in $O(\log N)$ memory and $O(N)$ operations using a single pass over the streams of \mathbf{f} or \mathbf{g} . The algorithm $\mathcal{S}_{\text{fold}}(\cdot)$ described in Figure 4.3 and analysed in Lemma 4.5.6 generates the stream of $\mathbf{g}^{(j)}$ from the stream of \mathbf{g} for any j with the required complexity parameters. Similarly, $\mathcal{S}_{\text{fold}}(\cdot)$ generates $\mathbf{f}^{(j)}$ from the stream of $\mathbf{f} \circ \mathbf{y}$ for any j . To complete the proof, we explain how to generate the stream of $\mathbf{f} \circ \mathbf{y}$ using a single pass over the stream of \mathbf{f} , and $v_0 \dots, v_{n-1}$, using $O(\log N)$ space.

The entries of $\mathbf{f} \circ \mathbf{y}$ are given by $(\mathbf{f}_{i_0,\dots,i_{n-1}} v_0^{i_0} \cdots v_{n-1}^{i_{n-1}})_{i_0,\dots,i_{n-1} \in \{0,1\}}$. To generate the stream of $\mathbf{f} \circ \mathbf{y}$ in a single pass over the stream of \mathbf{f} , begin by computing the sequence $v_0, v_0 v_1, \dots, v_0 \cdots v_{n-1}$. Then, compute the sequence $d_0 := v_0 v_1^{-1}, d_1 := v_0 v_1 v_2^{-1}, \dots, d_{n-1} := v_0 \cdots v_{n-1} v_n^{-1}$. Next, define $\mathcal{S}^{(n)}$ as follows. Let $\mathcal{S}^{(0)} := (v_0)$, and define $\mathcal{S}^{(j+1)}$ recursively as the concatenation $\mathcal{S}^{(j)} \parallel (d_j) \parallel \mathcal{S}^{(j)}$. Now, the sequence $(\mathbf{y}_{i_0,\dots,i_{n-1}})_{i_0,\dots,i_{n-1}}$ can be computed by starting with $z = v_0 v_1 \cdots v_{n-1}$ (which is equal to $\mathbf{y}_{1,\dots,1}$), and multiplying z by each element of the palindromic sequence $\mathcal{S}^{(n)} \in \mathbb{F}^{N-1}$ in turn.

This method generates the stream of \mathbf{y} in $O(\log N)$ space using $O(N)$ operations, and multiplying this stream by the stream of \mathbf{f} gives the stream of $\mathbf{f} \circ \mathbf{y}$ with the stated time and memory complexity. \square

4.6.4 Elastic scalar-product protocol (general case)

To verify \mathcal{R}_{TSP} , the prover and verifier begin by running one scalar-product subprotocol on $\mathbf{f} \circ \mathbf{g}$ and \mathbf{h} . This reduces the claim that $\langle \mathbf{f} \circ \mathbf{g}, \mathbf{h} \rangle = u$ to claims about $\mathbf{f} \circ \mathbf{g}$ and \mathbf{h} . Then, the claim about $\mathbf{f} \circ \mathbf{g}$ can be rewritten as claim about scalar-product between \mathbf{f} and \mathbf{g} . Finally, the prover and verifier run a second scalar-product subprotocol to reduce this to claims about \mathbf{f} and \mathbf{g} . The end result is a claim about each of \mathbf{f} , \mathbf{g} and \mathbf{h} .

Construction 4. We construct a PIOP for the indexed relation \mathcal{R}_{TSP} . The prover \mathcal{P} takes as input an index $\mathfrak{i} = \perp$, instance $\mathfrak{x} = (\mathbb{F}, N, u)$, and witness $\mathfrak{w} = (\mathbf{f}, \mathbf{g}, \mathbf{h})$; the verifier \mathcal{V} takes as input the index \mathfrak{i} and the instance \mathfrak{x} .

Interactive phase. Letting $\mathbf{f}^{(0)}(X) := \mathbf{f} \circ \mathbf{g}(X)$, $\mathbf{g}^{(0)}(X) := \mathbf{h}(X)$, $\mathbf{h}^{(0)}(X) := \mathbf{f}(X)$, and $\mathbf{k}^{(0)}(X) := \mathbf{g}(X)$ the protocol proceeds as follows.

- The prover and verifier run the interactive phase of the scalar-product protocol (Construction 3) with index $\mathfrak{i} = \perp$, instance $\mathfrak{x} = (\mathbb{F}, N, u)$, and witness $\mathfrak{w} = (\mathbf{f} \circ \mathbf{g}, \mathbf{h})$ to check that $\langle \mathbf{f} \circ \mathbf{g}, \mathbf{h} \rangle = u$. The protocol outputs claims that $u_A = \langle \mathbf{f} \circ \mathbf{g}, \otimes_j(1, \rho_j) \rangle$ and $u_B = \langle \mathbf{h}, \otimes_j(1, \rho_j) \rangle$, where $\rho_0, \dots, \rho_{n-1}$ is the verifier randomness used in the subprotocol.
- The prover and verifier run the interactive phase of the scalar-product protocol (Construction 3) with index $\mathfrak{i} = \perp$, instance $\mathfrak{x} = (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, u_A)$, and witness $\mathfrak{w} = (\mathbf{f} \circ \otimes_j(1, \rho_j), \mathbf{g})$ to check that $\langle \mathbf{f} \circ \otimes_j(1, \rho_j), \mathbf{g} \rangle = u_A$. The protocol outputs claims that $u_C = \langle \mathbf{f}, \otimes_j(1, \rho_j r_j) \rangle$ and $u_D = \langle \mathbf{g}, \otimes_j(1, r_j) \rangle$, where r_0, \dots, r_{n-1} is the verifier randomness used in the subprotocol.
- The output of the interactive phase consists of the three claims:

$$u_C = \langle \mathbf{f}, \otimes_j(1, \rho_j r_j) \rangle, \quad u_D = \langle \mathbf{g}, \otimes_j(1, r_j) \rangle, \quad u_B = \langle \mathbf{h}, \otimes_j(1, \rho_j) \rangle. \quad (4.21)$$

Query phase. The prover and the verifier run the **univariate PIOP** for tensor products from Construction 1 to check that the three claims from Equation (4.21) are true:

- one execution with $\mathfrak{x} = (\mathbb{F}, N, \rho_0 r_0, \dots, \rho_{n-1} r_{n-1}, u_C)$ and $\mathfrak{w} = \mathbf{f}$;
- one execution with $\mathfrak{x} = (\mathbb{F}, N, r_0, \dots, r_{n-1}, u_D)$ and $\mathfrak{w} = \mathbf{g}$; and
- one execution with $\mathfrak{x} = (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, u_B)$ and $\mathfrak{w} = \mathbf{h}$.

Lemma 4.6.14. *Construction 4 has perfect completeness.*

Sketch. Suppose that $\langle \mathbf{f} \circ \mathbf{g}, \mathbf{h} \rangle = u$. By Lemma 4.6.9 (completeness of Construction 3), the verifier for the first scalar-product subprotocol will accept, and the subprotocol produces correct claims about u_A and u_B . Writing $u_A = \langle \mathbf{f} \circ \mathbf{g}, \otimes_j(1, \rho_j) \rangle = \langle \mathbf{f} \circ \otimes_j(1, \rho_j), \mathbf{g} \rangle$ and applying similar reasoning to the second scalar-product subprotocol completes the proof. \square

Lemma 4.6.15. *Construction 4 has soundness error $2\epsilon_{SP}$.*

Proof. Suppose that $\langle \mathbf{f} \circ \mathbf{g}, \mathbf{h} \rangle \neq u$. By Lemma 4.6.10 (soundness of Construction 3), the probability that the verifier for the first scalar-product subprotocol accepts and that the claims about u_A and u_B are true is at most ϵ_{SP} . If the claim about u_A is false, then by Lemma 4.6.10, the probability that the verifier for the second scalar-product subprotocol accepts and that the claims about u_C and u_D are true is at most ϵ_{SP} .

Therefore, by a union bound, except with probability at most $2\epsilon_{SP}$, at least one of the claims about u_B , u_C or u_D is false, or the verifier rejects. \square

Lemma 4.6.16. *The prover in Construction 4 has arithmetic complexity $O(N)$, and the verifier has arithmetic complexity $O(\log N)$.*

Sketch. By Lemma 4.6.11, the prover in Construction 3 has arithmetic complexity $O(N)$ and the verifier in Construction 3 has arithmetic complexity $O(\log N)$. Construction 4 consists of two executions of Construction 3, so the result follows. \square

Lemma 4.6.17. *Construction 4 has an implementation with a streaming prover that uses $O(N \log N)$ arithmetic operations over \mathbb{F} , $O(\log N)$ space, and $O(\log N)$ passes over $\mathcal{S}(\mathbf{f})$, $\mathcal{S}(\mathbf{g})$ and $\mathcal{S}(\mathbf{h})$.*

Proof. This follows immediately from the space-efficient implementation of Construction 3 and its analysis in Lemma 4.6.12. \square

4.6.5 Hadamard-product protocol

We describe a Hadamard product protocol.

Construction 5. *We construct a PIOP for the indexed relation \mathcal{R}_{HP} . The prover \mathcal{P} takes as input an index $\mathfrak{i} = \perp$, instance $\mathfrak{x} = (\mathbb{F}, N)$, and witness $\mathfrak{w} = (\mathbf{f}, \mathbf{g}, \mathbf{h})$; the verifier \mathcal{V} takes as input the index \mathfrak{i} and the instance \mathfrak{x} .*

Interactive phase. *The protocol proceeds as follows.*

- *The verifier \mathcal{V} sends uniformly random challenge $v \in \mathbb{F}^\times$.*
- *The prover \mathcal{P} computes $u = \langle \mathbf{h}, \mathbf{y} \rangle$, where $\mathbf{y} := (1, v, \dots, v^{N-1})$, and sends the non-oracle message $u \in \mathbb{F}$. Note that in the space-efficient variant, the prover need not compute \mathbf{y} from v explicitly.*
- *The prover \mathcal{P} and verifier \mathcal{V} engage in a twisted scalar-product protocol with $\mathfrak{w} := (\mathbf{f}, \mathbf{g})$ and $\mathfrak{x} := (\mathbb{F}, N, v, u)$ to show that $\langle \mathbf{f} \circ \mathbf{y}, \mathbf{g} \rangle = u$. The twisted scalar product protocol outputs claims $u_A = \langle \mathbf{f}, \mathbf{y} \circ \otimes_j(1, \rho_j) \rangle$ and $u_B = \langle \mathbf{g}, \otimes_j(1, \rho_j) \rangle$.*

The output of the interactive phase consists of three claims

$$u_A = \langle \mathbf{f}, \mathbf{y} \circ \otimes_j(1, \rho_j) \rangle, \quad u_B = \langle \mathbf{g}, \otimes_j(1, \rho_j) \rangle, \quad u = \langle \mathbf{h}, \mathbf{y} \rangle.$$

Query phase. *The prover and the verifier run the **univariate** PIOP for tensor products from Construction 1 to check that the three claims from Equation (4.21) are true:*

- *one execution with $\mathfrak{x} = (\mathbb{F}, N, v^{2^0} \rho_0, \dots, v^{2^{n-1}} \rho_{n-1}, u_A)$ and $\mathfrak{w} = \mathbf{f}$;*
- *one execution with $\mathfrak{x} = (\mathbb{F}, N, \rho_0, \dots, \rho_{n-1}, u_B)$ and $\mathfrak{w} = \mathbf{g}$; and*
- *one execution with $\mathfrak{x} = (\mathbb{F}, N, v^{2^0}, \dots, v^{2^{n-1}}, u)$ and $\mathfrak{w} = \mathbf{h}$.*

Theorem 4.6.4 follows in a straightforward manner from Theorem 4.6.2, with soundness using the Schwartz–Zippel lemma.

4.7 A non-holographic protocol for R1CS

We present a non-holographic protocol for $\mathcal{R}_{\text{R1CS}}$.

Theorem 4.7.1. *For every finite field \mathbb{F} and positive integer N , there is a PIOP for the indexed relation $\mathcal{R}_{\text{R1CS}}$ for instances with $N \times N$ matrices with M non-zero entries, with the following complexity parameters:*

time-efficient prover	space-efficient prover	time-efficient verifier	space-efficient verifier	soundness error	round complexity	proof length	query complexity
$O(M)$ \mathbb{F} -ops	$O(M \log^2 N)$ \mathbb{F} -ops	$O(M)$ \mathbb{F} -ops	$O(M \log N)$ \mathbb{F} -ops				
$O(M)$ memory	$O(\log N)$ memory $O(\log N)$ passes	$O(M)$ memory	$O(\log N)$ memory $O(1)$ passes	$O\left(\frac{N}{ \mathbb{F} }\right)$	$O(\log N)$	$O(\log N)$	$O(\log N)$

Note that Theorem 4.7.1 features an elastic *verifier* as well as an elastic prover.

Construction 6. *We construct a PIOP for the indexed relation $\mathcal{R}_{\text{R1CS}}$. The indexer algorithm is trivial. The prover \mathcal{P} takes as input an index $\mathfrak{i} = (\mathbb{F}, N, M, A, B, C)$, instance $\mathfrak{x} = \mathbf{x}$, and witness $\mathfrak{w} = \mathbf{w}$; the verifier \mathcal{V} takes as input the index \mathfrak{i} and the instance \mathfrak{x} . The protocol proceeds as follows.*

- The prover \mathcal{P} sets $\mathbf{z} := (\mathbf{x}, \mathbf{w}) \in \mathbb{F}^N$ and sends the oracle message \mathbf{w} to the verifier. The prover computes $A\mathbf{z}, B\mathbf{z}, C\mathbf{z} \in \mathbb{F}^N$.
- The prover \mathcal{P} and verifier \mathcal{V} run the interactive phase of the Hadamard-product protocol (Construction 5) with $\mathfrak{x} := (\mathbb{F}, N)$ and $\mathfrak{w} := (A\mathbf{z}, B\mathbf{z}, C\mathbf{z})$ to show that $A\mathbf{z} \circ B\mathbf{z} = C\mathbf{z}$. This generates claims that $u_A = \langle A\mathbf{z}, \mathbf{y} \circ \otimes_j(1, \rho_j) \rangle$, $u_B = \langle B\mathbf{z}, \otimes_j(1, \rho_j) \rangle$ and $u_C = \langle C\mathbf{z}, \mathbf{y} \rangle$ for verifier randomness $\rho_0, \dots, \rho_{n-1}, v \in \mathbb{F}$ and $\mathbf{y} := (1, v, \dots, v^{N-1})$. Rewrite these claims as

$$u_A = \langle \mathbf{z}, \mathbf{a}^* \rangle, \quad (4.22)$$

$$u_B = \langle \mathbf{z}, \mathbf{b}^* \rangle, \quad (4.23)$$

$$u_C = \langle \mathbf{z}, \mathbf{c}^* \rangle, \quad (4.24)$$

where $\mathbf{a}^* := \mathbf{y}^\top \circ \otimes_j(1, \rho_j)^\top A$, $\mathbf{b}^* := \otimes_j(1, \rho_j)^\top B$ and $\mathbf{c}^* := \mathbf{y}^\top C$.

- Note that to run the holographic protocol, the prover and verifier use Construction 7 instead of what follows from this point.

The verifier \mathcal{V} samples random challenge $\eta \leftarrow \mathbb{F}^\times$ and sends η to the prover \mathcal{P} . The challenge is used to bundle the three claims into one:

$$u_A + \eta \cdot u_B + \eta^2 \cdot u_C = \langle \mathbf{z}, \mathbf{a}^* + \eta \cdot \mathbf{b}^* + \eta^2 \cdot \mathbf{c}^* \rangle.$$

The prover and verifier run the interactive phase of the scalar-product protocol (Construction 3) to check this claim. This produces two claims:

– a claim

$$u_D = \langle \mathbf{z}, \otimes_j(1, \rho'_j) \rangle$$

about \mathbf{z} which the verifier can check using the tensor product protocol (Construction 1) with $\mathbb{x} = (\mathbb{F}, N, \rho'_0, \dots, \rho'_{n-1}, u_D)$ and $\mathbb{w} = \mathbf{z}$;

– a claim

$$u_E = \langle \mathbf{a}^* + \eta \cdot \mathbf{b}^* + \eta^2 \cdot \mathbf{c}^*, \otimes_j(1, \rho'_j) \rangle$$

which the verifier can check for themselves. Here, $\rho'_0, \dots, \rho'_{n-1} \in \mathbb{F}$ is the verifier randomness used in the scalar-product protocol.

4.7.1 Proof of Theorem 4.7.1

Completeness follows from the completeness of each subprotocol and Equations (4.22) to (4.24). The soundness error of the protocol and query complexity follow from those of the scalar-product protocol, Hadamard product protocol and consistency-check protocol. We analyse the complexity of the time-efficient prover in Lemma 4.7.2. We analyse the complexity of the space-efficient prover in Lemma 4.7.3. We analyse the complexity of the time-efficient prover in Lemma 4.7.4. We analyse the complexity of the space-efficient prover in Lemma 4.7.5.

Lemma 4.7.2. *The prover in Construction 6 can be implemented with arithmetic complexity $O(M)$ and $O(M)$ memory.*

Proof sketch. The prover \mathcal{P} computes $A\mathbf{z}$, $B\mathbf{z}$ and $C\mathbf{z}$, which uses $O(M)$ arithmetic operations and $O(M)$ space. The prover \mathcal{P} uses the Hadamard-product protocol, scalar-product protocol and consistency-check protocol as subroutines, running the protocols on vectors of length N . This uses $O(N)$ arithmetic operations and $O(N)$ memory. \square

Lemma 4.7.3. *The prover in Construction 6 can be implemented with arithmetic complexity $O(M \log^2 N)$, $O(\log N)$ memory, and $O(\log N)$ passes over the streams $\mathcal{S}(\mathbf{z})$, $\mathcal{S}(A\mathbf{z})$, $\mathcal{S}(B\mathbf{z})$ and $\mathcal{S}(C\mathbf{z})$, and $\mathcal{S}_{\text{cmaj}}(U)$ for $U \in \{A, B, C\}$.*

Proof sketch. We explain how to implement the space-efficient prover for Construction 6. Recall that $\mathcal{S}_{\text{cmaj}}(U)$ provides streaming access to the vectors row_U , col_U and $\text{val}_U \in \mathbb{F}^M$ of row indices, column indices and non-zero entries of $U \in \{A, B, C\}$ in column major order.

The prover \mathcal{P} uses the Hadamard-product protocol, scalar-product protocol and tensor-product protocol as subroutines, running the protocols on vectors of length N . These protocols use $O(N \log N)$ arithmetic operations, $O(\log N)$ memory and $O(\log N)$ passes over the witnesses for the protocols.

The prover has direct access to streams for the witnesses for the Hadamard-product protocol and the consistency check. However, one of the witnesses for the scalar-product protocol is the vector $\mathbf{a}^* + \eta \cdot \mathbf{b}^* + \eta^2 \mathbf{c}^*$ where $\mathbf{a}^* = \mathbf{y}^\top \circ \otimes_j(1, \rho_j)^\top A$, $\mathbf{b}^* = \otimes_j(1, \rho_j)^\top B$ and $\mathbf{c}^* = \mathbf{y}^\top C$.

The prover can generate the streams of \mathbf{a}^* , \mathbf{b}^* and \mathbf{c}^* using the algorithm in Figure 4.4. The prover multiplies each of the M non-zero elements of A , B and C by $O(\log N)$ field elements

(the seeds used to generate vectors such as $\otimes_j(1, \rho_j)$). Since the Hadamard-product prover makes $O(\log N)$ passes over streams of the witness (e.g. \mathbf{z}_A), the prover uses $O(M \log^2 N)$ arithmetic operations. \square

Stream $\mathcal{S}(\mathbf{u}^*) = \mathcal{S}_{\text{mat-prod}}(\boldsymbol{\rho}, U)$	Stream $\mathcal{S}(\mathbf{y}) = \mathcal{S}_{\text{tensor}}(\boldsymbol{\rho})$
<p>Initialize:</p> <p style="padding-left: 20px;">$j' := N$</p> <p style="padding-left: 20px;">$(i, j, a) := \mathcal{S}_{\text{cmaj}}(U).\text{next}()$</p> <p>Next element:</p> <p style="padding-left: 20px;">Item = 0; Decrement j'</p> <p style="padding-left: 40px;">/ Accumulate the inner product until end of line is reached</p> <p>while ($j' = j$) :</p> <p style="padding-left: 40px;">/ Let b_k be the k-th bit of i, for $k \in [n]$</p> <p style="padding-left: 40px;">/ $t = y_i$ could be computed more efficiently</p> <p style="padding-left: 40px;">/ in specific RICS instances</p> <p style="padding-left: 20px;">$t := \prod_k \rho_k^{b_k}$</p> <p style="padding-left: 20px;">Item := Item + $t \cdot a$</p> <p style="padding-left: 20px;">$(i, j, a) := \mathcal{S}_{\text{cmaj}}(U).\text{next}()$</p> <p>yield Item</p>	<p>Initialize:</p> <p style="padding-left: 20px;">$j := N$</p> <p style="padding-left: 40px;">/ Let b_k be the k-th bit of N</p> <p style="padding-left: 20px;">$t := \prod_k \rho_k^{b_k}$</p> <p style="padding-left: 20px;">carry := $\left[\rho_i^{-1} \prod_{k < i} \rho_k \right]_{i \in [n]}$</p> <p>Next element:</p> <p style="padding-left: 20px;">Decrement j</p> <p style="padding-left: 40px;">/ k is the position of the borrow</p> <p style="padding-left: 20px;">$k := \text{least bit set of } j$</p> <p style="padding-left: 20px;">$t := t \cdot \text{carry}_k$</p> <p>yield t</p>

Figure 4.4: Streams $\mathcal{S}(\mathbf{a}^*)$, $\mathcal{S}(\mathbf{b}^*)$, $\mathcal{S}(\mathbf{c}^*)$, and $\mathcal{S}(\mathbf{y})$.

Lemma 4.7.4. *The verifier in Construction 6 can be implemented with $O(M)$ field operations.*

Proof. The verifier \mathcal{V} uses the scalar-product protocol and consistency-check protocol as subroutines. The dominant cost for the verifier is the computation of the claim

$$u_E = \langle \mathbf{a}^* + \eta \cdot \mathbf{b}^* + \eta^2 \mathbf{c}^*, \otimes_j(1, \rho'_j) \rangle = \langle (\mathbf{y}^\top \circ \otimes_j(1, \rho_j)^\top) \cdot A + \eta \otimes_j(1, \rho_j)^\top B + \eta^2 \mathbf{y}^\top C, \otimes_j(1, \rho'_j) \rangle .$$

This requires $O(M)$ field operations, because it costs $O(N)$ to calculate all entries of the vectors $\otimes_j(1, \rho_j)$, v and $\otimes_j(1, \rho'_j)$, and $\mathbf{y} \circ \otimes_j(1, \rho_j)$, $O(M)$ entries to perform multiplications by A , B and C and then $O(N)$ to evaluate the scalar product. Note that \mathcal{V} can evaluate $\mathbf{z}(X)$ at any point γ with $O(|x|)$ operations by querying \mathbf{w} at γ and computing the expression $\mathbf{x}(X) + X^{|\mathbf{x}|} \mathbf{w}(X)$ by using \mathbf{x} . \square

Lemma 4.7.5. *The verifier in Construction 6 can be implemented with arithmetic complexity $O(M \log N)$, $O(\log N)$ memory, and $O(1)$ passes over the streams $\mathcal{S}_{\text{cmaj}}(U)$ for $U \in \{A, B, C\}$.*

Proof sketch. As in Lemma 4.7.4, the main cost for the verifier in this protocol is the cost of checking the claim

$$\begin{aligned} u_E &= \langle \mathbf{a}^* + \eta \cdot \mathbf{b}^* + \eta^2 \mathbf{c}^*, \otimes_j(1, \rho'_j) \rangle \\ &= \langle (\mathbf{y}^\top \circ \otimes_j(1, \rho_j)^\top) \cdot A, \otimes_j(1, \rho'_j) \rangle + \eta \cdot \langle \otimes_j(1, \rho_j)^\top B, \otimes_j(1, \rho'_j) \rangle + \eta^2 \cdot \langle \mathbf{y}^\top C, \otimes_j(1, \rho'_j) \rangle . \end{aligned}$$

The verifier can compute the value on the right hand side as follows. Consider the term $\langle \otimes_j(1, \rho_j)^\top B, \otimes_j(1, \rho'_j) \rangle = \sum_{k \in [M]} [\otimes_j(1, \rho_j)^\top]_{\text{row}_{B,k}} \cdot \text{val}_{B,k} \cdot [\otimes_j(1, \rho'_j)]_{\text{col}_{B,k}}$. To compute this term, the verifier sets a running total z to be equal to 0, and for each $k \in [M]$, streams the k -th element of row_B , col_B and val_B to get values i , j and v . The verifier computes the binary decomposition of i and j and uses them to compute the i -th and j -th entries of $\otimes_j(1, \rho_j)^\top$ and $\otimes_j(1, \rho'_j)^\top$, by performing at most $O(\log N)$ multiplications. The verifier then multiplies these values together, multiplies by v , and adds the result to the running total z . Repeating this process for the terms containing A and C , it is easy to see that the verifier uses $O(M \log N)$ operations and $O(\log N)$ space. \square

4.8 Achieving holography

We extend the *non-holographic* elastic PIOP of the previous section to additionally achieve holography.

Theorem 4.8.1. *For every finite field \mathbb{F} and positive integers N, M , there is a holographic PIOP for the indexed relation $\mathcal{R}_{\text{R1CS}}$ for instances with $N \times N$ matrices with M non-zero entries and public input \mathbf{x} , with the following complexity parameters:*

time-efficient prover	space-efficient prover	verifier time	soundness error	round complexity	message complexity	query complexity
$O(M)$ \mathbb{F} -ops	$O(M \log^2 N)$ \mathbb{F} -ops					
$O(M)$ memory	$O(\log M)$ memory $O(\log M)$ passes	$O(\mathbf{x} + \log M)$ \mathbb{F} -ops	$O\left(\frac{M}{ \mathbb{F} }\right)$	$O(\log M)$	$O(\log M)$	$O(\log M)$

4.8.1 Proof of Theorem 4.8.1

Part of Construction 6 (the non-holographic R1CS protocol) generates the claims that

$$\begin{aligned} u_A &= \langle \mathbf{z}, \mathbf{y}^\top \circ \otimes_j (1, \rho_j)^\top \cdot A \rangle, \\ u_B &= \langle \mathbf{z}, \otimes_j (1, \rho_j)^\top B \rangle, \\ u_C &= \langle \mathbf{z}, \mathbf{y}^\top C \rangle, \end{aligned}$$

and then checks these claims using a scalar-product protocol. The holographic protocol checks the claims using the alternative construction following the strategy in [BCG20]. The key subprotocols are a look-up protocol in Section 4.8.2 and an entry-product protocol in Section 4.8.3. As discussed in Section 4.2, we leverage plookup in Section 4.8.2.1 and offline-memory checking in Section 4.8.2.2 to build the look-up protocol.

Remark 4.8.2. In the construction, we will assume that the matrices A, B and C have the same support, which means that $\text{row} := \text{row}_A = \text{row}_B = \text{row}_C$ and $\text{col} := \text{col}_A = \text{col}_B = \text{col}_C$. This can be achieved by padding $\text{val}_A, \text{val}_B$ and val_C with zeroes as required, and increases the length of the sparse representations of A, B and C by at most a factor of 3.

Construction 7. *We construct a PIOP for the indexed relation $\mathcal{R}_{\text{R1CS}}$. The indexer algorithm takes as input an index $(\mathbb{F}, N, M, A, B, C)$, outputs the oracle messages val_U for $U \in \{A, B, C\}$, and runs the indexer algorithm of Construction 8 on row and col .⁹ The prover \mathcal{P} takes as input the index $\mathfrak{i} = (\mathbb{F}, N, M, A, B, C)$, instance $\mathfrak{x} = \mathbf{x}$, and witness $\mathfrak{w} = \mathbf{w}$; the verifier \mathcal{V} takes as input the instance \mathfrak{x} and has query access to the indexer's oracle messages. The protocol proceeds as follows.*

- The protocol begins by running the first two steps of Construction 6.

⁹If the lookup protocol is implemented using the memory-checking protocol, the indexer runs the indexer of Construction 9 and Construction 8.

- The prover \mathcal{P} constructs the vectors

$$\begin{aligned} \mathbf{z}^* &:= \mathbf{z}|_{\text{col}} , & \mathbf{r}_A^* &:= (\mathbf{y} \circ \otimes_j(1, \rho_j))|_{\text{row}} , \\ & & \mathbf{r}_B^* &:= (\otimes_j(1, \rho_j))|_{\text{row}} , \\ & & \mathbf{r}_C^* &:= \mathbf{y}|_{\text{row}} . \end{aligned}$$

It sends $\mathbf{z}^*, \mathbf{r}_A^*, \mathbf{r}_B^*, \mathbf{r}_C^* \in \mathbb{F}^M$ as oracle messages to the verifier.

- The prover \mathcal{P} and verifier \mathcal{V} engage in the following scalar product subprotocols in parallel using the same verifier randomness for each subprotocol:

- one with statement $\mathbb{x} = (\mathbb{F}, M, u_A)$ and witness $\mathbb{w} = (\mathbf{r}_A^*, \text{val}_A, \mathbf{z}^*)$;
- one with statement $\mathbb{x} = (\mathbb{F}, M, u_B)$ and witness $\mathbb{w} = (\mathbf{r}_B^*, \text{val}_B, \mathbf{z}^*)$; and
- one with statement $\mathbb{x} = (\mathbb{F}, M, u_C)$ and witness $\mathbb{w} = (\mathbf{r}_C^*, \text{val}_C, \mathbf{z}^*)$.

- Let $\mathbf{r}_B := \otimes_j(1, \rho_j)$, $\mathbf{r}_C := \mathbf{y}$ and $\mathbf{r}_A := \mathbf{y} \circ \otimes_j(1, \rho_j)$. The prover \mathcal{P} and the verifier \mathcal{V} engage (in parallel) the following subprotocols:

- $\mathbf{z}^* \subseteq \mathbf{z}$: invoke lookup with index $\mathfrak{i} = \text{col}$ with statement $\mathbb{x} = (\mathbb{F}, M, N)$ and witness $\mathbb{w} = (\mathbf{z}^*, \mathbf{z})$
- $\mathbf{r}_A^* = \mathbf{r}_B^* \circ \mathbf{r}_C^*$: invoke Hadamard protocol protocol with statement $\mathbb{x} = (\mathbb{F}, M)$ and witness $\mathbb{w} = (\mathbf{r}_B^*, \mathbf{r}_C^*, \mathbf{r}_A^*)$
- $\mathbf{r}_B^* \subseteq \mathbf{r}_B$: invoke lookup with index $\mathfrak{i} = \text{row}$ with statement $\mathbb{x} = (\mathbb{F}, M, N)$ and witness $\mathbb{w} = (\mathbf{r}_B^*, \mathbf{r}_B)$
- $\mathbf{r}_C^* \subseteq \mathbf{r}_C$: invoke lookup with index $\mathfrak{i} = \text{row}$, statement $\mathbb{x} = (\mathbb{F}, M, N)$ and witness $\mathbb{w} = (\mathbf{r}_C^*, \mathbf{r}_B)$

4.8.2 Lookup protocol

Definition 4.8.3. The lookup relation \mathcal{R}_{LU} is the set of tuples $(\mathfrak{i}, \mathbb{x}, \mathbb{w}) = (\text{addr}, (\mathbb{F}, M, N), (\mathbf{f}^*, \mathbf{f}))$ where $\mathbf{f}^* \in \mathbb{F}^M$, $\mathbf{f} \in \mathbb{F}^N$, $\text{addr} \in [N]^M$ such that $\mathbf{f}_i^* = \mathbf{f}_{\text{addr}_i}$.

We can transform the relations in Construction 7 into the above lookup relations. For example, for the relation $\mathbf{z}^* \subseteq \mathbf{z}$, we consider representing \mathbf{z}^* as \mathbf{f}^* , \mathbf{z} as \mathbf{f} , and col as addr . It is non-trivial to construct this protocol for this relation in the streaming model. At first glance, it would appear that random access to the vector \mathbf{f} is necessary, since \mathbf{f}^* is indexed both by $i \in [N]$ and by addr_i . To overcome these issues, we require the stream of the index addr to be implemented as two different streaming oracles.

- $\mathcal{S}(\text{addr})$, the stream of the vector addr ;
- $\mathcal{S}_{\text{sort}}(\text{addr})$, the stream of the vector addr , sorted in decreasing order.

Stream $\mathcal{S}(\mathbf{z}^*) = \mathcal{S}_{\text{lu}}(\mathbf{z}, U)$	Stream $\mathcal{S}_{\text{tensor-lu}}(\mathbf{r}_U^*) = \mathcal{S}(\boldsymbol{\rho}, U)$
Initialize: $a := \perp$ / holds z_i $i := N$ Next message: / Get col_i $(_, c, _) := \mathcal{S}_{\text{cmaj}}(U).\text{next}()$ / Fast-forward $\mathcal{S}(\mathbf{z})$ to i while ($i > c$) Decrement i $a := \mathcal{S}(\mathbf{z}).\text{next}()$ / Since col is ordered, now $i = c$ yield a	Next message: / Get row_i $(r, _, _) := \mathcal{S}_{\text{cmaj}}(U).\text{next}()$ / Let b_k be the k -th bit of r / t could be computed more efficiently / in specific RICS instances $t := \prod_k \rho_k^{b_k}$ yield t

Figure 4.5: Stream of the vectors \mathbf{z}^* , \mathbf{r}_A^* , \mathbf{r}_B^* , \mathbf{r}_C^* . $\mathcal{S}_{\text{cmaj}}(U)$ produces the sparse representation triplets (row, col, value) in column-major; we use pattern-matching with “_” to assign the value we are interested in.

The previous literature studying PIOPs for \mathcal{R}_{LU} essentially follows two different approaches: the plookup protocol and the offline-memory checking protocol. Only the plookup protocol is compatible with our elastic model. In the following sections, we discuss both of these two lookup protocols and their limitations.

Theorem 4.8.4. *For every finite field \mathbb{F} and positive integer N , there is a holographic PIOP for the indexed relation \mathcal{R}_{LU} with:*

time-efficient prover	space-efficient prover	verifier time	soundness error	round complexity	message complexity	query complexity
$O(M)$ \mathbb{F} -ops	$O(M \log M)$ \mathbb{F} -ops	$O(\log M)$ \mathbb{F} -ops	$O\left(\frac{M}{ \mathbb{F} }\right)$	$O(\log M)$	$O(\log M)$	$O(\log M)$
$O(M)$ memory	$O(\log M)$ memory $O(\log M)$ passes					

Remark 4.8.5. In the lookup protocol, the verifier needs to query the polynomial $\mathbf{f}(x) = \sum_{i=0}^d i \cdot x^i$ at a random point α . This can be computed in time $O(\log d)$ with $\mathbf{f}(\alpha) = \frac{\alpha(1-\alpha^d)}{(1-\alpha)^2} - \frac{d\alpha^{d+1}}{1-\alpha}$, by differentiating the geometric series formula. If $\alpha = 1$, we obtain the well-known formula $\mathbf{f}(1) = d(d+1)/2$.

4.8.2.1 Plookup

This technique was previously employed in [GW20; BCG20]. It expresses the lookup as a polynomial relation of this form:

Lemma 4.8.6 ([GW20, Claim 3.1]). *Let $\mathbf{f}^* \in \mathbb{F}^M$ and $\mathbf{f} \in \mathbb{F}^N$. Then $\mathbf{f}^* \subseteq \mathbf{f}$ if and only if there exists $\mathbf{w} \in \mathbb{F}^{M+N}$ such that the equation below in $\mathbb{F}[Y, Z]$ is satisfied:*

$$\prod_{j=0}^{M+N-1} \left(Y(1+Z) + w_{j+1} + w_j \cdot Z \right) = (1+Z)^M \prod_{j=0}^{M-1} (Y + f_j) \prod_{j=0}^{N-1} \left(Y(1+Z) + f_{j+1} + f_j \cdot Z \right) \quad (4.25)$$

where indices are taken (respectively) modulo $M + N$, N . If $\mathbf{f}^* \subseteq \mathbf{f}$, then $\mathbf{w} := \text{sort}(\mathbf{f}^*, \mathbf{f})$ satisfies Equation (4.25).

In the above equation, we consider the subset relation $\mathbf{a} \subseteq \mathbf{d}$, and use \mathbf{a}_\circ to denote the rotation of the vector \mathbf{a} . Therefore, it is sufficient to test the above polynomial equality over two random points in the field.

As the above equation will prove a simpler subset statement that $\mathbf{a} \subseteq \mathbf{d}$, we will *hash* the values with their indices to prove the lookup relation \mathcal{R}_{LU} : the verifier sends $\eta \in \mathbb{F}$ sampled uniformly at random and then, the prover defines:

$$\begin{aligned} \mathbf{a} &:= \mathbf{z}^* + \eta \cdot \text{col} \\ \mathbf{d} &:= \mathbf{z} + \eta \cdot [N] \end{aligned} \quad (4.26)$$

We proceed similarly for \mathbf{r}_U^* , and \mathbf{r}_U . We note that there is no need to send these oracles to the verifier: the verifier simply needs to substitute Equation (4.26) and ask for an evaluation in the vectors composing it.

Remark 4.8.7. Note that the verifier does not have the oracle access to the shift vectors \mathbf{d}_\circ and \mathbf{w}_\circ . To avoid having the prover send the shifted oracles and having the verifier check consistency between them, the prover can add a leading zero to \mathbf{d} and \mathbf{w} . The plookup relation Equation (4.25) still holds as long as there is no zero entry in the set, which is guaranteed by the algebraic hash. As a result, the verifier can obtain the evaluation of the shifted oracles as follows:

$$\mathbf{d}_\circ(x) = x^N \mathbf{d}(x) \quad \mathbf{w}_\circ(x) = x^{N+M} \mathbf{w}(x) . \quad (4.27)$$

Construction 8. *We construct a PIOP for the indexed relation \mathcal{R}_{LU} . Given $\mathfrak{i} = \text{addr}$ as input, the indexer algorithm outputs addr as an oracle message. The prover \mathcal{P} takes as input an index $\mathfrak{i} = \text{addr}$, instance $\mathfrak{x} = (\mathbb{F}, M, N)$, and witness $\mathfrak{w} = (\mathbf{f}^*, \mathbf{f})$; the verifier \mathcal{V} has query access to the index \mathfrak{i} and the witness \mathfrak{w} and takes as input the instance \mathfrak{x} . The protocol proceeds as follows.*

- The prover \mathcal{P} constructs and sends the oracle message \mathfrak{w} to the verifier \mathcal{V} .
- The verifier \mathcal{V} samples random elements $v, \zeta \leftarrow \mathbb{F}^\times$ and sends them to \mathcal{P} .

- The prover \mathcal{P} computes:

$$e_a := \prod_{j=0}^{M-1} (a_j + \zeta)$$

$$e_d := \prod_{j=0}^{N-1} ((1 + \zeta)v + d_{\circ j} + \zeta d_j)$$

$$e_w := \prod_{j=0}^{M+N-1} ((1 + \zeta)v + w_{\circ j} + \zeta w_j)$$

and sends them to the verifier.

- The verifier \mathcal{V} checks that $(1 + v)^M e_a e_d = e_w$
- The prover \mathcal{P} and the verifier \mathcal{V} engage (in parallel) the entry-product subprotocol with the following statements and witnesses:
 - entry product with statement $\mathfrak{x} = (\mathbb{F}, M, e_a)$ and witness $\mathfrak{w} = (\mathbf{a} + v)$;
 - entry product with statement $\mathfrak{x} = (\mathbb{F}, N, e_d)$ and witness $\mathfrak{w} = ((1 + \zeta)v + \mathbf{d}_{\circ} + \zeta \mathbf{d})$;
 - entry product with statement $\mathfrak{x} = (\mathbb{F}, N + M, e_w)$ and witness $\mathfrak{w} = ((1 + \zeta)v + \mathbf{w}_{\circ} + \zeta \mathbf{w})$.

In order to achieve space-efficiency, we design the algorithm in Figure 4.6 to realize the stream of the vectors in Construction 8. For example, we can use $\mathcal{S}_{\text{set}}(\mathcal{S}_{\text{merge}}(\mathbf{z}_U^*, \mathbf{z}))$ to construct the stream of the sorted vector \mathbf{w} .

4.8.2.2 Memory checking

The offline memory-checking technique originated in [BEGKN91] and was used in prior argument systems [SATJ18; Set20]. It can be used to prove claims of the form $\mathbf{f}^* \subset \mathbf{f}$, but and unlike the plookup protocol some oracles can be computed by the indexer. The indexer, for the specific relation \mathcal{R}_{LU} , computes three oracles from the index addr : the read timestamp read-ts , the write timestamp write-ts , and the audit timestamp audit-ts as follows:

$$\text{read-ts} := (\max j \in [M] : \text{addr}_j = \text{addr}_i)_{i \in \text{addr}} , \quad (4.28)$$

$$\text{write-ts} := [N] , \quad (4.29)$$

$$\text{audit-ts} := (\max j < i : \text{addr}_j = \text{addr}_i)_{i \in \text{addr}} . \quad (4.30)$$

If, at the i -th position, no such maximum exists, the element is set to zero. At the core of the memory-checking protocol there is the following polynomial relation.

Stream $\mathcal{S}_{\text{merge}}(\mathcal{S}_1, \mathcal{S}_2)$	Stream $\mathcal{S}_{\text{subset}}(\mathcal{S}_1, \zeta)$	Stream $\mathcal{S}_{\text{set}}(\mathcal{S}_1, v, \zeta)$
<p>Initialize:</p> <p style="padding-left: 20px;">$a := \mathcal{S}_1.\text{next}()$</p> <p style="padding-left: 20px;">$b := \mathcal{S}_2.\text{next}()$</p> <p>Next message:</p> <p style="padding-left: 20px;">if $(a \neq b)$</p> <p style="padding-left: 40px;">$a := \mathcal{S}_1.\text{next}()$</p> <p style="padding-left: 20px;">else</p> <p style="padding-left: 40px;">$b := \mathcal{S}_2.\text{next}()$</p> <p style="padding-left: 20px;">yield a</p>	<p>Next message:</p> <p style="padding-left: 20px;">$a := \mathcal{S}_1.\text{next}()$</p> <p style="padding-left: 20px;">yield $a + \zeta$</p>	<p>Initialize:</p> <p style="padding-left: 20px;">$a := 0$</p> <p>Next message:</p> <p style="padding-left: 20px;">$b := a$</p> <p style="padding-left: 20px;">$a := \mathcal{S}_1.\text{next}()$</p> <p style="padding-left: 20px;">if $a \neq \perp$</p> <p style="padding-left: 40px;">yield $v(1 + \zeta) + a + \zeta \cdot b$</p> <p style="padding-left: 20px;">else</p> <p style="padding-left: 40px;">yield $v(1 + \zeta) + \zeta \cdot b$</p>

Figure 4.6: Stream of the vectors for the lookup protocol.

Theorem 4.8.8. Let $\mathbf{f}^* \in \mathbb{F}^M$, $\mathbf{f} \in \mathbb{F}^N$, and $\text{addr} \in [N]^M$. We have $\mathbf{f}_i^* = \mathbf{f}_{\text{addr}_i}$ for all $i \in [M]$ if and only if

$$\begin{aligned}
& \prod_{j=1}^N (X - H_Y(j, \mathbf{f}_j)) \cdot \prod_{j=1}^M (X - H_Y(\text{addr}_j, \mathbf{f}_j^*, \text{write-ts}_j)) \\
& = \prod_{j=1}^M (X - H_Y(\text{addr}_j, \mathbf{f}_j^*, \text{read-ts}_j)) \cdot \prod_{j=1}^N (X - H_Y(j, \mathbf{f}_j, \text{audit-ts}_j))
\end{aligned} \tag{4.31}$$

where $H_Y(a, b, c) = a + Y \cdot b + Y^2 \cdot c$, and read-ts , write-ts and audit-ts are vectors produced by the offline memory-checking procedure described in [Set20].

The prior work [Set20] expresses Equation (4.31) in the circuit, and leverages an external proof system to generate the proof. In contrast, we reduce the Equation (4.31) into four entry product arguments. The verifier samples random challenges $\beta, \sigma \in \mathbb{F}^\times$, and both parties engage in the entry product protocols for $X = \beta$ and $Y = \sigma$. As discussed in Section 4.2.7.2, the offline-memory checking is only compatible with the lookup claim $(\mathbf{z}^*, \text{col}) \subseteq (\mathbf{z}, [N])$.

The timestamp oracles read-ts , write-ts , and audit-ts can be computed in linear-time by the indexer, as shown in Fig. 4.7. Then if the underlying entry product protocol is linear-time, the overall offline memory-checking protocol is also linear-time. We now consider the space-efficient realization. The biggest challenge is to produce the streams of timestamp oracles read-ts , write-ts , and audit-ts . However, we observe that because we organize the non-zero entries in the column-major order, the elements in the stream of col_U are in non-descending order. As a result, the streams of read-ts , write-ts , and audit-ts can be computed by a single pass of col_U .

Indexer for Construction 9

```

read-ts := (0)i∈[N]
write-ts := (0)i∈[N]
audit-ts := (0)i∈[N]
for (i, a) ∈ [N] × addr
  write-ts[i] := i
  read-ts[i] := audit-ts[a]
  audit-ts[a] := i

```

Figure 4.7: Linear-time algorithm for generating the timestamp vectors in Construction 9 [BEGKN91; Set20].

Construction 9. We construct a holographic PIOP for the indexed relation \mathcal{R}_{LU} . On input addr , the indexer produces the oracle messages read-ts , audit-ts . The prover \mathcal{P} takes as input an index $\mathbb{i} = \text{addr}$, instance $\mathbb{x} = (\mathbb{F}, M, N)$, and witness $\mathbb{w} = (\mathbf{f}^*, \mathbf{f})$; the verifier \mathcal{V} takes as input the index \mathbb{i} and the instance \mathbb{x} . The prover \mathcal{P} and the verifier \mathcal{V} proceed as follows:

- The verifier \mathcal{V} samples random elements $\beta, \sigma \leftarrow \mathbb{F}^\times$ and sends them to \mathcal{P} .
- The prover \mathcal{P} computes:

$$\begin{aligned}
e_{\text{init}} &= \prod_{j=1}^N (\beta - H_\sigma(j, f_j)), \\
e_{\text{ws}} &= \prod_{j=1}^M (\beta - H_\sigma(\text{addr}_j, f_j^*, \text{write-ts}_j)), \\
e_{\text{rs}} &= \prod_{j=1}^M (\beta - H_\sigma(\text{addr}_j, f_j^*, \text{read-ts}_j)), \\
e_{\text{as}} &= \prod_{j=1}^N (\beta - H_\sigma(j, f_j, \text{audit-ts}_j)) .
\end{aligned}$$

and sends them to the verifier.

- The verifier \mathcal{V} checks that $e_{\text{init}} \cdot e_{\text{ws}} = e_{\text{rs}} \cdot e_{\text{as}}$
- Let $\text{init}, \text{as} \in \mathbb{F}^N$ and $\text{ws}, \text{rs} \in \mathbb{F}^M$ denote the vectors involved in each entry product. The prover \mathcal{P} and the verifier \mathcal{V} engage entry-product subprotocols with the following statements and witnesses:

- entry product with statement $\mathbb{x} = (\mathbb{F}, M, e_{\text{init}})$ and witness $\mathbb{w} = \text{init}$;
- entry product with statement $\mathbb{x} = (\mathbb{F}, M, e_{\text{ws}})$ and witness $\mathbb{w} = \text{ws}$;
- entry product with statement $\mathbb{x} = (\mathbb{F}, M, e_{\text{rs}})$ and witness $\mathbb{w} = \text{rs}$;
- entry product with statement $\mathbb{x} = (\mathbb{F}, N, e_{\text{as}})$ and witness $\mathbb{w} = \text{as}$.

An oracle query to e.g. **init** in a subprotocol is made by taking the appropriate linear combination of queries to $(0, \dots, N-1)$, **f** and 0^N .

4.8.3 Entry product

Let **f** be a monic polynomial whose product of coefficients is e . We design a proof system for proving that $e = \prod_i f_i$. More formally:

Definition 4.8.9. The **entry product relation** \mathcal{R}_{EP} is the set of tuples $(\mathfrak{i}, \mathbb{x}, \mathbb{w}) = (\perp, (\mathbb{F}, N, e), \mathbf{f})$ where $\mathbf{f} \in \mathbb{F}^N$ and $\prod_{i=0}^{N-1} f_i = e$.

Theorem 4.8.10. For every finite field \mathbb{F} and positive integer N , there is a PIOP for the indexed relation \mathcal{R}_{EP} with:

time-efficient prover	space-efficient prover	verifier time	soundness error	round complexity	message complexity	query complexity
$O(N)$ \mathbb{F} -ops	$O(N \log N)$ \mathbb{F} -ops	$O(\log N)$ \mathbb{F} -ops	$O\left(\frac{N}{ \mathbb{F} }\right)$	$O(\log N)$	$O(\log N)$	$O(\log N)$
$O(N)$ memory	$O(\log N)$ memory $O(\log N)$ passes					

We can evaluate Equation (4.32) at a random point α and check it using a scalar-product protocol. This requires one scalar-product protocol and two evaluation queries.

Construction 10. We construct a PIOP for the indexed relation \mathcal{R}_{EP} . The prover \mathcal{P} takes as input an index $\mathfrak{i} = \perp$, instance $\mathbb{x} = (\mathbb{F}, N, e)$, and witness $\mathbb{w} = \mathbf{f}$; the verifier \mathcal{V} takes as input the index \mathfrak{i} and the instance \mathbb{x} . The prover \mathcal{P} and the verifier \mathcal{V} proceed as follows:

- The prover \mathcal{P} constructs $\mathbf{g} := (\prod_{i \geq 0} f_i, \prod_{i \geq 1} f_i, \dots, f_{N-2}f_{N-1}, f_{N-1})$, and sends the field element f_{N-1} and the oracle message \mathbf{g} to the verifier.
- The verifier samples a random $\alpha \leftarrow \mathbb{F}^\times$ and sends it to the prover.
- The verifier queries the oracle for \mathbf{g} to learn $\mathbf{g}(\alpha)$, and the prover and verifier compute $u := \mathbf{g}(Y)Y + f_{N-1}(e - Y^N)$.

The prover \mathcal{P} and verifier \mathcal{V} engage in a twisted scalar-product protocol (Construction 3) with index $\mathfrak{i} = \perp$, instance $\mathbb{x} = (\mathbb{F}, N, \mathbf{y}, u)$, and witness $\mathbb{w} = (\mathbf{g} \circ \mathbf{y}, \mathbf{f}_\circ)$ and produces claims that $A = \langle \mathbf{g}, \otimes_j(1, v_j \rho_j) \rangle$ and $C = \langle \mathbf{f}_\circ, \otimes_j(1, \rho_j) \rangle$. where $\mathbf{y} = (\alpha^0, \dots, \alpha^{N-1}) = \otimes_j(1, v_j)$ and $(\rho_0, \dots, \rho_{n-1})$ are the verifier randomnesses in the scalar product protocol. They check these claims using the tensor product protocol (Construction 1).

Lemma 4.8.11. *Let $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$, with $f_{N-1} \neq 0$. Let $e \in \mathbb{F}$. Then*

$$\langle \mathbf{g} \circ \mathbf{Y}, \mathbf{f}_{\circ} \rangle = \mathbf{g}(Y)Y + f_{N-1}(e - Y^N) \quad (4.32)$$

if and only if

$$\mathbf{g} := (\prod_{i \geq 0} f_i, \prod_{i \geq 1} f_i, \dots, f_{N-2}f_{N-1}, f_{N-1}) .$$

and $\prod_{i \geq 0} f_i = e$.

Proof. If $\mathbf{f} = (f_0, f_1, f_2, \dots, f_{N-1})$, then $\mathbf{f}_{\circ} = (f_{N-1}, f_0, f_1, \dots, f_{N-2})$, so

$$\langle \mathbf{g} \circ \mathbf{Y}, \mathbf{f}_{\circ} \rangle = g_0 f_{N-1} + g_1 f_0 Y + \dots + g_{N-2} f_{N-3} Y^{N-2} + g_{N-1} f_{N-2} Y^{N-1} . \quad (4.33)$$

We also have

$$\mathbf{g}(Y)Y + f_{N-1}(e - Y^N) = \sum_{i=1}^{N-1} g_{i-1} Y^i + f_{N-1}e + g_{N-1} Y^N - f_{N-1} Y^N . \quad (4.34)$$

Comparing coefficients shows that the polynomials in Equation (4.33) and Equation (4.34) are equal if and only if $g_i = g_{i+1}f_i$ for $i \in [N-1]$, with $g_{N-1} = f_{N-1}$ and $g_0 f_{N-1} = e f_{N-1}$. Since $f_{N-1} \neq 0$, the claim follows. \square

Remark 4.8.12. Construction 10 requires \mathbf{f} to be a monic polynomial, however, the polynomial \mathbf{f}' in the lookup protocol does not satisfy this condition. To solve that, the prover and the verifier run the entry product for $\mathbf{f}(x) = \mathbf{f}'(x) + x^n$. Note that the claim of the entry product remains the same. Moreover, the verifier can easily query the shifted oracle as follows: $\mathbf{f}_{\circ}(x) = 1 + x\mathbf{f}'(x)$.

4.9 Polynomial commitment schemes

We use polynomial commitment schemes to compile our PIOPs into cryptographic arguments. We require elastic polynomial commitment schemes to compile elastic PIOPs into elastic argument systems. We will use the same polynomial commitment scheme as [CHMMVW20], which is a variant of the construction in [KZG10]. Our contribution is to show that this scheme has elastic commitment and opening algorithms.

Theorem 4.9.1. *The commitment scheme of [CHMMVW20] is elastic, with:*

setup time	time-efficient commitment	time-efficient opening	space-efficient commitment	space-efficient opening	check time	commitment and opening sizes
MSM(D) ops	MSM(D) ops $O(D)$ memory	MSM(D) ops $O(D)$ memory	$O(D)$ SM ops $O(1)$ SM memory $O(1)$ passes	$O(D)$ SM ops $O(1)$ SM memory $O(1)$ passes	$O(1)$ SM + $O(1)$ PA	$O(1)$ GE

In the above theorem, we let MSM denote multi-scalar exponentiation, SM denotes exponentiation, PA denotes pairing, GE denotes a group element, and D denotes the maximum degree of the polynomials. In the rest of this section, we give formal definitions for an elastic polynomial commitment scheme and show how to construct it.

4.9.1 Definition

A polynomial commitment scheme over a field family \mathcal{F} is a tuple of algorithms $\text{PC} = (\text{Setup}, \text{Com}, \text{Open}, \text{Check})$ with the following syntax.

- **Setup** $\text{PC.Setup}(1^\lambda, D) \rightarrow (\text{ck}, \text{rk})$. On input a security parameter λ (in unary), and a maximum degree bound $D \in \mathbb{N}$, PC.Setup samples a commitment key ck and verification key rk , which contain the description of a finite field $\mathbb{F} \in \mathcal{F}$.
- **Commit** $\text{PC.Com}(\text{ck}, \mathbf{p}) \rightarrow C$. On input ck and a univariate polynomial \mathbf{p} of degree at most D over the field \mathbb{F} , PC.Com outputs commitment C to the polynomial \mathbf{p} .
- **Open** $\text{PC.Open}(\text{ck}, \mathbf{p}, z) \rightarrow \pi$. On input ck , a univariate polynomial \mathbf{p} , degree bounds D , and a query point $z \in \mathbb{F}$, PC.Open outputs an evaluation proof π .
- **Check** $\text{PC.Check}(\text{rk}, C, z, v, \pi) \in \{0, 1\}$. On input rk , the commitment C , query point $z \in \mathbb{F}$, alleged evaluation v , and an evaluation proof π , PC.Check outputs 1 if π attests that the polynomial \mathbf{p} committed in C has degree at most D and evaluates to v at z .

A polynomial commitment scheme PC must satisfy *completeness* and *extractability*.

Definition 4.9.2 (Completeness). For every degree bound $D \in \mathbb{N}$ and efficient adversary \mathcal{A} ,

$$\Pr \left[\begin{array}{c} \deg(\mathbf{p}) \leq D \\ \Downarrow \\ \text{PC.Check}(\text{rk}, C, z, v, \pi, \xi) = 1 \end{array} \middle| \begin{array}{l} (\text{ck}, \text{rk}) \leftarrow \text{PC.Setup}(1^\lambda, D) \\ (\mathbf{p}, z) \leftarrow \mathcal{A}(\text{ck}, \text{rk}) \\ C \leftarrow \text{PC.Com}(\text{ck}, \mathbf{p}) \\ v = \mathbf{p}(z) \\ \pi \leftarrow \text{PC.Open}(\text{ck}, \mathbf{p}, z) \end{array} \right] = 1 .$$

Definition 4.9.3 (Extractability). For every degree bound $D \in \mathbb{N}$ and efficient adversary \mathcal{A} there exists an efficient extractor \mathcal{E} such that for every round bound $r \in \mathbb{N}$, efficient query sampler \mathcal{Q} , and efficient adversary \mathcal{B} the probability below is negligibly close to 1 (as a function of λ):

$$\Pr \left[\begin{array}{c} \text{PC.Check}(\text{rk}, C, z, v, \pi) = 1 \\ \Downarrow \\ \deg(\mathbf{p}) \leq D \text{ and } v = \mathbf{p}(z) \end{array} \middle| \begin{array}{l} (\text{ck}, \text{rk}) \leftarrow \text{PC.Setup}(1^\lambda, D) \\ C \leftarrow \mathcal{A}(\text{ck}, \text{rk}) \\ \mathbf{p} \leftarrow \mathcal{E}(\text{ck}, \text{rk}) \\ z \leftarrow \mathcal{Q}(\text{ck}, \text{rk}) \\ (\pi, v, \text{st}) \leftarrow \mathcal{B}(\text{ck}, \text{rk}, z) \end{array} \right] .$$

(The above definition captures the case where $\mathcal{A}, \mathcal{Q}, \mathcal{B}$ share the same random string to win the game.)

4.9.2 An elastic polynomial commitment scheme

The polynomial commitment scheme of [KZG10] consists of the following algorithms.

- **Setup** $\text{PC.Setup}(1^\lambda, D) \rightarrow \text{ck}$. First, PC.Setup samples a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, G, H, e) \leftarrow \text{SampleGrp}(1^\lambda)$. Next, PC.Setup samples $\beta \in \mathbb{F}$, computes βH and computes the vector

$$\Sigma = (G_D, G_{D-1}, \dots, G_1, G_0) := \left(\beta^D G \quad \beta^{D-1} G \quad \dots \quad \beta G \quad G \right) \in \mathbb{G}_1^{D+1} .$$

Finally, PC.Setup outputs $\text{ck} := ((\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, G, H, e), \Sigma)$ and $\text{rk} := (G, H, \beta H)$.

- **Commit** $\text{PC.Com}(\text{ck}, \mathbf{p}) \rightarrow C$. The commitment algorithm PC.Com parses $\mathbf{p}(X) \in \mathbb{F}[X]$ of degree $d \leq D$ as $\sum_{j=0}^d p_j X^j$ and outputs $C := \sum_{j=0}^d p_j \cdot G_j \in \mathbb{G}_1$.
- **Open** $\text{PC.Open}(\text{ck}, \mathbf{p}, z) \rightarrow \pi$. The opening algorithm PC.Open computes witness polynomial $\mathbf{w}(X) := \frac{\mathbf{p}(X) - \mathbf{p}(z)}{X - z}$, parses $\mathbf{w}(X)$ as $\sum_{j=0}^{d-1} w_j X^j$ and outputs the evaluation proof $\pi := \sum_{j=0}^{d-1} w_j G_j \in \mathbb{G}_1$.
- **Check** $\text{PC.Check}(\text{rk}, C, z, v, \pi) \in \{0, 1\}$. The check algorithm PC.Check outputs 1 if and only if $e(C - vG, H) = e(\pi, \beta H - zH)$.

Below we explain how to implement this scheme in small space using streaming algorithms. We only need to consider the Com and Open algorithms, since the Check algorithm is succinct and the setup algorithm has natural time-and-space-efficient realizations.

The stream $\mathcal{S}(\Sigma)$ of the vector Σ is simply the sequence (G_D, \dots, G_0) . The stream $\mathcal{S}(\mathbf{p})$ associated with the polynomial $\mathbf{p} = \sum_{j=0}^D p_j X^j$ is the sequence (p_d, \dots, p_0) . Note that it is important that $\mathcal{S}(\Sigma)$ and $\mathcal{S}(\mathbf{p})$ are ordered from highest powers of β and X to lowest in order to support an efficient PC.Open algorithm.

Commit. On input $((\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, G, H, e), \mathcal{S}(\Sigma), \mathcal{S}(\mathbf{p}))$, $\text{PC}_s.\text{Com}$ sets $C = 0 \in \mathbb{G}_1$. Then, for $j = d, d-1, \dots, 0$, $\text{PC}_s.\text{Com}$ uses the streaming oracles to get $p_j = \mathcal{S}(\mathbf{p}).\text{next}()$ and $G_j = \mathcal{S}(\Sigma).\text{next}()$ and computes $C := C + p_j G_j$. Otherwise, $\text{PC}_s.\text{Com}$ outputs $C \in \mathbb{G}_1$.

Note that $\text{PC}_s.\text{Com}$ only needs to store the value of C throughout the entire loop.

Open. On input $((\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, G, H, e), \mathcal{S}(\Sigma), \mathcal{S}(\mathbf{p}))$, evaluation point $z \in \mathbb{F}$, opening challenge $\xi \in \mathbb{F}$, $\text{PC}_s.\text{Open}$ sets $\pi = 0 \in \mathbb{G}_1$ and $w_d := 0 \in \mathbb{F}$, and consumes the first element of the stream $\mathcal{S}(\Sigma)$ by fetching $G_d = \mathcal{S}(\Sigma).\text{next}()$.

Then, for $j = d-1, \dots, 0$, $\text{PC}_s.\text{Open}$ uses the streaming oracles to get $p_{j+1} = \mathcal{S}(\mathbf{p}).\text{next}()$ and $G_j = \mathcal{S}(\Sigma).\text{next}()$, and computes $w_j := p_{j+1} + w_{j+1}z$ and $\pi := \pi + w_j G_j$.

Finally, $\text{PC}_s.\text{Com}$ outputs $\pi \in \mathbb{G}_1$.

Note that $\text{PC}_s.\text{Com}$ only needs to store the values of π, w_{j+1} for the next iteration of the loop.

Remark 4.9.4. The Open and Check algorithms (as well as the completeness and extractability properties) can be generalized as in [CHMMVW20; BDFG20] to allow *batched* opening and evaluation checking. Given polynomials $\mathbf{p}_0, \dots, \mathbf{p}_{m-1}$ and claimed evaluations v_0, \dots, v_{m-1} at a point z , the party verifying the commitment and openings selects a random opening challenge $\xi \in \mathbb{F}$, and the PC.Open and PC.Check algorithms are run on polynomial $\mathbf{p} := \sum_{i=0}^{m-1} \xi^i \mathbf{p}_i$ and claimed evaluation $v := \sum_{i=0}^{m-1} \xi^i v_i$. Given points z_0, \dots, z_ℓ and claimed evaluations v_0, \dots, v_ℓ , the evaluation proof defines $\mathbf{Z}(X) := \prod_i (X - z_i)$. Let $\mathbf{q}(X)$ and $\mathbf{r}(X)$ be respectively quotient and remainder of the Euclidian division between $\mathbf{p}(X)$ and $\mathbf{Z}(X)$. Let $R, Z \in \mathbb{G}$ be the group elements associated to $\mathbf{r}(X)$ and $\mathbf{Z}(X)$. The evaluation proof is $\pi := \langle \mathbf{q}(X), \text{ck} \rangle$ while the evaluations are obtained as $v_i := \mathbf{r}(z_i)$. The PC.Check algorithm reconstructs $\mathbf{r}(X)$ from the claimed evaluations via polynomial interpolation and checks that $e(C - R, H) = e(\pi, Z)$.

We briefly describe how the PC.Open algorithm can be implemented in small space. The stream $\mathcal{S}(\mathbf{p})$ can be computed by a streaming algorithm that uses $O(m\ell D)$ operations, $O(\ell)$ space (never storing more than $\mathbf{p}_i, \dots, \mathbf{p}_{i-\ell+1}$ coefficients for the partial computation of the opening), and a single pass over each of the m polynomials \mathbf{p}_i . Composing this streaming algorithm with the opening algorithm using Lemma 4.4.6 gives a batched opening algorithm.

The complexity of batched operations for PC is given in the following table.

time-efficient opening	space-efficient opening	check time	opening size
$O(D)$ SM + $O(m\ell D)$ \mathbb{F} ops	$O(D)$ SM + $O(m\ell D)$ \mathbb{F} ops	$O(m + \ell)$ SM + $O(1)$ PA	$O(1)$ GE
$O(mD + m\ell)$ memory	$O(\ell)$ SM memory $O(1)$ passes		

We note that, in Construction 1, $\ell = 3$ as we demand evaluations for $\beta, -\beta, \beta^2$ as displayed in Eq. (4.15).

4.10 Elastic argument systems

We describe a compiler that uses elastic polynomial commitment schemes and elastic PIOPs to construct elastic cryptographic arguments.

Theorem 4.10.1. *Consider the following.*

- A holographic PIOP over a field family \mathcal{F} , for an indexed relation \mathcal{R} , with:

indexer time	time-efficient prover	space-efficient prover	verifier time	soundness error	round complexity	message complexity	query complexity	communication complexity
$t_{\mathcal{I}}$	$t_{\mathcal{P}}$ time $s_{\mathcal{P}}$ memory	$t'_{\mathcal{P}}$ time $s'_{\mathcal{P}}$ memory $k_{\mathcal{P}}$ passes	$t_{\mathcal{V}}$ time	ϵ	k	l	q	cc

- a public-coin verifier and non-adaptive queries; and
- message schedule specified by o and d with output stream ordering $\{(a, b, c) : a \in [k], b \in [o(a)], c \in d(|i|, a, b)\}$ ordered first in ascending order by round number a , then ascending order by oracle number b , then descending order by polynomial coefficient c .
- maximum degree bound $D := \max_{a,b} \{d(|i|, a, b)\}$;

- A polynomial commitment scheme PC over a field family \mathcal{F} , with

setup time	time-efficient commitment	time-efficient opening	space-efficient commitment	space-efficient opening	check time	commitment and opening sizes
$t_{\text{PC.G}}$	$t_{\text{PC.Com}}$ time $s_{\text{PC.Com}}$ space	t_{O} time s_{O} memory	$t'_{\text{PC.Com}}$ time $s'_{\text{PC.Com}}$ memory $k_{\text{PC.Com}}$ passes	t'_{O} time s'_{O} memory po passes	$t_{\text{PC.Check}}$	$ \text{PC.Com} $

and input ordering $\{p_d, p_{d-1}, \dots, p_0\}$ for all streams of input polynomials $\mathbf{p}(X) = \sum_{i=0}^d p_i X^i$, for the PC.Com and PC.Open algorithms.

Then there is a preprocessing argument for \mathcal{R} with

generator time	indexer time	verifier time	soundness error	round complexity	communication complexity
$t_{\text{PC.G}}$	$t_{\mathcal{I}} + o(0) \cdot t_{\text{PC.Com}}$	$t_{\mathcal{V}} + q t_{\text{PC.Check}}$	$\epsilon + \text{negl}(\lambda)$	$k + 2$	$cc + \sum_{i=1}^k o(i) \cdot \text{PC.Com} + q \cdot \text{PC.Open} $

and prover complexity

time-efficient prover	space-efficient prover
$t_{\mathcal{P}} + \sum_{i=0}^{k-1} o(i) \cdot t_{\text{PC.Com}} + q(t_{\text{O}} + \text{te})$ time $s_{\mathcal{P}} + s_{\text{PC.Com}} + s_{\text{O}}$ memory	$(k_{\text{PC.Com}} + po)t'_{\mathcal{P}} + \sum_{i=0}^{k-1} o(i) \cdot t'_{\text{PC.Com}} + q \cdot (t'_{\text{O}} + \text{te})$ time $s'_{\mathcal{P}} + \sum_{i=0}^{k-1} o(i) \cdot s'_{\text{PC.Com}} + q \cdot s'_{\text{O}} + O(q)$ memory $(k_{\text{PC.Com}} + po) \cdot k_{\mathcal{P}}$ passes

In Section 4.10.1 we give formal definitions for preprocessing arguments with a universal structured reference string. In Section 4.10.2 we present a compiler and prove Theorem 4.10.1.

4.10.1 Preprocessing arguments with universal SRS

Following [CHMMVW20, Section 7], a preprocessing argument ARG with universal SRS for an indexed relation \mathcal{R} is a tuple of probabilistic polynomial-time algorithms $(\mathcal{G}_{\text{ARG}}, \mathcal{I}_{\text{ARG}}, \mathcal{P}_{\text{ARG}}, \mathcal{V}_{\text{ARG}})$ consisting of a generator \mathcal{G}_{ARG} , an indexer \mathcal{I}_{ARG} , a prover \mathcal{P}_{ARG} and a verifier \mathcal{V}_{ARG} such that the following properties hold.

- **Completeness.** For all size bounds $N \in \mathbb{N}$ and efficient \mathcal{A} ,

$$\Pr \left[\begin{array}{c} (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \notin \mathcal{R}_N \\ \vee \\ \langle \mathcal{P}_{\text{ARG}}(\text{ipk}, \mathfrak{x}, \mathfrak{w}), \mathcal{V}_{\text{ARG}}(\text{ivk}, \mathfrak{x}) \rangle = 1 \end{array} \mid \begin{array}{c} \text{srs} \leftarrow \mathcal{G}_{\text{ARG}}(1^\lambda, N) \\ (\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \leftarrow \mathcal{A}(\text{srs}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}_{\text{ARG}}^{\text{srs}}(\mathfrak{i}) \end{array} \right] = 1 .$$

- **Soundness.** For all size bounds $N \in \mathbb{N}$ and efficient $\tilde{\mathcal{P}} = (\tilde{\mathcal{P}}_1, \tilde{\mathcal{P}}_2)$,

$$\Pr \left[\begin{array}{c} (\mathfrak{i}, \mathfrak{x}) \notin \mathcal{L}(\mathcal{R}_N) \\ \wedge \\ \langle \tilde{\mathcal{P}}_2(\text{st}), \mathcal{V}_{\text{ARG}}(\text{ivk}, \mathfrak{x}) \rangle = 1 \end{array} \mid \begin{array}{c} \text{srs} \leftarrow \mathcal{G}_{\text{ARG}}(1^\lambda, N) \\ (\mathfrak{i}, \mathfrak{x}, \text{st}) \leftarrow \tilde{\mathcal{P}}_1(\text{srs}) \\ (\text{ipk}, \text{ivk}) \leftarrow \mathcal{I}_{\text{ARG}}^{\text{srs}}(\mathfrak{i}) \end{array} \right] = \text{negl}(\lambda) .$$

All of the constructions in this paper achieve the stronger property of *knowledge soundness* as defined in [CHMMVW20, Section 7], using the same proof as in [CHMMVW20].

4.10.2 Elastic PIOP to argument compiler

We show how to compile our elastic PIOPs into an elastic argument systems elastic commitment schemes. We follow the compiler construction analysed in [CHMMVW20, Theorem 8.1].

Construction 11. *As the input PIOP has a public-coin verifier and non-adaptive queries, we assume that all of the verifier queries in the PIOP take place at the end.*

Setup. $\text{srs} \leftarrow \mathcal{G}_{\text{ARG}}(1^\lambda, N)$: Let $D := \max_{i \in [k]} \max_{j \in [o(i)]} d(|\mathfrak{i}|, i, j)$. The generator \mathcal{G}_{ARG} runs $\text{PC.Setup}(1^\lambda, D)$ to get output ck , which contains the description of a finite field $\mathbb{F} \in \mathcal{F}$.

Offline phase. In the offline phase (“0-th round”), the indexer \mathcal{I}_{ARG} receives as input a commitment key ck , a field $\mathbb{F} \in \mathcal{F}$ and an index \mathfrak{i} for \mathcal{R} . Then \mathcal{I}_{ARG} runs the IOP indexer \mathcal{I} , which outputs $o(0)$ polynomials $\mathbf{p}_{0,1}, \dots, \mathbf{p}_{0,o(0)} \in \mathbb{F}[X]$ of degrees at most $d(|\mathfrak{i}|, 0, 1), \dots, d(|\mathfrak{i}|, 0, o(0))$ respectively. For each polynomial $\mathbf{p}_{0,i}$, the indexer \mathcal{I}_{ARG} computes $C_{0,i} = \text{PC}_s.\text{Com}(\text{ck}, \mathbf{p}_{0,i}, \perp)$.

Online phase. In the online phase, given the commitment key ck , an instance \mathfrak{x} and witness \mathfrak{w} such that $(\mathfrak{i}, \mathfrak{x}, \mathfrak{w}) \in \mathcal{R}$, the prover \mathcal{P}_{ARG} receives $(\mathbb{F}, \text{ck}, \mathfrak{i}, \mathfrak{x}, \mathfrak{w})$ and the verifier \mathcal{V} receives $(\mathbb{F}, \mathfrak{x}, \text{ck})$ and the commitments produced by $\mathcal{I}_{\text{ARG}}(\mathbb{F}, \mathfrak{i}, \text{ck})$. The prover \mathcal{P}_{ARG} and the verifier \mathcal{V}_{ARG} interact over $k + 2 = k(|\mathfrak{i}|) + 2$ rounds.

- For $i \in [k]$, in the i -th round of interaction, the verifier \mathcal{V}_{ARG} runs \mathcal{V} and forwards its message $\rho_i \in \mathbb{F}^\times$ to the prover \mathcal{P}_{ARG} . The prover \mathcal{P}_{ARG} forwards this message to \mathcal{P} which replies with $o(i)$ oracle polynomials $\mathbf{p}_{i,1}, \dots, \mathbf{p}_{i,o(i)} \in \mathbb{F}[X]$ of degrees at most $d(|\mathbb{i}|, i, 1), \dots, d(|\mathbb{i}|, i, o(i))$ respectively. For each polynomial $\mathbf{p}_{i,j}$, the prover \mathcal{P}_{ARG} computes the commitment $C_{i,j} := \text{PC.Com}(\text{ck}, \mathbf{p}_{i,j}, \perp)$ and sends $C_{i,j}$ to the verifier \mathcal{V}_{ARG} .
- The verifier \mathcal{V}_{ARG} runs \mathcal{V} to obtain opening queries, each consisting of an evaluation point $z \in \mathbb{F}$, and a pair of indices (i, j) which specify an oracle $\mathbf{p}_{i,j}$. The verifier \mathcal{V}_{ARG} forwards all of the queries to the prover \mathcal{P}_{ARG} . Then, for each query (z, i, j) , the prover \mathcal{P}_{ARG} computes $v := \mathbf{p}_{i,j}(z) \in \mathbb{F}$, computes $\pi := \text{PC.Open}(\text{ck}, \mathbf{p}_{i,j}, z)$, and sends v and π to \mathcal{V}_{ARG} . The verifier \mathcal{V}_{ARG} forwards the evaluation point v to \mathcal{V} .
- The verifier \mathcal{V}_{ARG} computes $b \leftarrow \text{PC.Check}(\text{ck}, C, z, v, w)$ for each opening query. If $b = 1$ for every execution of PC.Check , and \mathcal{V} accepts, then \mathcal{V}_{ARG} accepts. Otherwise, \mathcal{V}_{ARG} rejects.

4.10.3 Proof of Theorem 4.10.1

The completeness and soundness properties, and indexer, prover and verifier efficiency for the time-efficient argument produced by the compiler follow from the proof of [CHMMVW20, Theorem 8.1].

To prove Theorem 4.10.1, it remains to describe a space-efficient implementation of the prover algorithm Construction 11.

Construction 12 (space-efficient prover). *We describe a space-efficient prover algorithm for the online phase of ARG.*

For $i \in [k]$, in the i -th round of interaction, \mathcal{P}_{ARG} initializes $o(i)$ different sessions with the space efficient commitment algorithm PC.Com , using $o(i)$ different sessions for the streaming oracle $\mathcal{S}(\text{ck})$, and runs the space-efficient implementation of the prover algorithm \mathcal{P} (for the i -th round) $k_{\text{PC.Com}}$ times, forwarding each coefficient of each polynomial $\mathbf{p}_{i,j}(X)$ to the correct session for PC.Com .

To answer the verifier's queries, \mathcal{P}_{ARG} initializes q different sessions with the PC.Open algorithm, using q different sessions for the streaming oracle $\mathcal{S}(\text{ck})$. For each $i \in [k]$, consider the evaluation queries $(z_1, i, j_1), \dots, (z_t, i, j_t)$ made during the i -th round. The prover \mathcal{P}_{ARG} executes \mathcal{P} , po times, to produce po passes over the coefficients of the polynomials $\mathbf{p}_{i,0}(X), \dots, \mathbf{p}_{i,o(i)}(X)$. The prover \mathcal{P}_{ARG} forwards the coefficients produced during each pass to the correct session for the PC.Open algorithm. During the first pass over each of the polynomials $\mathbf{p}_{i,j_r}(X)$, \mathcal{P}_{ARG} computes $\mathbf{p}_{i,j_r}(z_r)$ using Horner's rule.

Note that input/output orderings of the \mathcal{P} algorithm and the PC.Com and PC.Open algorithms ensure that the polynomials produced by \mathcal{P} are fed to PC.Com and PC.Open in the correct order.

We justify each of the complexity parameters of Construction 12 in turn.

Prover time. For each round $i \in [k]$, the prover \mathcal{P}_{ARG} runs the prover algorithm \mathcal{P} for the i -th round a total of $k_{\text{PC.Com}} + po$ times. In total, this incurs a time cost of $k_{\text{PC.Com}} + po$ complete executions of \mathcal{P} .

In addition, \mathcal{P}_{ARG} commits to $\sum_{i=0}^{k-1} o(i)$ polynomials using PC.Com, and for q queries, evaluates a polynomial of degree D and runs PC.Open.

This gives a total prover time of $(k_{\text{PC.Com}} + \text{po})t'_{\mathcal{P}} + \sum_{i=0}^{k-1} o(i) \cdot t'_{\text{PC.Com}} + q \cdot (t'_O + \text{te})$.

Prover space. For each round $i \in [k]$, the prover \mathcal{P}_{ARG} runs the prover algorithm \mathcal{P} for the i -th round, while running $o(i)$ executions of PC.Com in parallel. This gives space costs of $s'_{\mathcal{P}} + o(i)s'_{\text{PC.Com}}$.

Subsequently, \mathcal{P}_{ARG} runs \mathcal{P} again to answer evaluation queries, running q executions of PC.Open and computing evaluations of at most q polynomials in parallel at any time. This gives space costs of $q \cdot s'_O + O(q)$.

Number of passes. The prover \mathcal{P}_{ARG} runs \mathcal{P} a total of $k_{\text{PC.Com}} + \text{po}$ times in order to provide enough passes for PC.Com and PC.Open. Each execution of \mathcal{P} uses at most $k_{\mathcal{P}}$, giving a total of $(k_{\text{PC.Com}} + \text{po}) \cdot k_{\mathcal{P}}$.

Chapter 5

Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust

Data-sharing systems are often used to store sensitive data. Both academia and industry have proposed numerous solutions to protect user privacy and data integrity from a compromised server. Practical state-of-the-art solutions, however, use weak threat models based on *centralized trust*—they assume that part of the server will remain uncompromised, or that the adversary will not perform active attacks. We propose Ghostor, a data-sharing system that, using only *decentralized trust*, (1) hides user identities from the server, and (2) allows users to detect server-side integrity violations. To achieve (1), Ghostor avoids keeping any per-user state at the server, requiring us to redesign the system to avoid common paradigms like per-user authentication and user-specific mailboxes. To achieve (2), Ghostor develops a technique called *verifiable anonymous history*. Ghostor leverages a blockchain *rarely*, publishing only a single hash to the blockchain *for the entire system* once every epoch. We measured that Ghostor incurs a 4–5x throughput overhead compared to an insecure baseline. Although significant, Ghostor’s overhead may be worth it for security- and privacy-sensitive applications.

This work was previously published in [HKP20].

5.1 Introduction

Systems for remote data storage and sharing have seen widespread adoption over the past decade. Every major cloud provider offers it as a service (e.g., Amazon S3, Azure Blobs), and it is estimated that 39% of corporate data uploaded to the cloud is related to file sharing [Kep15]. Given the relentless attacks on servers storing data [Ide18], a long-standing problem in academia [LKMS04; FZFF10; SCCKMS10; KL15; KFPC16; Bla93; GSMB03a; HAJSS14; LCSJLB14; Pop+14] and industry [Cry; Inc; Key; Pri; Vir] has been to provide useful security guarantees even when the storage server, and some users, are compromised by an adversary.

To address this, early systems [GSMB03a; KRSWF03] have users encrypt and sign files. However, a sophisticated adversary can still:

- observe metadata about *users' identities* [CWWZ10; GMNRS16; IKK12; Wha12]. Even if the files are encrypted, the adversary sees which users are sharing a file, which user is accessing a file at a given time, and the list of users in the system. Figure 5.1 shows an example where the attacker can conclude that Alice has cancer from such metadata. Further, this allows the attacker to learn the graph of user social relations [Sea18; SH12].
- perform active attacks. Despite the signatures, an adversary can revert a file to an earlier state as in a *rollback attack*, or hide users' updates from each other as in a *fork attack*, without being detected. These are dangerous if, for example, the shared file is Alice's medical profile, and she does not learn that her doctor changed her treatment.

Research over the past 15 years has striven to mitigate these attacks by providing *anonymity*—hiding users' identities from the storage server—or *verifiable consistency*—enabling users to detect rollback and fork attacks. In achieving these stronger security guarantees, however, state-of-the-art systems employ weaker threat models that rely on centralized trust: a trust assumption on a *few specific machines*. For example, they rely on a trusted party [SS13; MMRS17], split the server into two components assuming one is honest [KH12; PP12; KFPC16], or assume the adversary is honest-but-curious (not malicious) [ZYG11; BLNN15; MMRS15; BHKP16] meaning the attacker does not change the server's data or execution.

Attackers have notoriously performed highly targeted attacks, spreading malware with the ability to modify software, files, or source code [Zeta; Zetb; Lem]. In such attacks, a determined attacker can compromise any *few central servers*. Ideally, we would avoid *any trust* in the server or other clients, but unfortunately, that is impossible: Mazières and Shasha [MS02] proved that, if one cannot assume that clients are reliably online [KL15], clients cannot detect fork attacks without placing some trust in the server. Hence, this paper asks the question: Can we achieve strong privacy and integrity guarantees in a data-sharing system without relying on *centralized trust*?

To answer this question, we design and build Ghostor, an object store based on *decentralized trust* that achieves *anonymity* and *verifiable linearizability* (abbreviated VerLinear). At a high level, anonymity¹ means that the protocol does not reveal directly to the server any user identity with any request, as previously defined in the secure storage literature [ZYG11; KH12; PP12; MMRS15]. As

¹Outside of secure storage, *anonymity* is sometimes defined differently. In secure messaging, for example, an anonymous system is expected to hide the timing of accesses [HLZZ15a] and which files/mailboxes are accessed, but not necessarily the system's membership [CGBM15].

E2EE Systems	Ghostor's Anonymous E2EE
Alice and BobMD have accounts	This system has unknown users
Alice owns medical profile file F	File F exists with unknown owner
Alice and BobMD have access to F	F's Access Control List is unknown
Alice reads F at 2pm	Unknown reads F at 2pm
BobMD writes to F at 3pm	Unknown (could be same as above) writes to F at 3pm

Google search says BobMD is an oncologist. Each of these tells me that Alice might suffer from cancer.




Figure 5.1: An example of what a server attacker sees in a typical end-to-end encrypted (E2EE) system versus Ghostor’s Anonymous E2EE

shown in Figure 5.1, the server does not see which user owns which objects, which users have read or write permissions to a given object, or even who are the users of the system. The server essentially sees **ghosts** accessing the **storage**, hence the name “**Ghostor**.” VerLinear means clients can verify that each write is reflected in later reads, except for benign reordering of concurrent operations as formalized by linearizability [HW90]. To achieve these properties, we build Ghostor’s integrity on top of a consistent storage primitive based on decentralized trust, like a blockchain [Nak08; But+13; Zcab] or verifiable ledger [HB17; ELC], while using it only *rarely*.

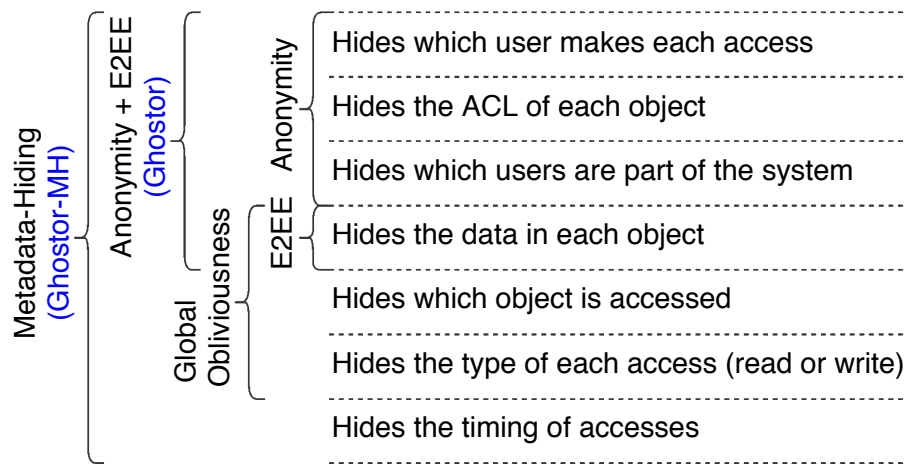


Figure 5.2: Information leakage in a data-sharing system and associated privacy properties

5.1.1 Hiding User Identities

Achieving anonymity in practical data-sharing systems like Ghostor is difficult because common system design paradigms, like user login, per-user mailboxes on the server, and client-side caching, let the server track users. As we explain in Section 5.4, even using user-specific keys to sign updates to data objects can reveal to the server which user performed the update, and requires knowledge of the ACL to check that the signer is an authorized user. We re-architect the system to avoid these paradigms (Section 5.4), using data-centric key distribution and encrypted key lists instead of server-side ACLs. Like prior systems [GSML16; AT83; KHAPC19], Ghostor uses cryptographic keys as capabilities, allowing the server and other users to verify that each access is performed by an authorized user. Ghostor also leverages this technique to achieve anonymity by having all users authorized to perform a particular operation on an object (e.g., all users with read access to an object) *share* the same capability for performing that operation on that object, and by distributing these capabilities to users without revealing ACLs to the server. We find this technique, *anonymously distributed shared capabilities*, interesting because anonymity is not typically a goal of public-key access control [GSML16; AT83] or capability-based systems [Lev84; SSF99; MWC10].

An additional challenge is to guard against resource abuse while preserving anonymity. This is typically done by enforcing per-user resource quotas (e.g., Google Drive requires users to pay for additional space), but this is incompatible with Ghostor’s anonymity. One solution is for users to pay for each operation via an anonymous cryptocurrency (e.g., Zcash [Zcab]), but this puts an expensive blockchain operation in the critical path. To avoid this, Ghostor leverages blind signatures [CPS94; Cha83; Cha84] to allow a user to pay the Ghostor server for service in bulk and in advance, while removing the linkage between payments and operations.

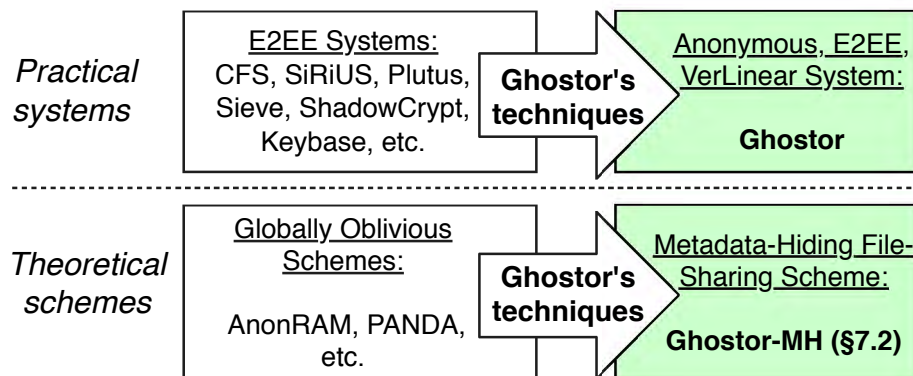


Figure 5.3: Ghostor’s contributions. Ghostor’s techniques can be applied to both oblivious and non-oblivious systems.

Relationship to obliviousness. Figure 5.2 positions Ghostor’s anonymity with respect to other privacy properties. Global obliviousness [BHKP16; MMRS17], which hides which *object* is accessed across all uncompromised objects and users in the system, is orthogonal to Ghostor’s anonymity, which hides which *user* performs each access. Obliviousness and anonymity are also

complementary: (1) In some cases, without obliviousness, users may be identified based on access patterns. (2) Without anonymity, knowing which user issued a request may reveal information about what data that request may access. Ghostor’s techniques for anonymity are a *transformation* (Figure 5.3):

- If to an E2EE system, we obtain **Ghostor, an anonymous E2EE system**.
- If applied to a globally oblivious scheme, we obtain **Ghostor-MH, a data-sharing scheme that hides all metadata** (except when initializing a group of objects or redeeming payments, as explained in Section 5.7.2).

Hiding metadata from a malicious adversary, as in Ghostor-MH, is a very strong guarantee—existing globally oblivious schemes inherently reveal user identities [MMRS17] or assume the adversary is honest-but-curious [MMRS15; BHKP16]. However, globally oblivious data-sharing schemes, like Ghostor-MH, are theoretical schemes that are far from practical. Thus, Ghostor-MH is only a proof of concept demonstrating the power of Ghostor’s techniques to lift a globally oblivious scheme all the way to virtually zero leakage for a malicious adversary.

5.1.2 Verifiable Consistency

To provide VerLinear, prior work has clients sign hashes [KL15] so the clients can verify that they see the same hash, or store hashes on a separate hash server [KFPC16], trusted not to collude with the storage server. Neither technique can be used in Ghostor: client signatures are at odds with anonymity, and the hash server is a trusted party, which Ghostor aims to avoid.

One way to adapt the prior designs to Ghostor’s decentralized trust is to store hashes on a blockchain, which can be accomplished by running the hash server in a smart contract. Unfortunately, this design is **too slow to be practical**. The client posts a hash on the blockchain for every object write, which is expensive: blockchains incur high latency per transaction, have low transaction throughput, and require cryptocurrency payment for each transaction [But+13; Nak08; Zcab].

To sidestep the limitations of a blockchain, we design Ghostor to only interact with the blockchain rarely and outside of the critical path. Ghostor divides time into intervals called *epochs*. At the end of each epoch, the Ghostor server publishes to the blockchain a small *checkpoint*, which summarizes the operations performed during that epoch for all objects and users in the system. Each user can then verify that the results of their accesses during the epoch are consistent with the checkpoint. The consistency properties of a blockchain ensure all clients see the same checkpoint, so the server is committed to a single history of operations and cannot perform a fork attack. Commit chains [KZFMSG18] and monitoring schemes [Bon16; TD17] are based on similar checkpoints, but Ghostor applies them to object storage while maintaining users’ anonymity.

A significant obstacle is that a hash-chain-based history is not amenable to concurrent appends. Each entry in the history contains the hash of the previous entry, causing one operation to fail if a concurrent operation appends a new entry. Existing techniques for concurrent operations, such as SUNDR’s VSLs [LKMS04], reveal *per-user* version numbers that would undermine Ghostor’s anonymity. Our insight in Ghostor is to have the *server*, not the client, populate the hash of the previous entry when appending a new entry. To make this safe despite a malicious adversary, we

carefully design a conflict resolution strategy, involving multiple *linked* entries in the history for each write, that prevents attackers from manipulating data via replay or time-stretch attacks.

We call the resulting design a *verifiable anonymous history*.

Goal	Technique
Anonymous user access control	Anonymously distributed shared capabilities (Section 5.4)
Anonymous server integrity verification	Verifiable anonymous history (Section 5.5)
Concurrent operations on a single object	Optimized GETs, two-phase protocol for PUTs (Section 5.5.4)
Anonymous resource abuse prevention	Blind signatures and proof of work (Section 5.6)
Hiding user IP addresses	Anon. network, e.g., Tor (Section 5.8)

Table 5.1: Our goals and how Ghostor achieves each one

5.1.3 Summary of Contributions

Our goals and techniques are summarized in Table 5.1. Overall, this paper’s contributions are:

- We design an object store providing anonymity and verifiable linearizability based only on *decentralized trust*.
- We develop techniques to (1) share capabilities for anonymity and distribute them anonymously, (2) create and checkpoint a verifiable anonymous history, and (3) support concurrent operations on a single object with a hash-chain-based history.
- We combine these with existing building blocks to instantiate Ghostor, an object store with anonymity and VerLinear.
- We also apply these to a globally oblivious scheme to instantiate Ghostor-MH, which hides nearly all metadata.

We also implemented Ghostor and evaluated it on Amazon EC2. Overall, Ghostor brings a 4-5x throughput overhead on top of a simplistic and completely insecure baseline. There are two types of latency overhead. Completing an individual operation takes several seconds. Afterward, it may take several minutes for a checkpoint to be incorporated into the blockchain, to confirm that no active attack has occurred for a batch of operations. We explain how these latencies play out in the context of a particular application, EHR Sharing (Section 5.7.1).

5.2 System Overview

Ghostor is an object store, which stores unstructured data items (“objects”) and allows shared access to them by multiple users. We instantiate Ghostor as an object store (as in Amazon S3 or Azure Blobs) because it is a basic primitive on top of which more complex systems can be built. Figure 5.4 illustrates Ghostor’s architecture. Multiple users, with separate clients, have shared access to objects on the Ghostor server.

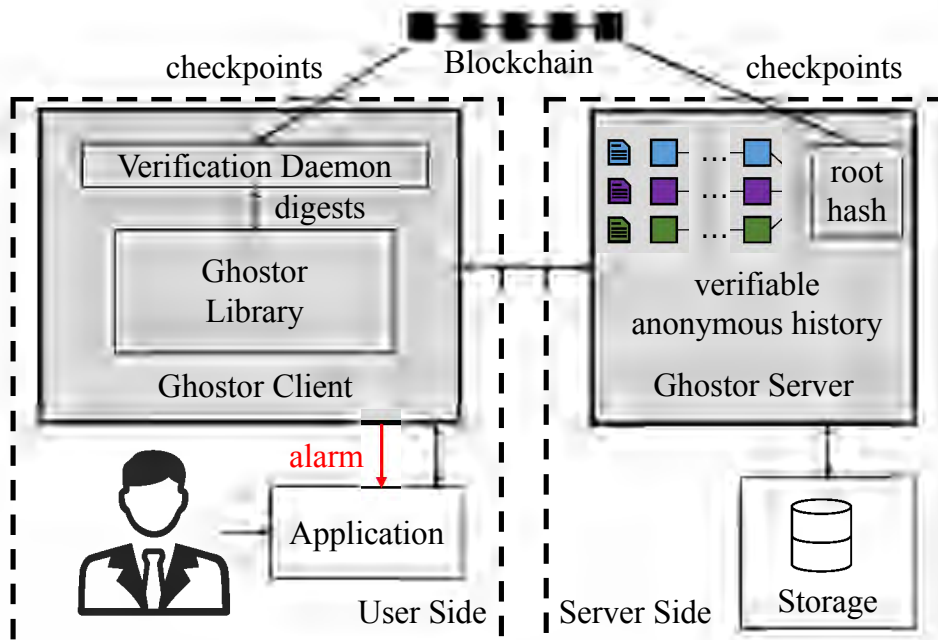


Figure 5.4: System overview of Ghostor. Shaded areas indicate components introduced by Ghostor.

Server. The Ghostor storage server processes requests from clients. At the end of each epoch, the server generates a single small checkpoint and publishes it to the blockchain.

Client. The client software consists of a Ghostor library, linked into applications, and a verification daemon, which runs as a separate process. The Ghostor library receives requests from the application and interacts with the server to satisfy each request. Upon accessing an object, the library forwards a digest summarizing the operation to the verification daemon. At the end of each epoch, the daemon (1) fetches object histories from the server, (2) verifies that they are consistent with the server’s checkpoint on the blockchain, and (3) checks that the digests collected during the epoch are consistent with the object histories, as explained in Section 5.5.

The daemon stores the user’s keypair. If a user loses her secret key, she loses access to all objects that she created or was granted access to. Similarly, an attacker who steals a user’s secret key can impersonate that user. To securely back up her key on multiple devices, a user can use standard techniques like secret sharing [Sha79; WMZV16; Sec]. A user who accesses Ghostor from multiple devices uses the same key on all devices.

Application developers interact with Ghostor using the API below. Developers can work with usernames, ACLs, and object IDs, but Ghostor clients will not expose them to the Ghostor server. Below is a high-level description of each API call; a step-by-step technical description is in the full version[HKP20].

- ◇ **create_user()**: Creates a Ghostor user by generating keys for a new user. This operation runs entirely in the Ghostor client—the server does not know this operation was invoked.
- ◇ **user.pay(*sum*)**: Users pay the server through an anonymous cryptocurrency such as Zcash [Zcab], and obtain *tokens* from the server proportional to the amount paid. These tokens can later be *anonymously redeemed* and used as proof of payment when invoking the below API functions.
- ◇ **user.create_object(*id*)**: Creates an object with ID *id*, owned by user who invokes this. The client expends one token obtained from a previous call to **pay**. The *id* can be a meaningful name (e.g., a file path). It lives only within the client—the server receives some cryptographic identifier—so different clients can assign different *ids* to the same object.
- ◇ **user.set_acl(*id*, *acl*)**: The user who invokes this must be the owner of the object with ID *id*. This function sets a new ACL for that object. For simplicity, only the owner of an object can set its ACL, but Ghostor can be extended to permit other users as well. The client encodes *acl* into an object header that hides user identities, as in Section 5.4. If new users are given access, they are notified via an out-of-band channel. Existing data-sharing systems also have this requirement; for example, Dropbox and Box send an email with an access URL to the user. In Ghostor, all keys are transferred in-band; the out-of-band channel is used only to *inform* the user that she has been given access. Ghostor does not require a specific out-of-band channel; for example, one could use Tor [DMS04a] or secure messaging [TGLZZ17a; HLZZ15a].
- ◇ **user.get_object(*id*)**, **user.put_object(*id*, *content*)**: The user can GET or PUT an object if permitted by its ACL.

5.3 Threat Model and Security Guarantees

Against a malicious attacker who has compromised the server, Ghostor provides:

- verifiable linearizability, as described in Section 5.3.2, and
- a notion of user anonymity, described in Section 5.3.3: briefly, it does not reveal user identities, but reveals object access patterns. Ghostor-MH additionally hides access patterns.

Ghostor does not protect against attacks to availability. Nevertheless, its anonymity makes it more difficult for the server to selectively deny service to (or fork views of) certain users. Users, and the Ghostor client instances running on their behalf, can be malicious and can collude with the server.

Formal definitions and proofs for these properties require a large amount of space, so we relegate them to the full version[HKP20]. Below, we include only *informal* definitions.

5.3.1 Assumptions

Ghostor is designed to derive its security from decentralized trust. Thus, our threat model assumes an adversary who can compromise any few machines, as described below.

Blockchain. Ghostor makes the standard assumption that the blockchain is immutable and consistent (all users see the same transaction history). This is based on the assumption that, in order to attack a blockchain, the adversary cannot simply compromise a few machines, but rather a significant fraction of the world’s computing power. Ghostor’s design is not tied to a specific blockchain. Our implementation uses Zcash [Zcab] because it supports both public and private transactions; we use Zcash’s private transactions for Ghostor’s anonymous payments. The privacy guarantees of Zcash can be implemented on top of other blockchains as well [BS+14].

Network. We assume clients communicate with the server in a way that does not reveal their network information. This can be done using mixnets [Cha81a] or secure messaging [TGLZZ17a; HLZZ15a] based on decentralized trust. Our implementation uses Tor [DMS04a].

5.3.2 Verifiable Linearizability

If an attack is immediately detectable to a user—for example, if the server fails to honor payment or provides a malformed response (e.g., bad signature)—we consider it an attack on *availability*, which Ghostor does not prevent.

Clients should be able to detect active attacks, including fork and rollback attacks. Some reordering of concurrent operations, however, is benign. We use *linearizability* [HW90] to define when reordering at the server is considered benign or malicious. *Informally*, linearizability requires that after a PUT completes, all later GETs return the value of either (1) that PUT, (2) a PUT that was concurrent with it, or (3) a PUT that comes after it. We provide a more formal definition in the full version[HKP20]. Ghostor provides *verifiable linearizability* (abbreviated *VerLinear*). This means that if the server deviates from linearizability, clients can detect it at the end of the epoch. We discuss how to choose the epoch length in Section 5.9. Ghostor does not provide consistency guarantees for malicious user, or for objects for which a malicious user has write access.

Guarantee 5.3.1 (Verifiable linearizability). *For any object F and any list E of consecutive epochs, suppose that, for each epoch in E , the set of honest users who ran the verification procedure includes all writers of F in that epoch (or is nonempty if F was not written). If the server did not linearizably execute the operations that verifying clients performed in the epochs that they verified, then at least one of the verifying clients will encounter an error in the verification procedure and can generate a proof that the server misbehaved.*

5.3.3 Anonymity

As explained in Section 5.1.1, Ghostor’s anonymity means that the server sees no user identities associated with any action. In particular, an adversary controlling the server cannot tell which user accesses each object, which users are authorized to access each object, or which users are part of the system.

Ghostor. We informally define Ghostor’s privacy via a *leakage function*: what the server learns when a user makes each API call (Section 5.2). For **create_object** – **put_object**, the server learns the object ID, the type of operation, and whether the user is authorized according to the object’s ACL (past and present). The server also sees the time of the operation, and the size of the encrypted ACL and encrypted object, which can be hidden via padding at an extra cost. **create_user** leaks no information to the server, and **pay** reveals the sum paid and when. The server learns no user identities, no object contents, and no ACLs. If the attacker has compromised some users, he learns the contents of objects those users can access, including prior versions encrypted under the same key. Collectively, the verification daemons leak the number of clients performing verification for each object. If all clients in an object’s ACL are honest and running, this equals the ACL size. If the ACL is padded to a maximum size, the owner should run verification more times to hide the ACL size. Ghostor does *not* hide access patterns or timing (Figure 5.2). An adversary who uses this information cannot see the contents of files and ACLs because they are encrypted. But such an adversary could try to deduce correlations between which users issue different operations based on access patterns and timing, and in some cases, identify the user based on that information. This can be partially mitigated by carefully designing the application using Ghostor (Section 5.4.5). In contrast, Ghostor-MH does hide access patterns. In the full version[HKP20], we formally define Ghostor’s privacy guarantee in the simulation paradigm of Secure MPC.

Ghostor-MH. We informally define Ghostor-MH’s privacy via a leakage function, as above. **create_object** reveals that a group of objects was created. **set_acl**, **get_object**, and **put_object** reveal nothing if the object’s ACL contains only honest users; otherwise, they reveal which object was accessed. **create_user** and **pay** have the same leakage as described for Ghostor above. The leakage function also includes the total number of honest users in the system.

5.4 Hiding User Identities

System design paradigms used in typical data-sharing systems are incompatible with anonymity. We identify the incompatible system design patterns and show how Ghostor replaces them. Ultimately, we arrive at *anonymously distributed shared capabilities*, which allow Ghostor to enforce access control for anonymous users without server-visible ACLs.

Keypair or Key	Description
(PVK, PSK)	Signing keypair used to set ACL
(RVK, RSK)	Signing keypair used to get object
(WVK, WSK)	Signing keypair used to put object
(OSK)	Symmetric key for object contents

Table 5.2: Per-object keys in Ghostor. The server uses the global signing keypair (SVK, SSK) to sign digests for objects.

5.4.1 No User Login or User-Specific Mailboxes

Data-sharing systems typically have some storage space on the server, called an *account file*, dedicated to a user's account. For example, Keybase [Key] has a user account and Mylar [Pop+14] has a user mailbox where the user receives a key to a new file. Accesses to the account file, however, can be used to *link user operations*. As an example, suppose that when a user accesses an object, her client first retrieves the decryption key from a user-specific mailbox. This violates anonymity because the server can tell whether or not two accesses were made by the same user, based on whether the same mailbox was accessed first. Instead, Ghostor's anonymity requires that *any sequence of API calls (Section 5.2) with the same inputs, when performed by any honest user, results in the same server-side accesses*.

Ghostor does not have any user-specific storage as in existing systems. To allow in-band key exchange, Ghostor associates a *header* with each object. The object header functions like an object-specific mailbox, in that it is used to distribute the object's keys among users who have access to the object. Unlike a user-specific mailbox, it preserves anonymity because, for a given object, each user reads the *same* header before accessing it.

5.4.2 No Server-Visible ACLs

An honest server must be able to prevent unauthorized users from modifying objects, and users must be able to verify that objects returned by the server were produced by authorized writers. This is

typically accomplished by having writers sign objects, and having the server check that the user who signed the object is on the object’s ACL. However, this requires the ACL to be visible to the server, which violates anonymity.

We observe that by switching to a design based on *shared capabilities*, we can allow the server and other users to verify that writes are indeed made by authorized users, without requiring the server or other users to know the ACL of the object, or which users are authorized. Every Ghostor object has three associated signing keypairs (Table 5.2). All users of the object (and the server) know the verifying keys PVK, RVK, and WVK because PVK is the name of the object, and RVK and WVK are in the object header; the associated signing keys PSK, RSK, and WSK are *capabilities* that grant access to set the ACL, get the object, and put the object, respectively. To distribute these capabilities to users in the object’s ACL, the owner places a *key list* in the object header. The key list contains, for each user in the ACL, a list of capabilities encrypted under that user’s public key. The owner randomly shuffles the key list and, optionally, pads it to a maximum size to hide each user’s position. If a user has read/write access to an object, her entry in the key list contains WSK, RSK, and OSK; a user with only read access is given a dummy key instead of WSK. Crucially, different users with the same permission *share the same capability*, so the server cannot distinguish between users on the basis of which capability they use. When accessing an object, a user downloads the header and decrypts her entry in the key list to obtain OSK (used to decrypt the object contents) and her capabilities for the object.

Users sign updates to the object with WSK, allowing the server and other users to verify that each update is made by a user with write access. PSK is stored locally by the owner and is used to sign the header. The owner can set the object’s ACL by (1) freshly sampling (RVK, RSK), (WVK, WSK), and OSK, (2) re-encrypting the object with OSK and signing it with WSK, (3) creating a new object header with an updated key list, (4) signing the new header with PSK, and (5) uploading it to the server. (RVK, RSK) will be relevant in Section 5.5.

Ghostor’s object layout is summarized in Figure 5.5.

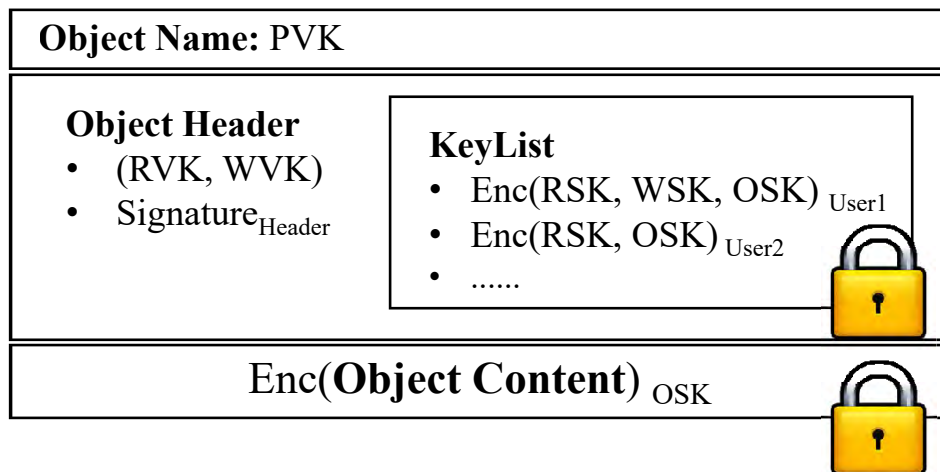


Figure 5.5: Object layout in Ghostor

5.4.3 No Server-Visible User Public Keys

Prior systems [LKMS04] reveal the user's public key to the server when the client interacts with it. For example, SUNDR requires users to provide a signature along with each operation. First, the signature itself could leak the user's public key. Second, to check the legitimacy of writes, the server needs to know the user's public key to verify the signature. The server can use the public key as a *pseudonym* to track users.

The key list in Section 5.4.2, however, potentially leaks users' public keys: each entry in the key list is a set of capabilities encrypted under a user's public key, but public-key encryption is only guaranteed to hide the message being encrypted, not the public key used to encrypt it. For example, an RSA ciphertext leaks which public key was used for encryption. Therefore, Ghostor uses *key-private* encryption [BBDP01], which is guaranteed to hide both the message and the public key.

In summary, Ghostor has users *share* capabilities for anonymity, and then distributes the capabilities anonymously, without revealing ACLs to the server. We call the resulting technique *anonymously distributed shared capabilities*.

5.4.4 No Client-Side Caching

Assuming that an object's ACL changes rarely, it may seem natural for clients to locally cache an object's keypairs (RVK, RSK) and (WVK, WSK), to avoid downloading the header on future accesses to that object. Unfortunately, the mere fact that a client did not download the header before performing an operation tells the server that the *same user* recently accessed that object. As a result, Ghostor's anonymity prohibits user-specific caching. That said, *server-side* caching of commonly accessed objects is allowed.

5.4.5 Careful Application Design

Ghostor does not hide access patterns or timing information from the server. A sophisticated adversary could, for example, deny or delay accesses to a particular object and see how access patterns shift, to try and deduce which user made which accesses. Therefore, one should carefully design the application using Ghostor to avoid leaking user identities in its access patterns. For example, just as Ghostor has no client-side caching or user-specific mailboxes, an application using Ghostor should avoid caching data locally to avoid requests to the server or using an object as a user-specific mailbox. Note that Ghostor-MH hides these access patterns.

5.5 Achieving Verifiable Consistency

Ghostor’s *verifiable anonymous history* achieves the “verifiable equivalent” of a blockchain for critical-path operations, while using the underlying blockchain rarely. It consists of: (1) a hash chain of digests, (2) periodic checkpoints on a real blockchain, and (3) a verification procedure that does not require knowledge of user identities.

5.5.1 Hash Chain of Digests in Ghostor

We now achieve fork consistency for a single object in Ghostor using techniques inspired from SUNDR [LKMS04], but modified because SUNDR is not anonymous. Each access to an object, whether a GET or a PUT, is summarized by a *digest* shown in Table 5.3. The object’s history is stored as a chain of digests.

Field	Description
Epoch	epoch when operation was committed
PVK, WVK, RVK	permission/writer/reader verifying key
Hash _{prev}	hash of previous digest in chain
Hash _{keylist}	hash of key list
Hash _{data}	hash of object contents
Sig _{client}	client signature with RSK, WSK, or PSK
Sig _{server}	server signature using SSK
nonce	random nonce chosen by client

Table 5.3: A digest for an operation in Ghostor

To access the object, a client first produces a digest summarizing that operation as in Table 5.3. This requires fetching the object header from the server, so that the client can obtain the secret key (RSK, WSK, or PSK) for the desired operation. Then the client fetches the latest digest for the object and computes Hash_{prev} in the new digest. To GET the object, the client copies Hash_{data} from the latest digest; to PUT it, the client hashes the new contents to obtain Hash_{data}. If the client is changing permissions, then Hash_{keylist} is calculated from the new header; otherwise, it is copied from the latest digest.

Then the client signs the digest with the appropriate key and provides the signed digest to the server. The server signs the digest using SSK, appends it to a log, and returns the signed digest and the result of the operation. At the end of the epoch, the client downloads the digest chain for that object and epoch, and verifies that (1) it is a valid history for the object, and that (2) it contains the operations performed by that client. We specify protocol details in the full version[HKP20].

Ghostor’s digests differ from SUNDR in two main ways. First, for anonymity, a client does not sign digests using the user’s secret key, but instead uses RSK, WSK, or PSK, which can be verified without knowing the user’s public key. When inspecting the digest, the server no longer learns

which user performed the operation, only that the user has the required permission. Second, each digest is signed by the server. Thus, if the server violates linearizability, the client can assemble the offending digests into a *proof of misbehavior*.

5.5.2 Checkpoint and Verification

The construction so far is susceptible to fork attacks [LKMS04], in which the server presents two users with different views over the same object. To detect fork attacks, Ghostor requires the server to produce a *checkpoint* at the end of each epoch, consisting of the hash of the object's latest digest and the epoch number, and publish the checkpoint to the blockchain. The *verification procedure* run by a client consists of fetching the checkpoint from the blockchain, checking it corresponds to the hash for the last digest in the list of digests obtained from the server, and running the verification in Section 5.5.1. The blockchain guarantees that all users see the same checkpoint. This prevents the server from forking two users' views, as the latest digests for two different views cannot both match the published checkpoint. In this way, we bootstrap the blockchain's consistency guarantees to achieve verifiable consistency over an entire epoch of operations.

5.5.3 Multiple Objects per Checkpoint

So far, the server puts one checkpoint in the blockchain *per object*, which is undesirable when there are many objects. We address this as follows. The server computes the hash of the final digest of each object, builds a Merkle tree over those hashes, and publishes the root hash in the blockchain as a single checkpoint for all objects. To verify integrity at the end of an epoch, a Ghostor client fetches the digest chain from the server for objects that are either (1) accessed by the client during the epoch or (2) owned by the client's user. It verifies that all operations that it performed on those objects are included in the objects' digest chains. Then, it requests Merkle proofs from the server to check that the hash of the latest digest is included in the Merkle tree at the correct position based on the object's PVK. Finally, it verifies that the Merkle root hash matches the published checkpoint.

Although we maintain a separate digest chain for each object, the collective history of operations, across all objects, is also linearizable. This follows from the classical result that linearizability is a local property [HW90]. Thus, Ghostor provides *verifiable linearizability across all objects, while supporting full concurrency for operations on different objects*.

5.5.4 Concurrent Operations on a Single Object

As explained in Section 5.5.1, the client must fetch the latest digest from the server to construct a digest for a new GET or PUT. If two clients attempt to GET or PUT an object concurrently, they may retrieve the same latest digest for that object, and therefore construct new digests that both have the same $\text{Hash}_{\text{prev}}$. An honest server can only accept one of them; the other operation must be aborted. A naïve fix is for clients to acquire locks (or leases) on objects during network round trips, but this limits single-object throughput according to client round-trip times. How can we allow concurrent

operations on a single object without holding server-side locks during round trips? We explain our techniques at a high level below; the full version[HKP20] contains a full description of our protocol. GETs. We optimize GETs so that clients need not fetch the latest digest, obviating the need to lock for a round trip. When a client submits a GET request to the server, the client need not include $\text{Hash}_{\text{prev}}$, $\text{Hash}_{\text{data}}$, or $\text{Hash}_{\text{keylist}}$ in the digest presented to the server. The client includes the remaining fields and a signature over only those fields. Then, the server chooses the hashes for the client and returns the resulting digest, signed by the server. Although the server can replay operations, this is harmless because GETs do not affect data. When the verification daemon verifies a GET, it checks the client signature without including $\text{Hash}_{\text{prev}}$, $\text{Hash}_{\text{data}}$, or $\text{Hash}_{\text{keylist}}$.

PUTs. The above technique does not apply to PUTs, because the server can roll back objects by replaying PUTs. Simply using a client-provided nonce to detect replayed PUTs is not sufficient, because the server can delay incorporating a PUT (which we call a *time-stretch* attack) to manipulate the final object contents. For PUTs, Ghostor uses a two-phase protocol. In the PREPARE phase, the client operates in the same way as GET, but signs the digest with WSK; the server fills in the hashes, signs the resulting digest, appends it to the object’s digest chain, and returns it to the client. In the COMMIT phase, the client creates the final digest for the operation—omitting $\text{Hash}_{\text{prev}}$ and appending an additional field $\text{Hash}_{\text{prep}}$, which is the hash of the server-signed digest obtained in the PREPARE phase—and uploads it to the server with the new object contents. The server fills in $\text{Hash}_{\text{prev}}$ based on the object’s digest chain (which could have changed since the PREPARE phase), signs the resulting digest, appends it to the object’s digest chain, and returns it to the client. The server can replay PREPARE requests, but it does not affect object contents. The server cannot generate a COMMIT digest for a replayed PREPARE request, because the client signed the COMMIT digest including the hash of the server-signed PREPARE digest, which includes $\text{Hash}_{\text{prev}}$. The server can replay a COMMIT request for a particular PREPARE request, but this is harmless because of our conflict resolution strategy described below.

Resolving Conflicts. If two accesses are concurrent (i.e., neither commits before the other prepares), then linearizability does not require any particular ordering of those operations, only that all clients perceive the same ordering. If a GET is concurrent with a PUT (GET digest between the PREPARE and COMMIT digests for a PUT), Ghostor linearizes the GET as happening before the PUT. This allows the result of the GET to be served immediately, without waiting for the PUT to finish. For concurrent PUTs, it is unsafe for the linearization order to depend on the COMMIT digest, because the server could perform a time-stretch or replay attack on a COMMIT digest, to manipulate which PUT wins. Therefore, Ghostor chooses as the winning PUT the one whose PREPARE digest is latest. The server can still delay PREPARE digests, but the client can choose not to COMMIT if the delay is unacceptably large. To simplify the implementation of this conflict resolution procedure, we require that the PREPARE and COMMIT phases happen over the same session with the client, during which the server can keep in-memory state for the relevant object. This allows the server to match PREPARE and COMMIT digests without additional accesses to secondary storage.

Verification Complexity. To verify PUTs, the verification daemon must check that $\text{Hash}_{\text{data}}$ only changes on COMMIT digests for winning writes. Thus, it must keep track of all PREPARE digests since the latest PREPARE digest whose corresponding COMMIT has been seen. We can bound this state by requiring that PUT requests do not cross an epoch boundary.

ACL Updates. We envision that updates to the ACL will be rare, so our implementation does not allow **set_acl** operations to proceed concurrently with GETs or PUTs. It may be possible to apply a two-phase technique, similar to our concurrent PUT protocol, to allow **set_acl** operations to proceed concurrently with other operations. We leave exploring this to future work.

5.6 Mitigating Resource Abuse

To prevent resource abuse, commercial data-sharing systems, like Google Drive and Dropbox, enforce per-user resource quotas. Ghostor cannot do this, because Ghostor’s anonymity prevents it from tracking users. Instead, Ghostor uses two techniques to prevent resource abuse without tracking users: anonymous payments and proof of work.

5.6.1 Anonymous Payments

A strawman approach is for users to use an anonymous cryptocurrency (e.g., Zcash [Zcab]) to pay for each expensive operation (e.g., operations that consume storage). Unfortunately, this requires a separate blockchain transaction for each operation, limiting the system’s overall throughput.

Instead, Ghostor lets users pay for expensive operations *in bulk* via the **pay** API call (Section 5.2). The server responds with a set of *tokens* proportional to the amount paid via Zcash, which can later be redeemed *without using the blockchain* to perform operations. Done naïvely, this violates Ghostor’s anonymity; the server can track users by their tokens (tokens issued for a single **pay** call belong to the same user).

To circumvent this issue, Ghostor uses *blind signatures* [CPS94; Cha83; Cha84]. A Ghostor client generates a random token and *blinds* it. After verifying that the client has made a cryptocurrency payment, the server signs the blinded token. The blind signature protocol allows the client to *unblind* it while preserving the signature. To redeem the token, the client gives the unblinded signed token to the server, who can verify the server’s signature to be sure it is valid. The server cannot link tokens at the time of use to tokens at the time of issue because the tokens were blinded when the server originally signed them.

5.6.2 Proof of Work (PoW)

Another way to mitigate resource abuse is **proof of work (PoW)** [Bac02]. Before each request from the client, the server sends a random challenge to the client, and the client must find a proof such that $\text{Hash}(\text{challenge}, \text{proof}, \text{request}) < \text{diff}$. *diff* controls the difficulty, which is chosen to offset the amplification factor in the server’s work. Because of the guarantees of the hash function, the client must iterate through different proofs until it finds one that works. In contrast, the server efficiently checks the proof by computing one hash.

5.6.3 Anonymous Payments & PoW in Ghostor

Ghostor uses anonymous payments and PoW together to mitigate resource abuse. Our implementation requires anonymous payment only for **create_object**, which requires the server to commit additional storage space for the new object. This is analogous to systems like Google Drive or Dropbox, which require payment to increase a user’s storage limit but do not charge based on the count or frequency of object accesses. Implicit in this model are hard limits on object size and per-object access frequency, which Ghostor can enforce. Although our implementation requires payment only

for **create_object**, an alternate implementation may choose to require payment for every operation except **pay**. Ghostor requires PoW for all API calls. This includes **pay** and **create_object**, to offset the cost of Zcash payments and verifying blind signatures.

5.7 Applying Ghostor to Applications

In this section, we discuss two applications of Ghostor that we implemented: EHR Sharing and Ghostor-MH.

5.7.1 Case Study: EHR Sharing

Our goal in this section is to show how a real application may interface with Ghostor’s semantics (e.g., ownership, key management, error handling) and how Ghostor’s security guarantees might benefit a real application. To make the discussion concrete, we explore a particular use case: multi-institutional sharing of electronic health records (EHRs). It has been of increasing interest to put patients in control of their data as they move between different healthcare providers [GCL; Sha; Hop]. As it is paramount to protect medical data in the face of attackers [Dav], various proposals for multi-institutional EHR sharing use a blockchain for access control and integrity [Med; AEVL16]. Below, we explore how to design such a system using Ghostor to store EHRs in a central object store, using only decentralized trust. We also implemented the system for Open mHealth [Ope].

Each patient owns one or more objects in the central Ghostor system representing their EHRs. Each patient’s Ghostor client (on her laptop or phone) is responsible for storing the PSKs for these objects. The PSKs could be stored in a wristband, as in [Med], in case of emergency situations for at-risk patients. When the patient seeks treatment from a healthcare provider, she can grant the healthcare provider access to the objects containing the relevant information in Ghostor. Each healthcare provider’s Ghostor client maintains a local *metadata database*, mapping patient identities (object IDs, Section 5.2) to PVKs. This mapping could be created when a patient checks in to the office for the first time (e.g., by sharing a QR code).

Benefits. Existing proposals leverage a blockchain to achieve integrity guarantees [Med; AEVL16] but use the blockchain more heavily than Ghostor: for example, they require a blockchain transaction to grant access to a healthcare provider, which results in poor performance and scalability. Additionally, Ghostor provides anonymity for sharing records.

Epoch Time. An important aspect of Ghostor’s semantics is that one has to wait until the next epoch before one can verify that no fork has occurred. It is reasonable to fetch a patient’s record at the time that they check in to a healthcare facility, but before they are called in for treatment. This allows the time to wait until the end of an epoch to overlap with the patient’s waiting time. In the case of scheduled appointments, the record can be fetched in advance so that integrity can be verified by the time of the appointment. An epoch time of 15–30 minutes would probably be sufficient.

Error Handling. If a healthcare provider detects a fork when verifying an epoch, it informs other healthcare providers of the integrity violation out-of-band of the Ghostor system. Ghostor does not constrain what happens next. One approach, used in Certificate Transparency (CT), is to abandon the Ghostor server for which the integrity violation was detected. We envision that there would be a few Ghostor servers in the system, similar to logs in CT, so this would require affected users to migrate their data to a new server. Another approach is to handle the error in the same way that blockchain-based systems [Med; AEVL16] handle cases where the hash on the blockchain does not match the hash of the data—treat it as an availability error. While neither solution is ideal,

it is better than the status quo, in which a malicious adversary is free to perform fork or rollback attacks undetected, causing patients to receive incorrect treatments based on old or incorrect data, potentially resulting in serious physical injury.

5.7.2 A Metadata-Hiding Data-Sharing Scheme

Ghostor’s anonymity techniques can be combined with a globally oblivious scheme, AnonRAM [BHKP16], to obtain a *metadata-hiding* object-sharing scheme, *Ghostor-MH*. Ghostor-MH is *not* a practical system, but only a theoretical scheme; our goal is to show that Ghostor’s techniques are complementary to and compatible with those in globally oblivious schemes. We apply Ghostor’s techniques in Ghostor-MH as follows. First, we apply Ghostor’s principle of switching from a user-centric to a data-centric design. Whereas each ORAM instance in AnonRAM corresponds to a user, each ORAM instance in Ghostor-MH corresponds to an *object group*, a fixed-sized set of objects with a shared ACL. Second, we apply the design of Ghostor’s object header in Ghostor-MH. This is accomplished by storing the ORAM secret state, encrypted, on the server. Finally, we use similar techniques to mitigate resource abuse in Ghostor-MH as we do in Ghostor.

In Section 5.7.2.2 below, we provide a more in-depth explanation of Ghostor-MH. We first provide more details about AnonRAM in Section 5.7.2.1. This is necessary because, as explained in Section 5.7.2, we construct Ghostor-MH by applying Ghostor’s techniques to AnonRAM [BHKP16].

5.7.2.1 Overview of AnonRAM

ORAM [GO96] is a technique to access objects on a remote server without revealing which objects are accessed. Many ORAM schemes, such as Path ORAM [Ste+13], allow a *single user* to access data. Path ORAM [Ste+13] works by having the client shuffle a small amount of server-side data with each access, such that the server cannot link requests to the same object. Clients store mutable *secret state*, including a stash and position map, used to find objects after shuffling.

AnonRAM extends single-user ORAM to support *multiple users*. Each AnonRAM user essentially has her own ORAM on the server. When a user accesses an object, she (1) performs the access as normal in her own ORAM, and (2) performs a *fake access* to all of the other users’ ORAMs. To the server, the fake accesses are indistinguishable from genuine accesses, so the server does not learn to which ORAM the user’s object belongs. This, together with each individual ORAM hiding which of its objects was accessed, results in global obliviousness across all objects in all ORAMs.

To support fake accesses, *re-randomizable* public-key encryption (e.g., El Gamal) is used to encrypt objects in each ORAM. To guard against malicious clients, the server requires a zero-knowledge proof with each real or fake access, to prove that *either* (1) the client knows the secret key for the ORAM, *or* (2) the new ciphertexts encrypt the same data as existing ciphertexts (i.e., they were re-randomized correctly).

A limitation of AnonRAM is that there is *no object sharing among users*; each user can access only the objects she owns. Furthermore, AnonRAM and similar schemes (Section 5.10) are *theoretical*—they consider oblivious storage from a cryptographic standpoint, but do not consider challenges like payment, user accounts, and resource abuse.

5.7.2.2 Ghostor-MH

Recall from Section 5.7.2 that we apply to AnonRAM Ghostor’s principle of switching from a user-centric to a data-centric design. Each ORAM now corresponds to an *object group*, which is a fixed-size set of objects with a shared ACL. Each object group has one object header and one digest chain.

Ghostor-MH uses Path ORAM, which organizes server-side storage as a binary tree. To guard against a malicious adversary controlling the server, we build a Merkle tree over the binary tree, and compute $\text{Hash}_{\text{data}}$ in each digest as the hash of the Merkle root and ORAM secret state. This allows each client to efficiently compute the new $\text{Hash}_{\text{data}}$ after each ORAM access, without downloading the entire ORAM tree. The ORAM secret state is stored on the server, encrypted with OSK, so multiple clients can access an object group. This is analogous to Ghostor’s object header, which stores an object’s keys encrypted on the server.

To access an object, a client (1) identifies the object group containing it, (2) downloads the object header and encrypted ORAM secret state, (3) obtains OSK from the object header, (4) decrypts the ORAM secret state, (5) uses it to perform the ORAM access, (6) encrypts and uploads the new ORAM secret state, (7) computes a new digest for the operation, (8) has the server sign it, and (9) sends it to the verification daemon. For all other object groups, the client performs a *fake access* that fetches data from the server and generates a digest, but only re-randomizes ciphertexts instead of performing a real access. This hides which object group contains the object. When writing an object, the client pads it to a maximum size (the ORAM block size) to hide the length of the object.

Below, we explain some more details about Ghostor-MH:

Fake accesses. OSK is replaced with an El Gamal keypair. This allows ciphertexts in the ORAM tree and the ORAM secret state to be re-randomized. We no longer attach a client signature to each digest, but instead modify the zero-knowledge proof in AnonRAM to prove that *either* the client can produce a signature over the digest with WSK, *or* the ciphertexts were properly re-randomized.

Hiding timing. Similar to secure messaging systems [HLZZ15a], Ghostor-MH operates in rounds (shorter than epochs) to hide timing. In each round, each client either accesses an object as described above, or performs a fake access on all ORAMs if there is no pending object access. Each client chooses a random time during the round to make its request to the server.

Using tokens. In a globally oblivious system like Ghostor-MH, it is impossible to enforce the per-object quotas discussed in Section 5.6.3. Thus, it is advisable to require users to expend tokens for *all* operations (except **pay**), not just **create_object**. Our PoW mechanism applies to Ghostor-MH unchanged.

Object group creation. The server can distinguish payment (to obtain tokens) and object group creation from GET/PUT operations. The most secure solution is to have a setup phase to create all object groups and perform all payment in advance. Barring this, we propose adding a special round at the start of each epoch, used only for creation and payment; all object accesses during an epoch happen after this special round.

List of object groups. To make fake accesses, each client must know the full list of object groups. To ensure this, we can add an additional digest chain to keep track of all created object groups, checkpointed every epoch with the rest of the system.

Changing permissions. In our solution so far, the server can distinguish a `set_acl` operation from object accesses. To fix this, we require the owner of each object group to perform exactly one `set_acl` for that object group during each epoch; if he does not wish to change it, he sets it to the same value.

Concurrency. When a client iterates over all ORAMs to make accesses (fake or real), the client locks each ORAM individually and releases it after the access. No “global lock” is held while a client makes fake accesses to all ORAMs.

5.8 Implementation

We implemented a prototype of Ghostor in Go. It consists of three parts, as in Figure 5.4, server (≈ 2100 LOC), client library (≈ 1000 LOC), and verification daemon (≈ 1000 LOC), which all depend on a set of core Ghostor libraries (≈ 1400 LOC).

Our implementation uses Ceph RADOS [WBMLM06] for consistent, distributed object storage. We use SHA-256 for the cryptographic hash and the NaCl secretbox library (which uses XSalsa20 and Poly1305) for authenticated symmetric-key encryption. For *key-private* asymmetric encryption (to encrypt signing keys in the object header), we implemented the El Gamal cryptosystem, which is *key-private* [BBDP01], on top of the Curve25519 elliptic curve. We use an existing blind signature implementation [Rsa] based on RSA with 2048-bit keys and 1536-bit hashes. We use Ed25519 for digital signatures.

As discussed in Section 5.3, Ghostor uses external systems for anonymous communication and payment. In our implementation, clients use Tor [DMS04a] to communicate with the server and Zcash 1.0.15 for anonymous payments. We build a Zcash test network, separate from the Zcash main network. Ghostor, however, could also be deployed on the Zcash main chain. Zcash is also used as the blockchain to post checkpoints. Our implementation runs as a *single* Ghostor server that stores its data in a scalable, fault-tolerant, distributed storage cluster. We discuss how to scale to *multiple* servers in the full version[HKP20].

We implemented a proof of concept of our theoretical scheme Ghostor-MH (Section 5.7.2), in ≈ 2100 additional LOC. As it is a theoretical scheme, our focus in evaluating Ghostor-MH is simply to understand the latency of operations.

5.9 Evaluation

We run experiments on Amazon EC2. Except in Section 5.9.3 and Section 5.9.5, Ghostor’s storage cluster consists of three `i3en.xlarge` servers. We configure Ceph to replicate each object (key-value pair) on two SSDs on different machines, for fault-tolerance.

5.9.1 Microbenchmarks

Basic Crypto Primitives. We measured the latency of crypto operations used in Ghostor’s critical path. En/decryption of object contents varies linearly with the object size, and takes ≈ 2 ms for 1 MiB. Key-private en/decryption for object headers and signing/verification of digests takes less than 150 us.

Blind Signatures. We also measure the blind signature scheme used for object creation, which consists of four steps. (1) The client *generates* a blinded hash of a random number. (2) The server *signs* the blinded hash. (3) The client *unblinds* the signature, obtaining the server’s signature over the original number. (4) The server *verifies* the signature and the number during object creation. Results are shown in Fig. 5.6.

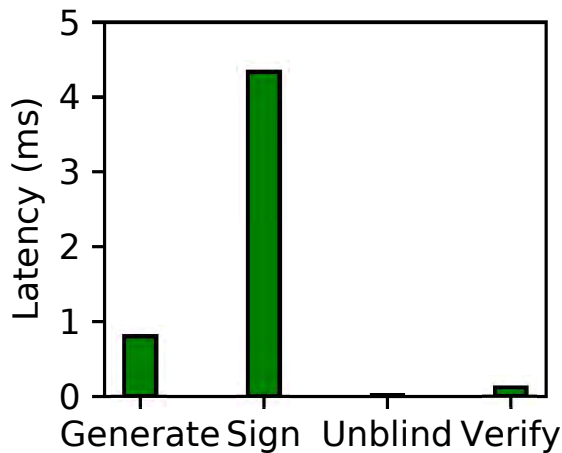


Figure 5.6: Blind signature

A	50% R, 50% W
B	95% R, 5% W
C	100% R
D	95% R, 5% Insert
E	95% R, 5% Range
F	50% R, 50% R-Modify-W

Figure 5.7: YCSB workloads (R: read, W: write)

Verification Procedure. In Figure 5.8, we measure the overhead of verification for digests in a single epoch. For client verification time, we perform an end-to-end test, measuring the total time to fetch digests and to verify them. The client has 1,000 signed digests for operations the client performed during the epoch that the client needs to check were included in the history of digests. We vary the total number of digests in the object’s history for that epoch. The reported values in Figure 5.8a are the total time to verify the object, divided by the total number of operations on the object, indicating the verification time *per digest*. The trend indicates a constant overhead when the

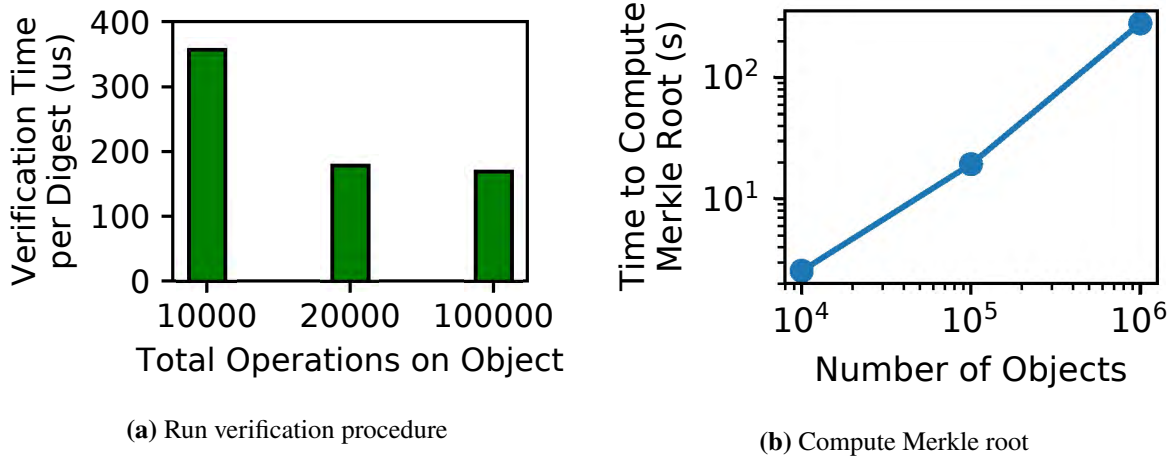


Figure 5.8: Operations for verification

total number of operations on the object is small, that is amortized when the number of operations is large.

Figure 5.8b shows the server’s overhead to compute the Merkle root. We inserted objects using YCSB (Section 5.9.2.2) during an epoch, and measured the time to compute the Merkle root at the end of that epoch. For 10,000 objects, this takes about 2.5 seconds; for 1,000,000 objects, it takes about 280 seconds. Reading the latest digest for each object (leaves of the Merkle tree) dominates the time to compute the Merkle root (2 seconds for 10,000 objects, 272 seconds for 1,000,000 objects). The reason is that our on-disk data structures are optimized for single-object operations, which are in the critical path. In particular, each object’s digest chain is stored as a separate batched linked list, so reading the latest digests requires a separate read for each object.

5.9.2 Server-Side Overhead

This section measures to what extent anonymity and VerLinear affect Ghostor’s performance. To ensure that the bottleneck was on the server, we set proof of work to minimum difficulty and do not use anonymous communication (Section 5.3), but we return to evaluating these in Section 5.9.3.

We measure the end-to-end performance of operations in Ghostor, both as a whole and for instantiations of Ghostor having only anonymity or VerLinear. We compare these to an insecure baseline as well as to competitive solutions for privacy and verifiable consistency, as we now describe.

1. *Insecure system (“Insec”).* This system uses the traditional ACL-based approach for serving objects. Each object access is preceded by a read to the object’s ACL to verify that the user has permission to access the object. Similarly, creating an object requires a read to a per-user account file. It provides no security against a compromised server.

2. *End-to-End Encrypted system (“E2EE”).* This system encrypts objects placed on the server using

end-to-end encryption similarly to SiRiUS [GSMB03a]. Such systems have an encrypted KeyList similar to Ghostor's, but clients can cache their keys locally on most accesses unlike Ghostor.

3. *Ghostor's anonymity system ("Anon")*. This is Ghostor with VerLinear disabled. This fits a scenario where one wants to hide information from a *passive* server attacker. Unlike the E2EE system above, this system cannot cache keys locally—every operation incurs an additional round trip to fetch the KeyList from the server. In addition, every operation incurs yet another round trip at the beginning for the client to perform a proof of work. On the positive side, the server does not maintain any per-user ACL.

4. *Fork Consistent system ("ForkC")*. This system maintains Ghostor's digest chain (Section 5.5.1), but does not post checkpoints. Each operation appends to a per-object log of digests, using the techniques in Section 5.5.4. This system also performs an ACL check when creating an object.

5. *Ghostor's VerLinear system ("VLinear")*. This system corresponds to the VerLinear mechanism in Section 5.5 (including Section 5.5.2). This matches a use case where one wants integrity, but does not care about privacy. We do not include the verification procedure, already evaluated in Section 5.9.1.

6. *Ghostor*. This system achieves both anonymity and VerLinear, and therefore incurs the costs of both guarantees.

5.9.2.1 Object Accesses

In each setup, we measured the latency for create, GET, and PUT operations (Figure 5.9a), throughput for GETs/PUTs to a single object (Figure 5.10a), and the throughput for creating objects and for GETs/PUTs to multiple objects (Figure 5.10b).

Fork consistency adds substantial overhead, because additional accesses to persistent storage are required for each operation, to maintain each object's log of digests. Ghostor, which both maintains a per-object log of digests and provides anonymity, incurs additional overhead because clients do not cache keys, requiring the server to fetch the header for each operation. In contrast, for Anon, the additional cost of reading the header is offset by the lack of ACL check. For 1 MiB objects, en/decryption adds a visible overhead to latency.

End-to-end encryption adds little overhead to throughput; this is because we are measuring throughput at the *server*, whereas encryption and decryption are performed by *clients*. The only factor affecting server performance is that the ciphertexts are 40 bytes larger than plaintexts.

Single-object throughput is lower for ForkC, VLinear, and Ghostor, because maintaining a digest chain requires requests to be serialized across multiple accesses to persistent storage. In contrast, Insec, E2EE, and Anon serve requests in parallel, relying on Ceph's internal concurrency control.

In the multi-object experiments, in which no two concurrent requests operate on the same object, this bottleneck disappears. For small objects, throughput drops in approximately an inverse pattern to the latency, as expected. For large objects, however, all systems perform commensurately. This is likely because reading/writing the object itself dominated the throughput usage for these experiments, without any concurrency overhead at the object level to differentiate the setups.

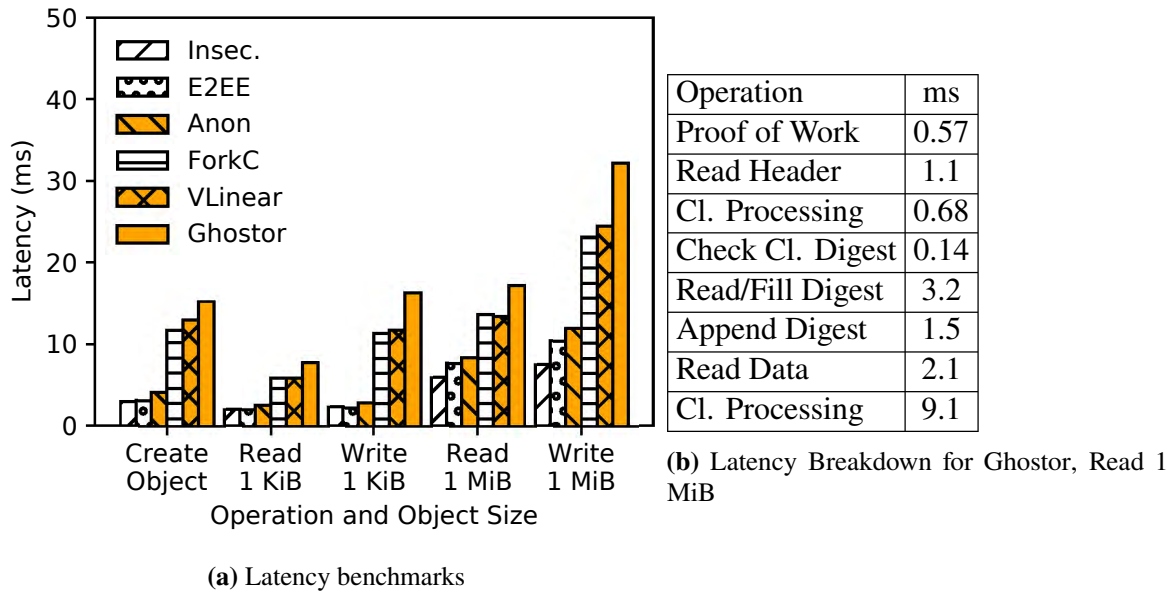


Figure 5.9: Latency measurements

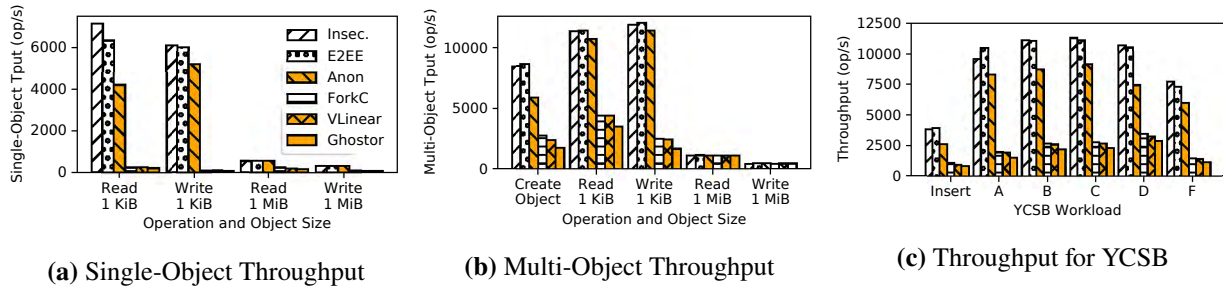


Figure 5.10: Benchmarks comparing throughput of the six setups described in Section 5.9.2

5.9.2.2 Yahoo! Cloud Serving Benchmark

In this section, we evaluate our system using the Yahoo! Cloud Serving Benchmark (YCSB). YCSB provides different workloads representative of various use cases, summarized in Figure 5.7. We do not use Workload E because it involves range queries, which Ghostor does not support. As shown in Figure 5.10c, anonymity incurs up to a 25% overhead for benchmarks containing insertions, owing to the additional accesses to storage required to store used object creation tokens. However, it shows essentially no overhead for GETs and PUTs. Fork consistency adds a 3–4x overhead compared to the Insec baseline. VerLinear adds essentially no overhead on top of fork consistency; this is to be expected, because the overhead of VerLinear is outside of the critical path (except for insertions, where the overhead is easily amortized). Ghostor, which provides both anonymity and VerLinear, must forgo client-side caching, and therefore incurs additional overhead, with a 4–5x throughput

reduction overall compared to the Insec baseline.

5.9.3 End-to-End Latency

We analyze Ghostor’s performance from the client’s perspective, including PoW and anonymous communication (Section 5.3). In these experiments, we use three `m4.10xlarge` instances each with three `gp2` SSDs for Ghostor’s storage cluster.

5.9.3.1 Microbenchmarks

The latency experienced by a Ghostor client is the latency measured in Figure 5.9, plus the additional overhead due to the proof of work mechanism and anonymous communication. The difficulty of the proof of work problem is adjustable. For the purpose of evaluation, we set it to a realistic value to prevent denial of service. Figure 5.9b indicates that it takes ≈ 32 ms for a Ghostor operation; therefore, we set the proof of work difficulty such that it takes the client, on average, 100 times longer to solve (≈ 3.2 s). Figure 5.11 shows the distribution of latency for the client to solve the proof of work problem. As expected, the distribution appears to be memoryless.

In our implementation, a client connects to a Ghostor server by establishing a circuit through the Tor [DMS04a] network. The performance of the connection, in terms of both latency and throughput, varies according to the circuit used. Figure 5.11 shows the distribution of (1) circuit establishment time, (2) round-trip time, and (3) network bandwidth. We used a fresh Tor circuit for each measurement. Based on our measurements, a Tor circuit usually provides a round-trip time less than 1 second and bandwidth of at least 2 Mb/s.

5.9.3.2 Macrobenchmarks

We now measure the end-to-end latency of each operation in Ghostor’s client API (Section 5.2), including all overheads experienced by the client. As explained in Section 5.9.3.1, the overhead due to proof of work and Tor is quite variable; therefore, we repeat each experiment 1000 times, using a separate Tor circuit each time, and report the distribution of latencies for each operation in Figure 5.13. Comparing Figure 5.13 to Figure 5.9, the client-side latency is dominated by the cost of PoW and Tor; Ghostor’s core techniques in Figure 5.9 have relatively small latency overhead. For the pay operation, we measure only the time to redeem a Zcash payment for a single token, not the time for proof of work or making the Zcash payment (see Section 5.9.4 for a discussion of this overhead). GET and PUT for large objects are the slowest, because Tor network bandwidth becomes a bottleneck. The `create_user` operation (not shown in Figure 5.13) is only 132 microseconds, because it generates an El Gamal keypair locally without any interaction with the server.

5.9.4 Zcash

In our implementation, we build our own Zcash test network to avoid the expense from Zcash’s main network. Since our system leverages Zcash in a minimal way, the overhead of Zcash is not on

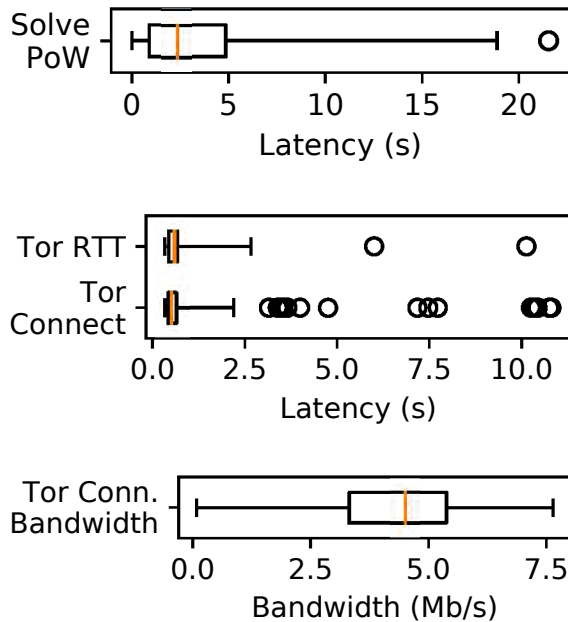
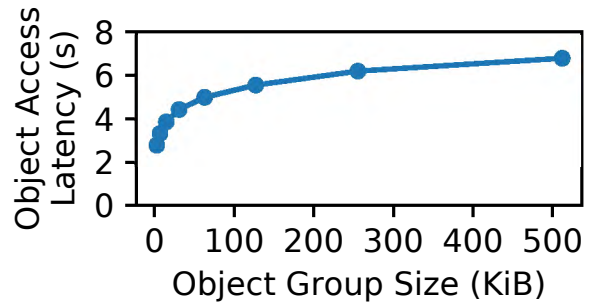
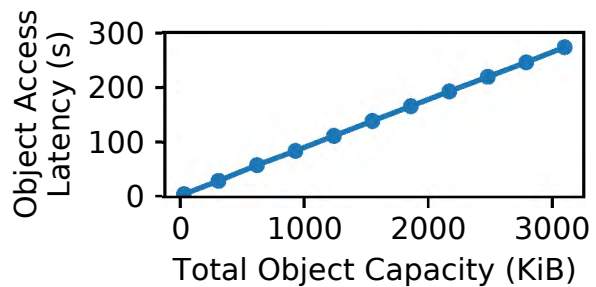


Figure 5.11: Microbenchmarks of PoW mechanism and Tor



(a) Latency vs. Object Group Size



(b) Latency vs. No. Object Groups

Figure 5.12: Ghostor-MH

the critical path of our protocol. According to the Zcash website [Zcab] and block explorer [Bitb], the block size limit is about 2 MiB, and block interval is about 2.5 minutes. In the past six months, the maximum block size has been less than 150 KiB and the average transaction fee has been much less than 0.001 ZEC (0.05 USD at the time of writing). Hence, even with shorter epochs (less time for misbehavior detection), the price of Ghostor’s checkpoints is modest since there is a single checkpoint per epoch for the whole system.

5.9.5 Ghostor-MH

For completeness, we evaluate the *theoretical* Ghostor-MH scheme presented in Section 5.7.2 using the EC2 setup from Section 5.9.3, focusing only on the latency of accessing an object. We do not use Tor and we set the PoW difficulty to minimum. Latency is dominated by en/decryption on the client, because object contents and ORAM state are encrypted with El Gamal encryption, which is much slower than symmetric-key encryption. Figure 5.12a shows the object access latency for an object group, as we vary its size. It scales logarithmically, as expected from Path ORAM. An additional overhead of ≈ 2 s comes from re-encrypting ORAM client state (32 KiB, after padding and encryption) on each access. Figure 5.12b shows the object access latency as we vary the number of object groups (each object group is 31 KiB). It scales linearly, because the client makes fake

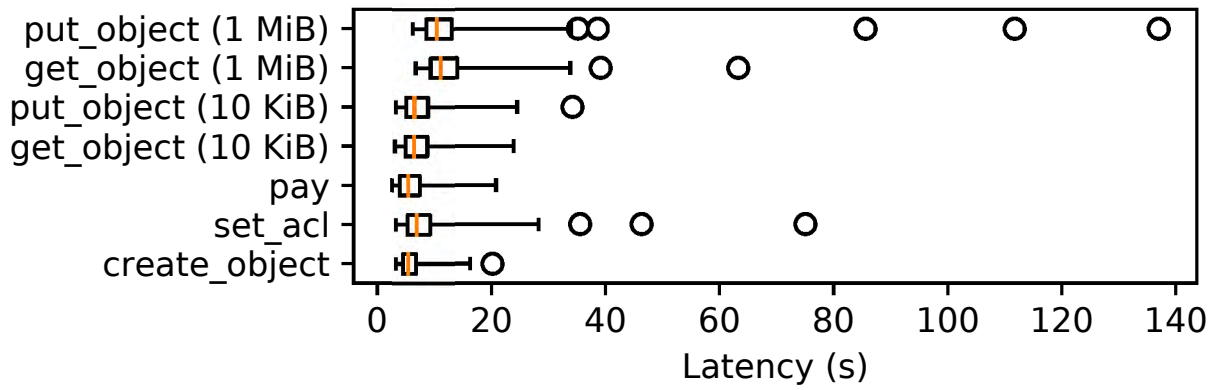


Figure 5.13: End-to-end latencies of client-side operations

accesses to *all* other object groups to hide which one it truly accessed. Latency could potentially be improved by using multiple client CPU cores.

5.10 Related Work

Systems Providing Consistency. We have already compared extensively with SUNDR [LKMS04]. Venus [SCCKMS10] achieves eventual consistency; however, Venus requires some clients to be frequently online and is vulnerable to malicious clients. Caelus [KL15] has a similar requirement and does not resist collusion of malicious clients and the server. Verena [KFPC16] trusts one of two servers. SPORC [FZFF10], which combines fork consistency with operational transformation, allows clients to recover from a fork attack, but does not resist faulty clients. Depot [Mah+11] can tolerate faulty clients, but achieves a weaker notion of consistency than VerLinear. Furthermore, its consistency techniques are at odds with anonymity. Ghostor and these systems use hash chains [HS90; MB02] as a key building block.

Systems Providing E2EE. Many systems provide end-to-end encryption (E2EE), but leak significant user information as discussed in Section 5.3.3: academic systems such as Persona [BBSBS09], DEPSKY [BCQAS13], CFS [Bla93], SiRiUS [GSMB03a], Plutus [KRWF03], ShadowCrypt [HA-JSS14], M-Aegis [LCSJLB14], Mylar [Pop+14] and Sieve [WMZV16] or industrial systems such as Crypho [Cry], Tresorit [Inc], Keybase [Key], PreVeil [Pre], Privly [Pri] and Virtru [Vir].

Systems Using Trusted Hardware. Some systems, such as Haven [BPH15] and A-SKY [CVPPFR19], protect against a malicious server by using trusted hardware. Existing trusted hardware, like Intel SGX, however, suffer from side-channel attacks [VB+18].

Oblivious Systems. A complementary line of work to Ghostor aims to hide access patterns: *which* object was accessed. Standard Oblivious RAM (ORAM) [GO96; SCSL11; WCS15] works in the single-client setting. Multi-client ORAM [BHKP16; HOWW19a; KPK17; MMRS17; MMRS15; SZEALT16; SS13] extends ORAM to support multiple clients. These works either rely on central trust [SZEALT16; SS13] (either a fully trusted proxy or fully trusted clients) or provide limited functionality (not providing global object *sharing* [BHKP16], or revealing user identities [MMRS17]). GORAM [MMRS15] assumes the adversary controlling the server does not collude with clients. Furthermore, it only provides obliviousness within a single data owner’s objects, not *global obliviousness* across all data owners.

AnonRAM [BHKP16] and PANDA [HOWW19a] provide global obliviousness and hide user identity, but are slow. They do not provide for sharing objects or mitigate resource abuse. One can realize these features by applying Ghostor’s techniques to these schemes, as we did in Section 5.7.2 to build Ghostor-MH. Unlike these schemes, Ghostor-MH is a *metadata-hiding object-sharing scheme* providing both global obliviousness and anonymity without trusted parties or non-collusion assumptions.

Decentralized Storage. Peer-to-peer storage systems, like OceanStore [Kub+00], Pastry [RD01], CAN [RFHKS01], and IPFS [Ben14], allow users to store objects on globally distributed, untrusted storage without any coordinating central trusted party. These systems are vulnerable to rollback/fork attacks on mutable data by malicious storage nodes (unlike Ghostor’s VerLinear). While some of them encrypt objects for privacy, they do not provide a mechanism to distribute secret keys while preserving anonymity, as Ghostor does. Recent blockchain-based decentralized storage systems, like Storj [Sto], Swarm [TFJ16], Filecoin [Fil], and Sia [Sia], have similar shortcomings.

Decentralized Trust. As discussed in Section 5.1, blockchain systems [CL99; YMRGA19; Nak08;

But+13] and verifiable ledgers [LLK13a; MBBFF15] can serve as the source of decentralized trust in Ghostor.

Another line of work aims to provide efficient auditing mechanisms. EthIKS [Bon16] leverages smart contracts [But+13] to monitor key transparency systems [MBBFF15]. Catena [TD17] builds log systems based on Bitcoin transactions, which enables efficient auditing by low-power clients. It may be possible to apply techniques from those works to optimize our verification procedure in Section 5.5.2. However, none of them aim to build secure data-sharing systems like Ghostor.

Secure Messaging. Secure messaging systems [CGBM15; TGLZZ17a; HLZZ15a] hide network traffic patterns, but they do not support object storage/sharing as in our setting. Ghostor can complementarily use them for its anonymous communication network.

Chapter 6

JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT

As the Internet of Things (IoT) emerges over the next decade, developing secure communication for IoT devices is of paramount importance. Achieving end-to-end encryption for large-scale IoT systems, like smart buildings or smart cities, is challenging because multiple principals typically interact *indirectly* via intermediaries, meaning that the recipient of a message is not known in advance. This paper proposes JEDI (**J**oining **E**ncryption and **D**elegation for **I**oT), a many-to-many end-to-end encryption protocol for IoT. JEDI encrypts and signs messages end-to-end, while conforming to the decoupled communication model typical of IoT systems. JEDI's keys support expiry and fine-grained access to data, common in IoT. Furthermore, JEDI allows principals to delegate their keys, restricted in expiry or scope, to other principals, thereby granting access to data and managing access control in a scalable, distributed way. Through careful protocol design and implementation, JEDI can run across the spectrum of IoT devices, including ultra low-power deeply embedded sensors severely constrained in CPU, memory, and energy consumption. We apply JEDI to an existing IoT messaging system and demonstrate that its overhead is modest.

This work was previously published in [KHAPC19].

6.1 Introduction

As the Internet of Things (IoT) has emerged over the past decade, smart devices have become increasingly common. This trend is only expected to continue, with tens of billions of new IoT devices deployed over the next few years [Cis14]. The IoT vision requires these devices to communicate to discover and use the resources and data provided by one another. Yet, these devices collect privacy-sensitive information about users. A natural step to secure privacy-sensitive data is to use *end-to-end encryption* to protect it during transit.

Existing protocols for end-to-end encryption, such as SSL/TLS and TextSecure [FMBBSH16], focus on *one-to-one* communication between two principals: for example, Alice sends a message to Bob over an insecure channel. Such protocols, however, appear not to be a good fit for large-scale industrial IoT systems. Such IoT systems demand *many-to-many* communication among **decoupled** senders and receivers, and require **decentralized delegation of access** to enforce which devices can communicate with which others.

We investigate existing IoT systems, which currently do not encrypt data end-to-end, to understand the requirements on an end-to-end encryption protocol like JEDI. We use *smart cities* as an example application area, and data-collecting sensors in a large organization as a concrete use case. We identify three central requirements, which we treat in turn below:

▷ **Decoupled senders and receivers.** IoT-scale systems could consist of thousands of principals, making it infeasible for consumers of data (e.g., applications) to maintain a separate session with each producer of data (e.g., sensors). Instead, senders are typically **decoupled** from receivers. Such decoupling is common in *publish-subscribe* systems for IoT, such as MQTT, AMQP, XMPP, and Solace [Sol]. In particular, many-to-many communication based on publish-subscribe is the *de-facto* standard in smart buildings, used in systems like BOSS [DH+13], VOLTTRON [Vol], Brume [MRK18] and bw2 [AKCCK17], and adopted commercially in AllJoyn and IoTivity. Senders publish messages by addressing them to *resources* and sending them to a *router*. Recipients *subscribe* to a resource by asking the router to send them messages addressed to that resource.

Many systems for smart buildings/cities, like sMAP [DHJTOC10], SensorAct [ABCSSS12], bw2 [AKCCK17], VOLTTRON [Vol], and BAS [KFKC12], organize resources as a **hierarchy**. A resource hierarchy matches the organization of IoT devices: for instance, smart cities contain buildings, which contain floors, which contain rooms, which contain sensors, which produce streams of readings. We represent each resource—a leaf in the hierarchy—as a Uniform Resource Indicator (**URI**), which is like a file path. For example, a sensor that measures temperature and humidity might send its readings to the two URIs `buildingA/floor2/roomLHall/sensor0/temp` and `buildingA/floor2/roomLHall/sensor0/hum`. A user can subscribe to a URI prefix, such as `buildingA/floor2/roomLHall/*`, which represents a subtree of the hierarchy. He would then receive all sensor readings in room “LHall.”

¹Image credits: <https://tweakers.net/pricewatch/1275475/asus-f5401a-dm1201t.html>, <https://www.lg.com/uk/mobile-phones/lg-H791>, <https://www.bestbuy.com/site/nest-learning-thermostat-3rd-generation-stainless-steel/4346501.p?skuId=4346501>, <https://www.macys.com/shop/product/fitbit-charge-2-heart-rate-fitness-wristband?ID=2999458>

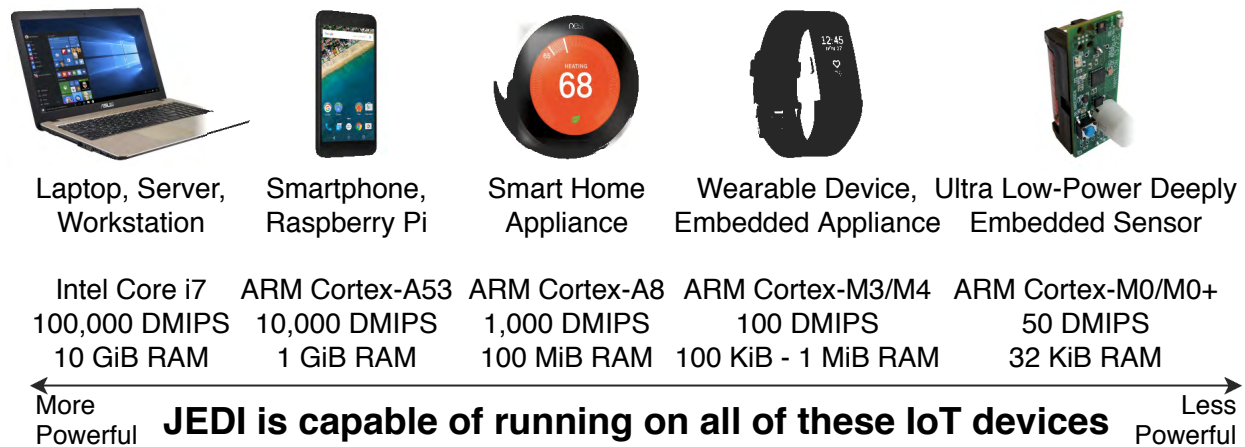


Figure 6.1: IoT comprises a diverse set of devices, which span more than four orders of magnitude of computing power (estimated in Dhrystone MIPS).¹

▷ **Decentralized delegation.** Access control in IoT needs to be fine-grained. For example, if Bob has an app that needs access to temperature readings from a single sensor, that app should receive the decryption key for only that one URI, even if Bob has keys for the entire room. In an IoT-scale system, it is not scalable for a central authority to individually give fine-grained decryption keys to each person’s devices. Moreover, as we discuss in Section 6.2, such an approach would pose increased security and privacy risks. Instead, Bob, who himself has access to readings for the entire room, should be able to delegate temperature-readings access to the app. Generally, a principal with access to a set of resources can give another principal access to a subset of those resources.

Vanadium [TS16] and bw2 [AKCCK17] introduced *decentralized delegation* (SPKI/SDSI [CEEFMR01] and Macaroons [BPETVL14]) in the smart buildings space. Since then, decentralized delegation has become the state-of-the-art for access control in smart buildings, especially those geared toward large-scale commercial buildings or organizations [FC15; HK18]. In these systems, a principal can access a resource if there exists a *chain* of delegations, from the owner of the resource to that principal, granting access. At each link in the chain, the extent of access may be qualified by *caveats*, which add restrictions to which resources can be accessed and when. While these systems provide delegation of permissions, they do not provide protocols for encrypting and decrypting messages end-to-end.

▷ **Resource constraints.** IoT devices vary greatly in their capabilities, as shown in Figure 6.1. This includes devices constrained in CPU, memory, and energy, such as wearable devices and low-cost environmental sensors.

In smart buildings/cities, one application of interest is *indoor environmental sensing*. Sensors that measure temperature, humidity, or occupancy may be deployed in a building; such sensors are *battery-powered* and transmit readings using a *low-power* wireless network. To see ubiquitous deployment, they must cost only *tens of dollars* per unit and have *several years* of battery life. To achieve this price/power point, sensor platforms are heavily resource-constrained, with mere *kilobytes* of memory (farthest right in Figure 6.1) [Ham; Par; Fel09; LLLMSL14; BMPR14; AKC17;

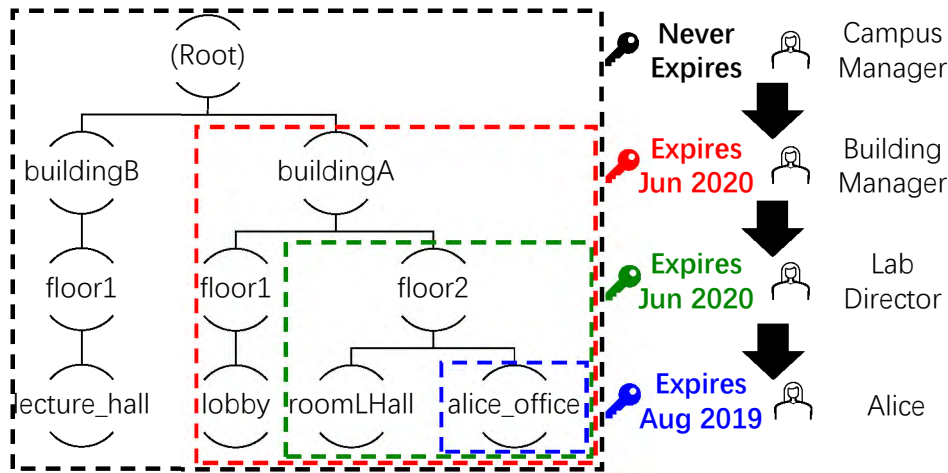


Figure 6.2: JEDI keys can be qualified and delegated, supporting decentralized, cryptographically-enforced access control via key delegation. Each person has a decryption key for the indicated resource subtree that is valid until the indicated expiry time. Black arrows denote delegation.

AFC16]. The *power consumption* of encryption is a serious challenge, even more so than its latency on a slower CPU; the CPU and radio must be used sparingly to avoid consuming energy too quickly [YHE02; Kim+18]. For example, on the sensor platform used in our evaluation, an average CPU utilization of merely 5% would result in less than a year of battery life, *even if the power cost of using the transducers and network were zero.*

6.1.1 Overview of JEDI

This paper presents JEDI, a *many-to-many* end-to-end encryption protocol compatible with the above three requirements of IoT systems. JEDI encrypts messages end-to-end for confidentiality, signs them for integrity while preserving anonymity, and supports delegation with caveats, all while allowing senders and receivers to be decoupled via a resource hierarchy. JEDI differs from existing encryption protocols like SSL/TLS, requiring us to overcome a number of *challenges*:

1. Formulating a new system model for end-to-end encryption to support **decoupled senders and receivers** and **decentralized delegation** typical of IoT systems (Section 6.1.1.1)
2. Realizing this expressive model while working within the **resource constraints** of IoT devices (Section 6.1.1.2)
3. Allowing receivers to verify the integrity of messages, while preserving the anonymity of senders (Section 6.1.1.3)
4. Extending JEDI’s model to support revocation (Section 6.1.1.4)

Below, we explain how we address each of these challenges.

6.1.1.1 JEDI's System Model (Section 6.2)

Participants in JEDI are called *principals*. Any principal can create a **resource hierarchy** to represent some resources that it owns. Because that principal owns all of the resources in the hierarchy, it is called the *authority* of that hierarchy.

Due to the setting of **decoupled senders and receivers**, the sender can no longer encrypt messages with the receiver's public key, as in traditional end-to-end encryption. Instead, JEDI models principals as interacting with resources, rather than with other principals. Herein lies the key difference between JEDI's model and other end-to-end encryption protocols: the publisher of a message encrypts it according to the URI to which it is published, not the recipients subscribed to that URI. Only principals permitted to subscribe to a URI are given keys that can decrypt messages published to that URI.

IoT systems that support **decentralized delegation** (Vanadium, bw2), as well as related non-IoT authorization systems (e.g., SPKI/SDSI [CEEFMR01] and Macaroons [BPETVL14]) provide principals with tokens (e.g., certificate chains) that they can present to prove they have access to a certain resource. Providing tokens, however, is not enough for end-to-end encryption; unlike these systems, JEDI allows *decryption keys* to be distributed via chains of delegations. Furthermore, the URI prefix and expiry time associated with each JEDI key can be restricted at each delegation. For example, as shown in Figure 6.2, suppose Alice, who works in a research lab, needs access to sensor readings in her office. In the past, the campus facilities manager, who is the authority for the hierarchy, granted a key for `buildingA/*` to the building manager, who granted a key for `buildingA/floor2/*` to the lab director. Now, Alice can obtain the key for `buildingA/floor2/alice_office/*` directly from her local authority (the lab director).

6.1.1.2 Encryption with URIs and Expiry (Section 6.3)

JEDI supports *decoupled* communication. The resource to which a message is published acts as a *rendezvous point* between the senders and receivers, used by the underlying system to route messages. Central to JEDI is the challenge of finding an analogous *cryptographic rendezvous point* that senders can use to encrypt messages without knowledge of receivers. A number of IoT systems [SBDHR18; PSWCT01] use only simple cryptography like AES, SHA2, and ECDSA, but these primitives are not expressive enough to encode JEDI's rendezvous point, which must support hierarchically-structured resources, non-interactive expiry, and decentralized delegation.

Existing systems [WMZV16; WLW10; WLWG11] with similar expressivity to JEDI use Attribute-Based Encryption (ABE) [GPSW06; BSW07]. Unfortunately, ABE is not suitable for JEDI because it is too expensive, especially in the context of **resource constraints** of IoT devices. Some IoT systems rule it out due to its latency alone [SBDHR18]. In the context of low-power devices, encryption with ABE would also consume too much power. JEDI circumvents the problem of using ABE or basic cryptography with two insights: (1) Even though ABE is too heavy for low-power devices, this does not mean that we must resort to only symmetric-key techniques. We show that certain IBE schemes [AKN] can be made practical for such devices. (2) **Time is another resource hierarchy**: a timestamp can be expressed as `year/month/day/hour`, and in this

hierarchical representation, any time range can be represented efficiently as a logarithmic number of subtrees. With this insight, we can simultaneously support URIs and expiry via a nonstandard use of a certain type of IBE scheme: WKD-IBE [AKN]. Like ABE, WKD-IBE is based on bilinear groups (pairings), but it is an order-of-magnitude less expensive than ABE as used in JEDI. To make JEDI practical on low-power devices, we design it to invoke WKD-IBE *rarely*, while relying on AES most of the time, much like session keys. Thus, JEDI achieves expressivity commensurate to IoT systems that do not encrypt data—significantly more expressive than AES-only solutions—while allowing several years of battery life for low-power low-cost IoT devices.

6.1.1.3 Integrity and Anonymity (Section 6.4)

In addition to being encrypted, messages should be signed so that the recipient of a message can be sure it was not sent by an attacker. This can be achieved via a certificate chain, as in SPKI/SDSI or bw2. Certificates can be distributed in a decentralized manner, just like encryption keys in Figure 6.2.

Certificate chains, however, are insufficient if anonymity is required. For example, consider an office space with an occupancy sensor in each office, each publishing to the same URI `buildingA/occupancy`. In aggregate, the occupancy sensors could be useful to inform, e.g., heating/cooling in the building, but individually, the readings for each room could be considered privacy-sensitive. The occupancy sensors in different rooms could use different certificate chains, if they were authorized/installed by different people. This could be used to deanonymize occupancy readings. To address this challenge, we adapt the WKD-IBE scheme that we use for end-to-end encryption to achieve an *anonymous* signature scheme that can encode the URI and expiry and support decentralized delegation. Using this technique, anonymous signatures are practical even on low-power embedded IoT devices.

6.1.1.4 Revocation (Section 6.5)

As stated above, JEDI keys support expiry. Therefore, it is possible to achieve a lightweight revocation scheme by delegating each key with short expiry and periodically renewing it to extend the expiry. To revoke a key, one simply does not renew it. We expect this expiry-based revocation to be sufficient for most use cases, especially for low-power devices, which typically just “sense and send.”

Enforcing revocation cryptographically, without relying on expiration, is challenging. As we discuss in Section 6.5, any cryptographically-enforced scheme that provides immediate revocation (i.e., keys can be revoked without waiting for them to expire) is subject to the fundamental limitation that the sender of a message must know which recipients are revoked when it encrypts the message. JEDI provides a form of immediate revocation, subject to this constraint. We use techniques from tree-based broadcast encryption [NNL01; DF02] to encrypt in such a way that all decryption keys for that URI, *except for ones on a revocation list*, can be used to decrypt. Achieving this is nontrivial because we have to combine broadcast encryption with JEDI’s semantics of hierarchical resources, expiry, and delegation. First, we modify broadcast encryption to support delegation, in such a way

that if a key is revoked, all delegations made with that key are also implicitly revoked. Then, we integrate broadcast revocation, in a *non-black-box* way, with JEDI's encryption and delegation, as a third resource hierarchy alongside URIs and expiry.

6.1.2 Summary of Evaluation

For our evaluation, we use JEDI to encrypt messages transmitted over bw2 [AKCCK17; Bw2], a deployed open-source messaging system for smart buildings, and demonstrate that JEDI's overhead is small in the critical path. We also evaluate JEDI for a commercially available sensor platform called "Hamilton" [Ham], and show that a Hamilton-based sensor sending one sensor reading every 30 seconds would see several years of battery lifetime when sending sensor readings encrypted with JEDI. As Hamilton is among the least powerful platforms that will participate in IoT (farthest to the right in Figure 6.1), this validates that JEDI is practical across the IoT spectrum.

6.2 JEDI's Model and Threat Model

A principal can post a message to a resource in a hierarchy by encrypting it according to the resource's URI, hierarchy's public parameters, and current time, and passing it to the underlying system that delivers it to the relevant subscribers. Given the secret key for a resource subtree and time range, a principal can generate a secret key for a subset of those resources and subrange of that time range, and give it to another principal, as in Figure 6.2. The receiving principal can use the delegated key to decrypt messages that are posted to a resource in that subset at a time during that subrange.

JEDI does not require the structure of the resource hierarchy to be fixed in advance. In Figure 6.2, the campus facilities manager, when granting access to `buildingA/*` to the building manager, need not be concerned with the structure of the subtree rooted at `buildingA`. This allows the building manager to organize `buildingA/*` independently.

6.2.1 Trust Assumptions

A principal is trusted for the resources it owns or was given access to (for the time ranges for which it was given access). In other words, an adversary who compromises a principal can read all resources that principal can read and forge new messages as if it were that principal. In particular, an adversary who compromises the authority for a resource hierarchy gains control over that resource hierarchy.

JEDI allows each principal to act as an authority for its own resource hierarchy in its own trust domain, without a single authority spanning all hierarchies. In particular, *principals* are not organized hierarchically; a principal may be delegated multiple keys, each belonging to a different resource hierarchy. In the example in Figure 6.2, Alice might also receive JEDI keys from her landlord granting access to resources in her apartment building, in a separate hierarchy where her landlord is the authority. If Alice owns resources she would like to delegate to others, she can set up her own hierarchy to represent those resources. Existing IoT systems with decentralized delegation, like `bw2` and `Vanadium`, use a similar model.

6.2.2 Applying JEDI to an Existing System

As shown in Figure 6.3, JEDI can be applied as a wrapper around existing many-to-many communication systems, including publish-subscribe systems for smart cities. The transfer of messages from producers to consumers is handled by the existing system. A common design used by such systems is to have a central broker (or router) forward messages; however, an adversary who compromises the broker can read all messages. In this context, JEDI's end-to-end encryption protects data from such an adversary. Publishers encrypt their messages with JEDI before passing them to the underlying communication system (without knowledge of who the subscribers are), and subscribers decrypt them with JEDI after receiving them from the underlying communication system (without knowledge of who the publishers are).

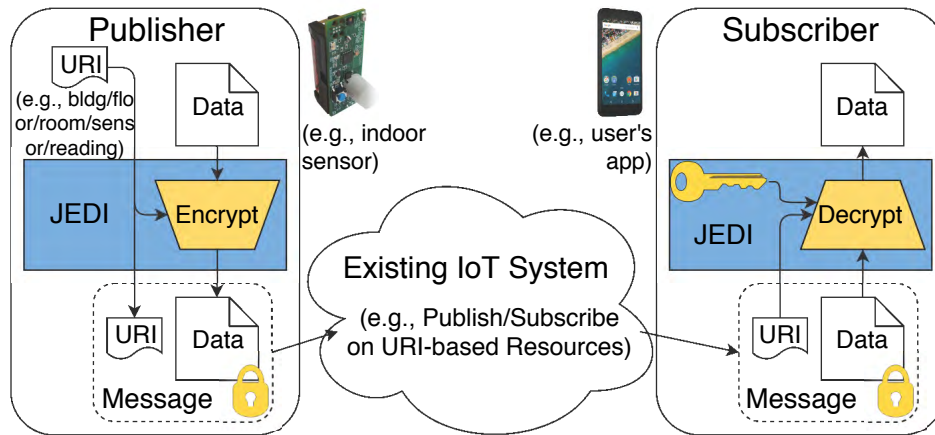


Figure 6.3: Applying JEDI to a smart buildings IoT system. Components introduced by JEDI are shaded. The subscriber's key is obtained via JEDI's decentralized delegation (Figure 6.2).

6.2.3 Comparison to a Naïve Key Server Model

To better understand the benefits of JEDI's model, consider the natural strawman of a trusted key server. This key server generates a key for every URI and time. A publisher encrypts each message for that URI with the same key. A subscriber requests this key from the trusted key server, which must first check if the subscriber is authorized to receive it. The subscriber can decrypt messages for a URI using this key, and contact the key server for a new key when the key expires. JEDI's model is better than this key server model as follows:

- *Improved security.* Unlike the trusted key server, which must always be online, the authority in JEDI can delegate qualified keys to some principals *and then go offline*, leaving these principals to qualify and delegate keys further. While the authority is offline, it is more difficult for an attacker to compromise it and easier for the authority to protect its secrets because it need only access them rarely. This reasoning is the basis of root Certificate Authorities (CAs), which access their master keys infrequently. In contrast, the trusted key server model requires a central trusted party (key server) to be online to grant/revoke access to any resource.
- *Improved privacy.* No single participant sees all delegations in JEDI. An adversary in JEDI who steals an authority's secret key can decrypt all messages for that hierarchy, but still does not learn who has access to which resource, and cannot access separate hierarchies to which the first authority has no access. In contrast, an adversary who compromises the key server learns who has access to which resource and can decrypt messages for all hierarchies.
- *Improved scalability.* In the campus IoT example above, if a building admin receives access to all sensors and all their different readings for a building, the admin must obtain a potentially very large number of keys, instead of one key for the entire building. Moreover, the campus-wide key server needs to grant decryption keys to each application owned by each employee or student at the university. Finally, the campus-wide key server must understand which delegations are allowed at lower levels in the hierarchy, requiring the entire hierarchy to be centrally administered.

6.2.4 IoT Gateways

Low-power wireless embedded sensors, due to power constraints, often do not use network protocols like Wi-Fi, and instead use specialized low-power protocols such as Bluetooth or IEEE 802.15.4. It is common for these devices to rely on an *application-layer gateway* to send data to computers outside of the low-power network [ZKCAJD15]. This gateway could be in the form of a phone app (e.g., Fitbit), or in the form of a specialized border router [Zig; Bra+12]. In some traditional setups, the gateway is responsible for performing encryption/authentication [PSWCT01]. JEDI accepts that gateways may be necessary for Internet connectivity, but does not rely on them for security—JEDI's cryptography is lightweight enough to run directly on the low-power sensor nodes. This approach prevents the gateway from becoming a single point of attack; an attacker who compromises the gateway cannot see or forge data for any device using that gateway.

6.2.5 Generalizability of JEDI's Model

Since JEDI decouples senders from receivers, it has no requirements on what happens at any intermediaries (e.g., does not require messages to be forwarded from publishers to subscribers in any particular way). Thus, JEDI works even when messages are exchanged in a broadcast medium, e.g., multicast. This also means that JEDI is more broadly applicable to systems with hierarchically organized resources. For example, URIs could correspond to filepaths in a file system, or URLs in a RESTful web service.

6.2.6 Security Goals

JEDI's goal is to ensure that principals can only read messages from or send messages to resources they have been granted access to receive from or send to. In the context of publish-subscribe, JEDI also hides the content of messages from an adversary who controls the router.

JEDI does not attempt to hide metadata relating to the actual transfer of messages (e.g., the URIs on which messages are published, which principals are publishing or subscribing to which resources, and timing). Hiding this metadata is a complementary task to achieving delegation and end-to-end encryption in JEDI, and techniques from the secure messaging literature [CSPZKA16; CGBM15; HLZZ15a] will likely be applicable.

6.3 End-to-End Encryption in JEDI

A central question answered in this section is: How should publishers encrypt messages before passing them to the underlying system for delivery (Section 6.3.4)? As explained in Section 6.1.1.2, although ABE, the obvious choice, is too heavy for low-power devices, we identify WKD-IBE, a more lightweight identity-based encryption scheme, as sufficient to achieve JEDI’s properties. The primary challenge is to encode a sufficiently expressive rendezvous point in the WKD-IBE ID (called a *pattern*) that publishers use to encrypt messages (Section 6.3.4).

6.3.1 Building Block: WKD-IBE

We first explain WKD-IBE [AKN], the encryption scheme that JEDI uses as a building block. Throughout this paper, we denote the security parameter as κ .

In WKD-IBE, messages are encrypted with *patterns*, and keys also correspond to patterns. A pattern is a list of values: $P = (\mathbb{Z}_p^* \cup \{\perp\})^\ell$. The notation $P(i)$ denotes the i th component of P , 1-indexed. A pattern P_1 *matches* a pattern P_2 if, for all $i \in [1, \ell]$, either $P_1(i) = \perp$ or $P_1(i) = P_2(i)$. In other words, if P_1 specifies a value for an index i , P_2 must match it at i . Note that the “matches” operation is not commutative; “ P_1 matches P_2 ” does not imply “ P_2 matches P_1 ”.

We refer to a component of a pattern containing an element of \mathbb{Z}_p^* as *fixed*, and to a component that contains \perp as *free*. To aid our presentation, we define the following sets:

Definition 6.3.1. For a pattern S , we define:

$$\begin{aligned} \text{fixed}(S) &= \{(i, S(i)) \mid S(i) \neq \perp\} \\ \text{free}(S) &= \{i \mid S(i) = \perp\} \end{aligned}$$

A key for pattern P_1 can decrypt a message encrypted with pattern P_2 if $P_1 = P_2$. Furthermore, a key for pattern P_1 can be used to derive a key for pattern P_2 , as long as P_1 matches P_2 . In summary, the following is the syntax for WKD-IBE.

- **Setup** $(1^\kappa, 1^\ell) \rightarrow \text{Params, MasterKey}$;
- **KeyDer** $(\text{Params, Key}_{\text{Pattern}_A}, \text{Pattern}_B) \rightarrow \text{Key}_{\text{Pattern}_B}$, derives a key for Pattern_B , where either $\text{Key}_{\text{Pattern}_A}$ is the MasterKey , or Pattern_A matches Pattern_B ;
- **Encrypt** $(\text{Params, Pattern}, m) \rightarrow \text{Ciphertext}_{\text{Pattern}, m}$;
- **Decrypt** $(\text{Key}_{\text{Pattern}}, \text{Ciphertext}_{\text{Pattern}, m}) \rightarrow m$.

We use the WKD-IBE construction in §3.2 of [AKN], based on BBG HIBE [BBG05]. Like the BBG construction, it has constant-size ciphertexts, but requires the maximum pattern length ℓ to be known at Setup time. In this WKD-IBE construction, patterns containing \perp can only be used in **KeyDer**, not in **Encrypt**; we extend it to support encryption with patterns containing \perp . We include the WKD-IBE construction with our optimizations in [KHAPC19].

6.3.2 Concurrent Hierarchies in JEDI

WKD-IBE was originally designed to allow delegation in a *single* hierarchy. For example, the original suggested use case of WKD-IBE was to generate secret keys for a user’s email addresses in all valid subdomains, such as `sysadmin@*.univ.edu` [AKN].

JEDI, however, uses WKD-IBE in a nonstandard way to simultaneously support *multiple* hierarchies, one for URIs and one for expiry (and later in Section 6.5, one for revocation), each in the vein of HIBE. We think of the ℓ components of a WKD-IBE pattern as “slots” that are initially empty, and are progressively filled in with calls to `KeyDer`. To combine a hierarchy of maximum depth ℓ_1 (e.g., the URI hierarchy) and a hierarchy of maximum depth ℓ_2 (e.g., the expiry hierarchy), one can `Setup` WKD-IBE with the number of slots equal to $\ell = \ell_1 + \ell_2$. The first ℓ_1 slots are filled in left-to-right for the first hierarchy and the remaining ℓ_2 slots are filled in left-to-right for the second hierarchy (Figure 6.4).

6.3.3 Overview of Encryption in JEDI

Each principal maintains a **key store** containing WKD-IBE decryption keys. To create a resource hierarchy, any principal can call the WKD-IBE `Setup` function to create a resource hierarchy. It releases the *public parameters* and stores the *master secret key* in its key store, making it the authority of that hierarchy. To delegate access to a URI prefix for a time range, a principal (possibly the authority) searches its key store for a set of keys for a superset of those permissions. It then qualifies those keys using `KeyDer` to restrict them to the specific URI prefix and time range (Section 6.3.5), and sends the resulting keys to the recipient of the delegation.² The recipient accepts the delegation by adding the keys to its key store.

Before sending a message to a URI, a principal encrypts the message using WKD-IBE. The pattern used to encrypt it is derived from the URI and the current time (Section 6.3.4), which are included along with the ciphertext. When a principal receives a message, it searches its key store, using the URI and time included with the ciphertext, for a key to decrypt it.

In summary, JEDI provides the following API:

`Encrypt(Message, URI, Time) → Ciphertext`

`Decrypt(Ciphertext, URI, Time, KeyStore) → Message`

`Delegate(KeyStore, URIPrefix, TimeRange) → KeySet`

`AcceptDelegation(KeyStore, KeySet) → KeyStore'`

Note that the WKD-IBE public parameters are an implicit argument to each of these functions. Finally, although the above API lists the arguments to `Delegate` as `URIPrefix` and `TimeRange`, JEDI actually supports succinct delegation over more complex sets of URIs and timestamps (see Section 6.3.7).

²JEDI does not govern *how* the key set is transferred to the recipient, as there are existing solutions for this. One can use an existing protocol for one-to-one communication (e.g., TLS) to securely transfer the key set. Or, one can encrypt the key set with the recipient’s (normal, non-WKD-IBE) public key, and place it in a common storage area.

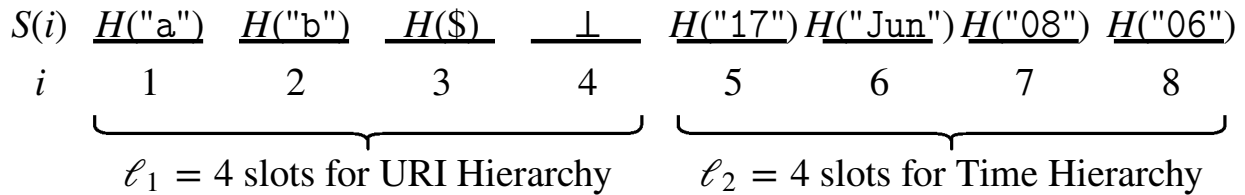


Figure 6.4: Pattern S used to encrypt message sent to a/b on June 08, 2017 at 6 AM. The figure uses 8 slots for space reasons; JEDI is meant to be used with more slots (e.g., 20).

6.3.4 Expressing URI/Time as a Pattern

A message is encrypted using a pattern derived from (1) the URI to which the message is addressed, and (2) the current time. Let $H : \{0, 1\}^* \rightarrow \mathbb{Z}_p^*$ be a collision-resistant hash function. Let $\ell = \ell_1 + \ell_2$ be the pattern length in the hierarchy's WKD-IBE system. We use the first ℓ_1 slots to encode the URI, and the last ℓ_2 slots to encode the time.

Given a URI of length d , such as a/b/c ($d = 3$ in this example), we split it up into individual components, and append a special terminator symbol $\$$: ("a", "b", "c", "\$"). Using H , we map each component to \mathbb{Z}_p^* , and then put these values into the first $d + 1$ slots. If S is our pattern, we would have $S(1) = H("a")$, $S(2) = H("b")$, $S(3) = H("c")$, and $S(4) = H("$")$ for this example. Now, we encode the time range into the remaining ℓ_2 slots. Any timestamp, with the granularity of an hour, can be represented hierarchically as (year, month, day, hour). We encode this into the pattern like the URI: we hash each component, and assign them to consecutive slots. The final ℓ_2 slots encode the time, so the depth of the time hierarchy is ℓ_2 . The terminator symbol $\$$ is not needed to encode the time, because timestamps always have exactly ℓ_2 components. For example, suppose that a principal sends a message to a/b on June 8, 2017 at 6 AM. The message is encrypted with the pattern in Figure 6.4.

6.3.5 Producing a Key Set for Delegation

Now, we explain how to produce a key set corresponding to a URI prefix and time range. To express a URI prefix as a pattern, we do the same thing as we did for URIs, without the terminator symbol $\$$. For example, a/b/* is encoded in a pattern S as $S(1) = H("a")$, $S(2) = H("b")$, and all other slots free. Given the private key for S , one can use WKD-IBE's **KeyDer** to fill in slots $3 \dots \ell_1$. This allows one to generate the private key for a/b, a/b/c, etc.—any URI for which a/b is a prefix. To grant access to only a specific resource (a full URI, not a prefix), the $\$$ is included as before.

In encoding a time range into a pattern, single timestamps (e.g., granting access for an hour) are done as before. The hierarchical structure for time makes it possible to succinctly grant permission for an entire day, month, or year. For example, one may grant access for all of 2017 by filling in slot ℓ_2 with $H("2017")$ and leaving the final $\ell_2 - 1$ slots, which correspond to month, day, and year, free. Therefore, to grant permission over a time range, *the number of keys granted is logarithmic in the length of the time range*. For example, to delegate access to a URI from October 29, 2014 at 10 PM until December 2, 2014 at 1 AM, the following keys

need to be generated: 2014/Oct/29/23, 2014/Oct/29/24, 2014/Oct/30/*, 2014/Oct/31/*, 2014/Nov/*, 2014/Dec/01/*, and 2014/Dec/02/01. The tree can be chosen differently to support longer time ranges (e.g., additional level representing decades), change the granularity of expiry (e.g., minutes instead of hours), trade off encryption time for key size (e.g., deeper/shallower tree), or use a more regular structure (e.g., binary encoding with logarithmic split). For example, our implementation uses a depth-6 tree (instead of depth-4), to be able to delegate time ranges with fewer keys.

In summary, to produce a key set for delegation, first determine which subtrees in the time hierarchy represent the time range. For each one, produce a separate pattern, and encode the time into the last ℓ_2 slots. Encode the URI prefix in the first ℓ_1 slots of each pattern. Finally, generate the keys corresponding to those patterns, using keys in the key store.

6.3.6 Optimizations for Low-Power Devices

On low-power embedded devices, performing a single WKD-IBE encryption consumes a significant amount of energy. Therefore, we design JEDI with optimizations to WKD-IBE.

6.3.6.1 Hybrid Encryption and Key Reuse

JEDI uses WKD-IBE in a hybrid encryption scheme. To encrypt a message m in JEDI, one samples a symmetric key k , and encrypts k with JEDI to produce ciphertext c_1 . The pattern used for WKD-IBE encryption is chosen as in Section 6.3.4 to encode the *rendezvous point*. Then, one encrypts m using k to produce ciphertext c_2 . The JEDI ciphertext is (c_1, c_2) .

For subsequent messages, one reuses k and c_1 ; the new message is encrypted with k to produce a new c_2 . One can keep reusing k and c_1 until the WKD-IBE pattern for encryption changes, which happens at the end of each hour (or other interval used for expiry). At this time, JEDI performs *key rotation* by choosing a new k , encrypting it with WKD-IBE using the new pattern, and then proceeding as before. Therefore, *most messages only incur cheap symmetric-key encryption*.

This also reduces the load on subscribers. The JEDI ciphertexts sent by a publisher during a single hour will all share the same c_1 . Therefore, the subscriber can decrypt c_1 once for the first message to obtain k , and *cache* the mapping from c_1 to k to avoid expensive WKD-IBE decryptions for future messages sent during that hour.

Thus, expensive WKD-IBE operations are only performed upon key rotation, which happens *rarely*—once an hour (or other granularity chosen for expiry) for each resource.

6.3.6.2 Precomputation with Adjustment

Even with hybrid encryption and key reuse to perform WKD-IBE encryption rarely, WKD-IBE contributes significantly to the overall power consumption on low-power devices. Therefore, this section explores how to perform individual WKD-IBE encryptions more efficiently.

Most of the work to encrypt under a pattern S is in computing the quantity $Q_S = g_3 \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i}$, where g_3 and the h_i are part of the WKD-IBE public parameters. One may

consider computing Q_S once, and then reusing its value when computing future encryptions under the same pattern S . Unfortunately, this alone does not improve efficiency because the pattern S used in one WKD-IBE encryption is different from the pattern T used for the next encryption.

JEDI, however, observes that S and T are similar; they match in the ℓ_1 slots corresponding to the URI, and the remaining ℓ_2 slots will correspond to adjacent leaves in the time tree. JEDI takes advantage of this by efficiently *adjusting* the precomputed value Q_S to compute Q_T as follows:

$$Q_T = Q_S \cdot \prod_{\substack{(i,b_i) \in \text{fixed}(T) \\ i \in \text{free}(S)}}} h_i^{b_i} \cdot \prod_{\substack{(i,a_i) \in \text{fixed}(S) \\ i \in \text{free}(T)}}} h_i^{-a_i} \cdot \prod_{\substack{(i,a_i) \in \text{fixed}(S) \\ (i,b_i) \in \text{fixed}(T) \\ a_i \neq b_i}} h_i^{b_i - a_i}$$

This requires one \mathbb{G}_1 exponentiation per differing slot between S and T (i.e., the Hamming distance). Because S and T usually differ in only the final slot of the time hierarchy, this will usually require one \mathbb{G}_1 exponentiation total, substantially faster than computing Q_T from scratch. Additional exponentiations are needed at the end of each day, month, and year, but they can be eliminated by maintaining additional precomputed values corresponding to the start of the current day, current month, and current year.

The protocol remains secure because a ciphertext is distributed identically whether it was computed from a precomputed value Q_S or via regular encryption as in [KHAPC19].

6.3.7 Extensions

Via simple extensions, JEDI can support (1) wildcards in the *middle* of a URI or time, and (2) forward secrecy. We describe these extensions in the appendix of our extended paper [KHAPC19].

6.3.8 Security Guarantee

We formalize the security of JEDI's encryption below.

Theorem 6.3.2. *Suppose JEDI is instantiated with a Selective-ID CPA-secure [BB04; AKN], history-independent (defined in our extended paper [KHAPC19]) WKD-IBE scheme. Then, no probabilistic polynomial-time adversary \mathcal{A} can win the following security game against a challenger \mathcal{C} with non-negligible advantage:*

Initialization. \mathcal{A} selects a (URI, time) pair to attack.

Setup. \mathcal{C} gives \mathcal{A} the public parameters of the JEDI instance.

Phase 1. \mathcal{A} can make three types of queries to \mathcal{C} :

1. \mathcal{A} asks \mathcal{C} to create a principal; \mathcal{C} returns a name in $\{0, 1\}^*$, which \mathcal{A} can use to refer to that principal in future queries. A special name exists for the authority.
2. \mathcal{A} asks \mathcal{C} for the key set of any principal; \mathcal{C} gives \mathcal{A} the keys that the principal has. At the time this query is made, the requested key may **not** contain a key whose URI and time are both prefixes of the challenge (URI, time) pair.
3. \mathcal{A} asks \mathcal{C} to make any principal delegate a key set of \mathcal{A} 's choice to another principal (specified by names in $\{0, 1\}^*$).

Challenge. When \mathcal{A} chooses to end Phase 1, it sends \mathcal{C} two messages, m_0 and m_1 , of the same length. Then \mathcal{C} chooses a random bit $b \in \{0, 1\}$, encrypts m_b under the challenge (URI, time) pair, and gives \mathcal{A} the ciphertext.

Phase 2. \mathcal{A} can make additional queries as in Phase 1.

Guess. \mathcal{A} outputs $b' \in \{0, 1\}$, and wins the game if $b = b'$. The advantage of an adversary \mathcal{A} is $\left| \Pr[\mathcal{A} \text{ wins}] - \frac{1}{2} \right|$.

We prove this theorem in our extended paper [KHAPC19].

Although we only achieve selective security in the standard model (like much prior work [BBG05; AKN]), one can achieve adaptive security if the hash function H in Section 6.3.5 is modeled as a random oracle [AKN]. It is sufficient for JEDI to use a CPA-secure (rather than CCA-secure) encryption scheme because JEDI messages are signed, as detailed below in Section 6.4.

6.4 Integrity in JEDI

To prevent an attacker from flooding the system with messages, spoofing fake data, or actuating devices without permission, JEDI must ensure that a principal can only send a message on a URI if it has permission. For example, an application subscribed to `buildingA/floor2/roomLHall/sensor0/temp` should be able to verify that the readings it is receiving are produced by `sensor0`, not an attacker. In addition to subscribers, an intermediate party (e.g., the router in a publish-subscribe system) may use this mechanism to filter out malicious traffic, without being trusted to read messages.

6.4.1 Starting Solution: Signature Chains

A standard solution in the existing literature, used by SPKI/SDSI [CEEFM01], Vanadium [TS16], and bw2 [AKCCK17], is to include a certificate chain with each message. Just as permission to subscribe to a resource is granted via a chain of delegations in Section 6.3, permission to publish to a resource is also granted via a chain of delegations. Whereas Section 6.3 includes WKD-IBE keys in each delegation, these integrity solutions delegate signed certificates. To send a message, a principal encrypts it (Section 6.3), signs the ciphertext, and includes a certificate chain that proves that the signing keypair is authorized for that URI and time.

6.4.2 Anonymous Signatures

The above solution reveals the sender's identity (via its public key) and the particular chain of delegations that gives the sender access. For some applications this is acceptable, and its auditability may even be seen as a benefit. For other applications, the sender must be able to send a message anonymously. See Section 6.1.1.3 for an example. How can we reconcile *access control* (ensuring the sender has permission) and *anonymity* (hiding who the sender is)?

6.4.2.1 Starting Point: WKD-IBE Signatures

Our solution is to use a signature scheme based on WKD-IBE. Abdalla et al. [AKN] observe that WKD-IBE can be extended to a signature scheme in the same vein as has been done for IBE [BF01] and HIBE [GS02]. To sign a message $m \in \mathbb{Z}_p^*$ with a key for pattern S , one uses `KeyDer` to fill in a slot with m , and presents the decryption key as a signature.

This is our starting point for designing anonymous signatures in JEDI. A message can be signed by first hashing it to \mathbb{Z}_p^* and signing the hash as above. Just as consumers receive decryption keys via a chain of delegations (Section 6.3), publishers of data receive these signing keys via chains of delegations.

6.4.2.2 Anonymous Signatures in JEDI

The construction in Section 6.4.2.1 has two shortcomings. First, signatures are *large*, linear in the number of fixed slots of the pattern. Second, it is unclear if they are truly *anonymous*.

Signature size. As explained in Section 6.3, we use a construction of WKD-IBE based on BBG HIBE [BBG05]. BBG HIBE supports a property called *limited delegation* in which a secret key can be reduced in size, in exchange for limiting the depth in the hierarchy at which subkeys can be generated from it. We observe that the WKD-IBE construction also supports this feature. Because we need not support **KeyDer** for the decryption key acting as a signature, we use limited delegation to compress the signature to just two group elements.

Anonymity. The technique in Section 6.4.2.1 transforms an encryption scheme into a signature scheme, but the resulting signature scheme is not necessarily anonymous. For the particular construction of WKD-IBE that we use, however, we prove that the resulting signature scheme is indeed anonymous. Our insight is that, for this construction of WKD-IBE, keys are *history-independent* in the following sense: **KeyDer**, for a fixed Params and Pattern_B, returns a private key Key_{Pattern_B} with the *exact same distribution* regardless of Key_{Pattern_A} (see Section 6.3.1 for notation). Because signatures, as described in Section 6.4.2.1, are private keys generated with **KeyDer**, they are also history-independent; a signature for a pattern has the same distribution regardless of the key used to generate it. This is precisely the anonymity property we desire.

6.4.3 Optimizations for Low-Power Devices

As in Section 6.3.6.1, we must avoid computing a WKD-IBE signature for every message. A simple way to do this is to sample a digital signature keypair each hour, sign the verifying key with WKD-IBE at the beginning of the hour, and sign messages during the hour with the corresponding signing key.

Unfortunately, this may still be too expensive for low-power embedded devices because it requires a digital signature, which requires asymmetric-key cryptography, for *every* message. We can circumvent this by instead (1) choosing a *symmetric* key k every hour, (2) signing k at the start of each hour (using WKD-IBE for anonymity), and (3) using k in an *authenticated broadcast protocol* to authenticate messages sent during the hour. An authenticated broadcast protocol, like μ TESLA [PSWCT01], generates a MAC for each message using a key whose hash is the previous key; thus, the single signed key k allows the recipient to verify later messages, whose MACs are generated with hash preimages of k . In general, this design requires stricter time synchronization than the one based on digital signatures, as the key used to generate the MAC depends on the time at which it is sent. However, for the sense-and-send use case typical of smart buildings, sensors anyway publish messages on a fixed schedule (e.g., one sample every x seconds), allowing the key to depend only on the message index. Thus, timely message delivery is the only requirement. Our scheme differs from μ TESLA because the first key (end of the hash chain) is signed using WKD-IBE.

Additionally, we use a technique similar to precomputation with adjustment (Section 6.3.6.2) for anonymous signatures. Conceptually, **KeyDer**, which is used to produce signatures, can be understood as a two-step procedure: (1) produce a key of the correct form and structure (called **NonDelegableKeyDer**), and (2) re-randomize the key so that it can be safely delegated (called **ResampleKey**). Re-randomization can be accelerated using the same precomputed value Q_S that JEDI uses for encryption (Section 6.3.6.2), which can be efficiently adjusted from one pattern to the next. The result of **NonDelegableKeyDer** can also be adjusted to obtain the corresponding

result for a similar pattern more efficiently. We fully explain our adjustment technique for signatures in our extended paper [KHAPC19].

Finally, WKD-IBE signatures as originally proposed (Section 6.4.2.1) are verified by encrypting a random message under the pattern corresponding to the signature, and then attempting to decrypt it using the key acting as a signature. We provide a more efficient signature verification algorithm for this construction of WKD-IBE in our extended paper [KHAPC19].

6.4.4 Security Guarantee

The integrity guarantees of the method in this section can be formalized using a game very similar to the one in Theorem 6.3.2, so we do not present it here for brevity. We do, however, formalize the anonymous aspect of WKD-IBE signatures:

Theorem 6.4.1. *For any well-formed keys k_1, k_2 corresponding to the same (URI, time) pair in the same resource hierarchy, and any message $m \in \mathbb{Z}_p^*$, the distribution of signatures over m produced using k_1 is information-theoretically indistinguishable from (i.e., equal to) the distribution of signatures over m produced using k_2 .*

This implies that even a powerful adversary who observes the private keys held by all principals cannot distinguish signatures produced by different principals, for a fixed message and pattern. No computational assumptions are required. We prove Theorem 6.4.1 in the appendix of our extended paper [KHAPC19].

6.5 Revocation in JEDI

This section explains how JEDI keys may be revoked.

6.5.1 Simple Solution: Revocation via Expiry

A simple solution for revocation is to rely on expiration. In this solution, all keys are time-limited, and delegations are periodically refreshed, according to a higher layer protocol, by granting a new key with a later expiry time. In this setup, the principal who granted a key can easily revoke it by not refreshing that delegation when the key expires. We expect this solution to be sufficient for many applications of JEDI.

6.5.2 Immediate Revocation

Some disadvantages of the solution in Section 6.5.1 are that (1) principals must periodically come online to refresh delegations, and (2) revocation only takes effect when the delegated key expires. We would like a solution without these disadvantages.

However, any revocation scheme that does not wait for keys to expire is subject to set of *inherent* limitations. The recipient of the revoked delegation still has the revoked decryption key, so it can still decrypt messages encrypted in the same way. This means that we must either (1) rely on intermediate parties to modify ciphertexts so that revoked keys cannot decrypt them, or (2) require senders to be aware of the revocation, and encrypt messages in a different way so that revoked keys cannot decrypt them. Neither solution is ideal: (1) makes assumptions about how messages are delivered, which we have avoided thus far (Section 6.2), and requires trust in an intermediary to modify ciphertexts, and (2) weakens the decoupling of senders and receivers (Section 6.1.1). We adopt the second compromise: while senders will not need to know who are the receivers, they will need to know who has been revoked.

6.5.3 Immediate Revocation in JEDI

We extend tree-based broadcast encryption [NNL01; DF02] to support decentralized delegation of decryption keys, and incorporate it into JEDI. We use tree-based broadcast encryption because it only requires senders to know about *revoked* users when encrypting messages, as opposed to *all* users in the system (as is required by other broadcast encryption schemes).

6.5.3.1 Tree-based Broadcast Encryption

Existing work [NNL01; DF02] proposes two methods of tree-based broadcast encryption: Complete Subtree (CS) and Subset Difference (SD). We focus on the CS method here.

The CS method is based on a binary tree (Figure 6.5) where each node corresponds to a separate keypair. Each user corresponds to a leaf of the tree and has the secret keys for all nodes on the root-to-leaf path. To encrypt a message that is decryptable by a subset of users, one finds

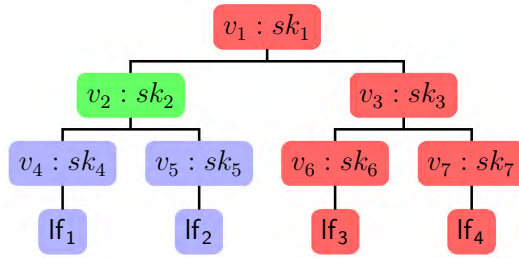


Figure 6.5: Key management of the CS method. Red nodes indicate nodes associated with revoked leaves. The green node is the root of the subtree covering unrevoked leaves.

a collection of subtrees that include all leaves except those corresponding to revoked users and encrypts the message multiple times using the public keys corresponding to the root of each subtree. By associating each node with an ID and encrypting with IBE, one can avoid generating a separate keypair for each node.

6.5.3.2 Modifying Broadcast Encryption for Delegation

Users in broadcast encryption do not map one-to-one to users in JEDI. To avoid confusion, we refer to “users” in broadcast encryption as “leaves” (abbreviated lf).

We modify the CS method to support delegation, as follows. Each key corresponds to a range of consecutive leaves. When a user qualifies a key to delegate to another principal, she produces a new key corresponding to a subrange of the leaves of the original key. When a key is revoked, publishers are informed of the range of leaves corresponding to the revoked key. Then, they encrypt new messages using the CS method, choosing subtrees that cover all leaves except those corresponding to revoked leaves. If a key is revoked, that key and all keys derived from it can no longer decrypt messages, which is a property that we want. Thus, if Alice has k leaves, she must store secret keys for $O(k + \log n)$ nodes, where n is the total number of leaves (so the depth of the tree is $\log n$).

In JEDI, we reduce this to $O(\log n)$ secret keys by using HIBE. We give each node v_i an identifier $\text{id}(v_i) \in \{0, 1\}^*$ that describes the path from the root of the tree to that node. In particular, if v_j is an ancestor of v_i , then $\text{id}(v_j)$ is a prefix of $\text{id}(v_i)$. Note that if we use HIBE with these IDs directly, a user with the secret key for the root can generate keys for all nodes in the tree. To fix this, we use a property called *limited delegation*, introduced by prior work [BBG05], to generate a HIBE key that is unqualifiable (i.e., cannot be extended). For example, if Alice has leaves lf₃ to lf₄ in Figure 6.5, she stores an unqualifiable key for node v_1 and a qualifiable key for node v_3 . In general, each user must store $O(\log k)$ qualifiable keys and $O(\log n)$ unqualifiable keys, thus $O(\log k + \log n)$ total.

6.5.3.3 Using Delegable Broadcast Encryption in JEDI

Secret keys in our modified broadcast encryption scheme consist of HIBE keys, so incorporating it into JEDI is simple. As discussed in Section 6.3.2, JEDI uses WKD-IBE in a way that provides multiple concurrent hierarchies, each in the vein of HIBE. Therefore, we can instantiate a third hierarchy of depth $\ell_3 = \log n$ and use it for revocation.

Let r be the number of revoked keys. The CS method has $O(r \log \frac{n}{r})$ -size ciphertexts, so JEDI ciphertexts grow to this size when revocation is used. When encrypting a message, senders use the same encryption protocol from Section 6.3 for the first $\ell_1 + \ell_2$ slots, and repeat the process, filling in the remaining ℓ_3 slots with the ID of each node used for broadcast encryption. The size of secret keys is $O(\log k + \log n)$ after our modifications to the CS method, so JEDI keys grow by this factor, to a total of $O((\log k + \log n) \cdot \log T)$ WKD-IBE keys, where T is the length of the time range for expiry.

The construction in this section works to revoke decryption keys, but cannot be used with anonymous signatures (Section 6.4.2). Extensions of tree-based broadcast encryption to signatures exist [LPY12a; LPY12b], and we expect them to be useful to develop a construction for anonymous signatures.

How can JEDI inform publishers which leaves are revoked? One simple option is to have a global revocation list, which principals can append to. However, storing this information in a single list becomes a central point of attack, which we have avoided in our system thus far (Section 6.2). To avoid this, one can store the revocation list in a global-scale blockchain, such as Bitcoin or Ethereum, which would require an adversary to be exceptionally powerful to mount a successful attack. When revoking a set of leaves, a principal uses those keys to sign a predetermined object (as in Section 6.4.2), proving it owns an ancestor of that key in the hierarchy. To keep the revocation list private, one can use JEDI’s encryption to ensure that only principals with permission to publish to a particular resource can see which keys are revoked for that resource (since publishers too have signing keys, as described in Section 6.4).

6.5.4 Security Guarantee

The security guarantee for immediate revocation can be stated as a modification to the game in Theorem 6.3.2. In the Initialization Phase, when \mathcal{A} gives \mathcal{C} the challenge (URI, time), \mathcal{A} additionally submits a list of revoked leaves. Furthermore, \mathcal{A} may compromise principals in possession of private keys that can decrypt the challenge (URI, time) pair during Phases 1 and 2, as long as all leaves corresponding to those keys are in the revocation list submitted in the Initialization Phase. We provide a proof in the appendix of the extended paper [KHAPC19].

6.5.5 Optimizing JEDI’s Immediate Revocation

A single JEDI ciphertext, with revocation enabled, consists of $O(r \log \frac{n}{r})$ WKD-IBE ciphertexts. To compute them efficiently, we observe that there is a large overlap in the patterns used in individual WKD-IBE encryptions, allowing us to use the “precomputation with adjustment” strategy from Section 6.3.6.2.

Even with the above optimization, immediate revocation substantially increases the cost of JEDI’s cryptography. To reduce this cost, we make three observations. First, to extend JEDI’s hybrid encryption to work with revocation, it is sufficient to additionally rotate keys whenever the revocation list changes, in addition to the end of each hour (as in Section 6.3.6.1). This means that, in the common case where the revocation list does not change in between two messages, efficient

symmetric-key encryption can be used. Second, the revocation list used to encrypt a message need only contain revoked leaves for the *particular URI* to which the message is sent. This not only makes the broadcast encryption more efficient (smaller r), but also causes the effective revocation list for a stream of data to change even more rarely, allowing JEDI to benefit more from hybrid encryption. Third, we can do the same thing as above using the expiry time rather than the URI, allowing us to *cull* the revocation list by removing keys from it once they expire.

The efficiency of hybrid encryption depends on the revocation list changing *rarely*. We believe this is a reasonable assumption; most revocation will be handled by expiry, so immediate revocation is only needed if a principal must lose access *unexpectedly*. In the smart buildings use case (Section 6.1), for example, a key would need to be revoked if a principal unexpectedly transfers to another job.

The SD method for tree-based broadcast encryption can also be extended to support delegation and incorporated into JEDI (described in the appendix of our extended paper [KHAPC19]), The SD method has smaller ciphertexts but larger keys.

6.6 Implementation

We implemented JEDI as a library in the Go programming language. We expect that only a few applications will require the anonymous signature protocol in Section 6.4.2 or the tree-based revocation protocol in Section 6.5.3; most applications can use signature chains (Section 6.4.1) for integrity and expiry for revocation (Section 6.5.1). Therefore, our implementation makes anonymous signatures optional and implements revocation separately. We expect JEDI’s key delegation to be computed on relatively powerful devices, like laptops, smartphones, or Raspberry Pis; less powerful devices (e.g., right half of Figure 6.1) will primarily send and receive messages, rather than generate keys for delegation. Therefore, our focus for low-power platforms was on the “sense-and-send” use case [BMPR14; DCS07; Fel09] typical of indoor environmental sensing, where a device periodically publishes sensor readings to a URI. Whereas our Go library provides higher-level abstractions, we expect low-power devices to use JEDI’s crypto library directly.

6.6.1 C/C++ Library for JEDI’s Cryptography

As part of JEDI, we implemented a cryptography library optimized in assembly for three different architectures typical of IoT platforms (Figure 6.1). It implements WKD-IBE and JEDI’s optimizations and modifications (in Section 6.3.6, Section 6.4.3, and our full paper). The construction of WKD-IBE is based on a bilinear group in which the Bilinear Diffie-Hellman Exponent assumption holds. We originally planned to use Barreto-Naehrig elliptic curves [KKSK16; Che06] to implement WKD-IBE. Unfortunately, a recent attack on Barreto-Naehrig curves [KB16] reduced their estimated security level from 128 bits to at most 100 bits [BD]. Therefore, we use the recent BLS12-381 elliptic curve [Bow18].

State-of-the-art cryptography libraries implement BLS12-381, but none of them, to our knowledge, optimize for microarchitectures typical of low-power embedded platforms. To improve energy consumption, we implemented BLS12-381 in C/C++, profiled our implementation, and re-wrote performance-critical routines in assembly. We focus on ARM Cortex-M, an IoT-focused family of 32-bit microprocessors typical of contemporary low-power embedded sensor platforms [Ham; Cam17; Imi]. Cortex-M processors have been used in billions of devices, including commercial IoT offerings such as Fitbit and Nest Protect. Our assembly targets Cortex-M0+, which is among the least powerful of processors in the Cortex-M series, and of those used in IoT devices (farthest to the right in Figure 6.1). By demonstrating the practicality of JEDI on Cortex-M0+, we establish that JEDI is viable across the spectrum of IoT devices (Figure 6.1).

The main challenge in targeting Cortex-M0+ is that the 32-bit multiply instruction provides only the lower 32 bits of the product. Even on more powerful microarchitectures without this limitation (e.g., Intel Core i7), most CPU time ($\geq 80\%$) is spent on multiply-intensive operations (e.g., BigInt multiplication and Montgomery reduction), so the lack of such an instruction was a performance bottleneck. As a workaround, our assembly code emulates multiply-accumulate with carry in 23 instructions. Cortex-M3 and Cortex-M4, which are more commonly used than Cortex-M0+, have instructions for 32-bit multiply-accumulate which produce the entire 64-bit result; we expect JEDI to be more efficient on those processors.

We also wrote assembly to optimize BLS12-381 for x86-64 and ARM64, representative of server/laptop and smartphone/Raspberry Pi, respectively (first two tiers in Figure 6.1). Thus, our Go library, which runs on these non-low-power platforms, also benefits from low-level assembly optimizations.

6.6.2 Application of JEDI to bw2

We used our JEDI library to implement end-to-end encryption in bw2, a syndication and authorization system for IoT. bw2's syndication model is based on publish-subscribe, explained in Section 6.1. Here we discuss bw2's authorization model. Access to resources is granted via certificate chains from the authority of a resource hierarchy to a principal. Individual certificates are called Declarations of Trust (DOTs). bw2 maintains a publicly accessible registry of DOTs, implemented using blockchain smart contracts, so that principals can find the DOTs they need to form DOT chains. A *trusted* router enforces permissions granted by DOTs. Principals must present DOT chains when publishing/subscribing to resources, and the router verifies them. Note that a compromised router can read messages.

We use JEDI to enforce bw2's authorization semantics with end-to-end encryption. DOTs granting permission to subscribe now contain WKD-IBE keys to decrypt messages. By default, DOTs granting permission to publish to a URI remain unchanged, and are used as in Section 6.4.1. WKD-IBE keys may also be included in DOTs granting publish permission, for anonymous signatures (Section 6.4.2). Using our library for JEDI, we implemented a wrapper around the bw2 client library. It transparently encrypts and decrypts messages using WKD-IBE, and includes WKD-IBE parameters and keys in DOTs and principals, as needed for JEDI. bw2 signs each message with a digital signature (first alternative in Section 6.4.3).

The bw2-specific wrapper is less than 900 lines of Go code. Our implementation required no changes to bw2's client library, router, blockchain, or core—it is a separate module. Importantly, it provides the same API as the standard bw2 client library. Thus, it can be used as a drop-in replacement for the standard bw2 client library, to easily add end-to-end encryption to existing bw2 applications with minimal changes.

6.7 Evaluation

We evaluate JEDI via microbenchmarks, determine its power consumption on a low-power sensor, measure the overhead of applying it to bw2, and compare it to other systems.

6.7.1 Microbenchmarks

Benchmarks labeled “Laptop” were produced on a Lenovo T470p laptop with an Intel Core i7-7820HQ CPU @ 2.90 GHz. Benchmarks labeled “Raspberry Pi” were produced on a Raspberry Pi 3 Model B+ with an ARM Cortex-A53 @ 1.4 GHz. Benchmarks labeled “Sensor” were produced on a commercially available ultra low-power environmental sensor platform called “Hamilton” with an ARM Cortex-M0+ @ 48 MHz. We describe Hamilton in more detail in Section 6.7.3.

6.7.1.1 Performance of BLS12-381 in JEDI

Operation	Laptop	Rasp. Pi	Sensor
\mathbb{G}_1 Mul. (Chosen Scalar)	109 μ s	1.33 ms	509 ms
\mathbb{G}_2 Mul. (Chosen Scalar)	343 μ s	3.86 ms	1.44 s
\mathbb{G}_T Mul. (Rand. Scalar)	504 μ s	5.47 ms	1.90 s
\mathbb{G}_T Mul. (Chosen Scalar)	507 μ s	5.48 ms	2.81 s
Pairing	1.29 ms	14.0 ms	3.83 s

Table 6.1: Latency of JEDI’s implementation of BLS12-381

Table 6.1 compares the performance of JEDI’s BLS12-381 implementation on the three platforms, with our assembly optimizations. As expected from Figure 6.1, the Raspberry Pi performance is an order of magnitude slower than Laptop performance, and performance on the Hamilton sensor is an additional two-to-three orders of magnitude slower.

6.7.1.2 Performance of WKD-IBE in JEDI

Figure 6.6 depicts the performance of JEDI’s cryptography primitives. Figure 6.6 does not include the sensor platform; Section 6.7.3 thoroughly treats performance of JEDI on low-power sensors.

In Figure 6.6a, we used a pattern of length 20 for all operations, which would correspond to, e.g., a URI of length 14 and an Expiry hierarchy of depth 6. To measure decryption and signing time, we measure the time to decrypt the ciphertext or sign the message, plus the time to generate a decryption key for that pattern or ID. For example, if one receives a message on a/b/c/d/e/f, but has the key for a/*, he must generate the key for a/b/c/d/e/f to decrypt it.

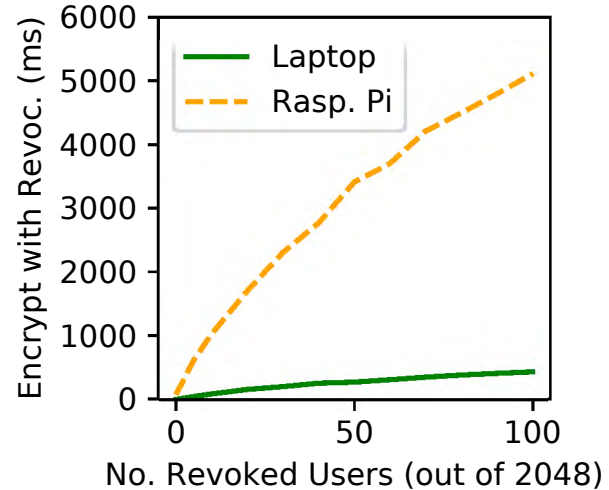
Figure 6.6a demonstrates that the JEDI encrypts and signs messages and generates qualified keys for delegation at practical speeds. On a laptop, all WKD-IBE operations take less than 10 ms with

up to 20 attributes. On a Raspberry Pi, they are 10x slower (as expected), but still run at interactive speeds.

6.7.1.3 Performance of Immediate Revocation in JEDI

	Laptop	Rasp. Pi
Enc.	3.08 ms	37.3 ms
Dec.	3.61 ms	43.9 ms
KeyD.	4.77 ms	58.5 ms
Sign	4.80 ms	61.2 ms
Verify	4.78 ms	56.3 ms

(a) Latency of Encrypt, Decrypt, KeyDer, Sign, and Verify with 20 attributes



(b) Encryption with Revocation

Figure 6.6: Performance of JEDI’s cryptography

Figure 6.6b shows the cost of JEDI’s immediate revocation protocol (Section 6.5). A private key containing k leaves consists of $O(\log k + \log n)$ WKD-IBE secret keys where n is the total number of leaves. Therefore, the performance of immediate revocation depends primarily on the number of leaves.

To encrypt a message, one WKD-IBE encryption is performed for each subtree needed to cover all unrevoked leaves. In general, encryption is $O(r \log \frac{n}{r})$, where r is the number of revoked leaves. Each key contains a set of *consecutive* leaves, so encryption is also $O(R \log \frac{n}{R})$, where R is the number of revoked JEDI keys. Decryption time remains almost the same, since only one WKD-IBE decryption is needed.

To benchmark revocation, we use a complete binary tree of depth 16 ($n = 65536$). The time to generate a new key for delegation is essentially independent of the number of leaves conveyed in that key, because $\log k \ll \log n$. We empirically confirmed this; the time to generate a key for delegation was constant at 2.4 ms on a laptop and 31 ms on a Raspberry Pi as the number of leaves in the key was varied from 5 to 1,000.

To benchmark encryption with revocation, we assume that there exist 2,048 users in the system each with 32 leaves. We measure encryption time with a pattern with 20 fixed slots (for URI and time) as we vary the number of revoked users. Figure 6.6b shows that encryption becomes expensive when the revocation list is large (500 milliseconds on laptop and ≈ 5 seconds on Raspberry Pi). However, such an encryption only needs to be performed by a publisher when the URI, time, or

revocation list changes; subsequent messages can reuse the underlying symmetric key (Section 6.5.5). Furthermore, the revocation list includes only revoked keys that match the (URI, time) pair being used, so it is not expected to grow very large.

6.7.2 Performance of JEDI in bw2

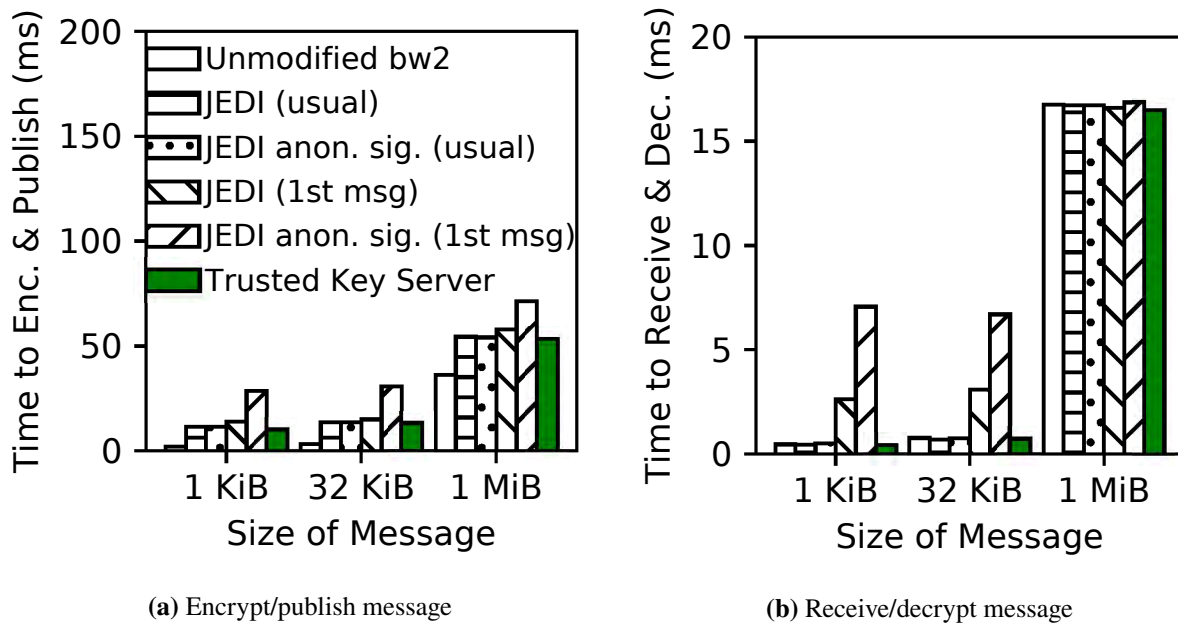


Figure 6.7: Critical-path operations in bw2, with/without JEDI

In bw2, the two critical-path operations are publishing a message to a URI, and receiving a message as part of a subscription. We measure the overhead of JEDI for these operations because they are core to bw2’s functionality and would be used by any messaging application built on bw2. Our methodology is to perform each operation repeatedly in a loop, to measure the sustained performance (operations/second), and report the average time per operation (inverse). To minimize the effect of the network, the router was on the same link as the client, and the link capacity was 1 Gbit/s. In our experiments, we used a URI of length 6 and an Expiry tree of depth 6. We also include measurements from a strawman system with pre-shared AES keys—this represents the critical-path overhead of an approach based on the Trusted Key Server discussed in Section 6.2. Our results are in Figure 6.7.

We implement the optimizations in Section 6.3.6.1, so only symmetric key encryption/decryption must be performed in the common case (labeled “usual” in the diagram). However, the symmetric keys will *not* be cached for the first message sent every hour, when the WKD-IBE pattern changes. A WKD-IBE operation must be performed in this case (labeled “1st message” in the diagram). For large messages, the cost of symmetric key encryption dominates. JEDI has a particularly small overhead for 1 MiB messages in Figure 6.7b, perhaps because 1 MiB messages take several

milliseconds to transmit over the network, allowing the client to decrypt a message while the router is sending the next message.

We also consider creating DOTs and initiating subscriptions, which are not in the critical path of bw2. These results are in Figure 6.8 (note the log scale in Figure 6.8a). Creating DOTs is slower with JEDI, because WKD-IBE keys are generated and included in the DOT. Initiating a subscription in bw2 requires forming a DOT chain; in JEDI, one must also derive a private key from the DOT chain. Figure 6.8a shows the time to form a short one-hop DOT chain, and in the case of JEDI, includes the time to derive the private key. For JEDI’s encryption (Section 6.3), these additional costs are incurred only by DOTs that grant permission to subscribe. With anonymous signatures, DOTs granting permission to publish incur this overhead as well, as WKD-IBE keys must be included. Figure 6.8b puts this in context by measuring the end-to-end latency from initiating a subscription to receiving the first message (measured using bw2’s “query” functionality).

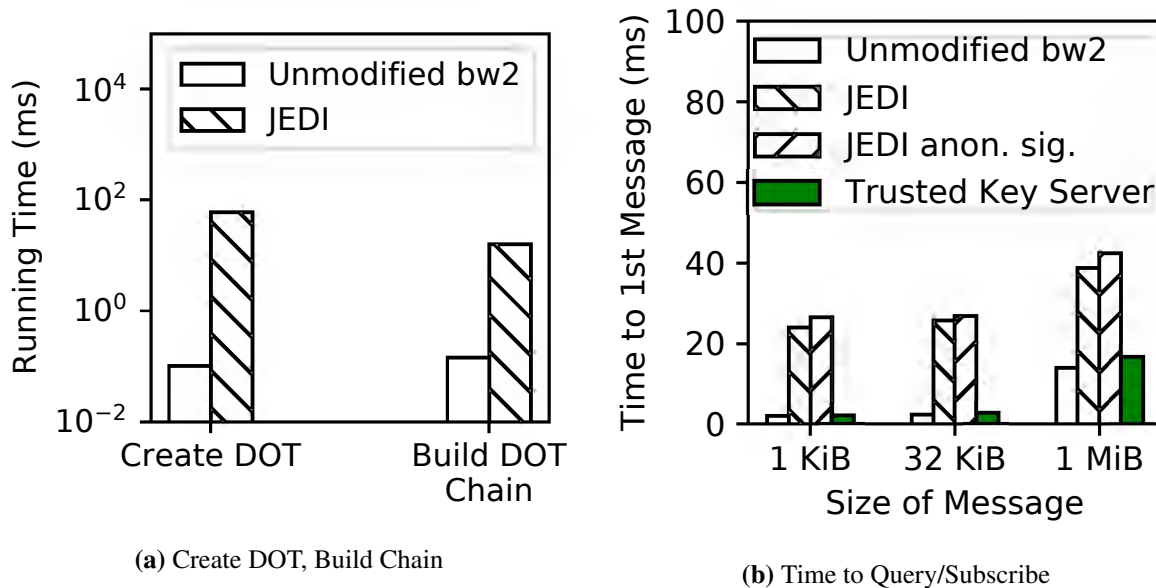


Figure 6.8: Occasional bw2 operations, with and without JEDI

For a DOT to be usable, it must be inserted into bw2’s registry. This requires a blockchain transaction (not included in Figure 6.8). An important consideration in this regard is *size*. In the unmodified bw2 system, a DOT that grants permission on a/b/c/d/e/f is 198 bytes. With JEDI, each DOT also contains multiple WKD-IBE keys, according to the time range. In the “worst case,” where the start time of a DOT is Jan 01 at 01:00, and the end time is Dec 31 at 22:59, a total of 45 keys are needed. Each key is approximately 1 KiB, so the size of this DOT is approximately 45 KiB.

Because bw2’s registry of DOTs is implemented using blockchain smart contracts, the bandwidth for inserting DOTs is limited. Using JEDI would increase the size of DOTs as above, resulting in an approximately 100-400x decrease in aggregate bandwidth for creating DOTs. However, this can be mitigated by changing bw2 to not store DOTs directly in the blockchain. DOTs can be stored in

Table 6.2: CPU and power costs on the Hamilton platform

Operation	Time	Average Current
Sleep (Idle)	N/A	0.0063 mA
WKD-IBE Encrypt	6.50 s	10.2 mA
WKD-IBE Encrypt and Sign	9.89 s	10.2 mA

untrusted storage, with only their hashes stored in the blockchain-based registry. Such a solution could be based on Swarm [TFJ16] or Filecoin [Fil].

6.7.3 Feasibility on Ultra Low-Power Devices

We use a commercially available sensor platform called “Hamilton” [Ham; AKC17] built around the Atmel SAMR21 system-on-chip (SoC). The SAMR21 costs approximately \$2.25 per unit [Ele] and integrates a low-power microcontroller and radio. The sensor platform we used in this study costs \$18 to manufacture [Kim+18]. For battery lifetime calculations, we assume that the platform is powered using a CR123A Lithium battery that provides 1400 mAh at 3.0 V (252 J of energy). Such a battery costs \$1. The SAMR21 is heavily constrained: it has only a 48 MHz CPU frequency based on the ARM Cortex-M0+ microarchitecture, and a total of only 32 KiB of data memory (RAM). Our goal is to validate that JEDI is practical for an ultra low-power sensor platform like Hamilton, in the context of a “sense-and-send” application in a smart building. Since most of the platform’s cost (\$18) comes from the on-board transducers and assembly, rather than the SAMR21 SoC, *using an even more resource-constrained SoC would not significantly decrease the platform’s cost*. An analogous argument applies to energy consumption, as the transducers account for more than half of Hamilton’s idle current [Kim+18].

Hamilton/SAMR21 is on the lower end of platforms typically used for sense-and-send applications in buildings. Some older studies [Fel09; LLLMSL14] use even more constrained hardware like the TelosB; this is because those studies were constrained by hardware available at the time. Modern 32-bit SoCs, like the SAMR21, offer substantially better performance at a similar price/power point to those older platforms [Kim+18].

6.7.3.1 CPU Usage

Table 6.2 shows the time for encryption and anonymous signing in JEDI on Hamilton. The results use the optimizations discussed in Section 6.3.6 and Section 6.4.3, and include the time to “adjust” precomputed state. They indicate that symmetric keys can be encrypted and anonymously signed in less than 10 seconds. This is feasible given that encryption and anonymous signing occur rarely, once an hour, and need not be produced at interactive speeds in the normal “sense-and-send” use case.

Table 6.3: Average current and expected battery life (for 1400 mAh battery) for sense-and-send, with varying sample interval

	AES Only	JEDI (enc)	JEDI (enc & sign)
10 s	32 μ A / 5.1 y	50 μ A / 3.2 y	60 μ A / 2.6 y
20 s	20 μ A / 8.1 y	38 μ A / 4.2 y	48 μ A / 3.3 y
30 s	15 μ A / 10 y	34 μ A / 4.7 y	44 μ A / 3.6 y

6.7.3.2 Power Consumption

To calculate the impact on battery lifetime, we consider a “sense-and-send” application, in which the Hamilton device obtains readings from its sensors at regular intervals, and immediately sends the readings encrypted over the wireless network. We measured the average current consumed for varying sample intervals, when each message is encrypted with AES-CCM, without using JEDI (“AES Only” in Table 6.3). We estimate JEDI’s average current based on the current, duration, and frequency (once per hour, for these estimates) of JEDI operations, and add it to the average current of the “AES Only” setup. Our estimates assume that the μ TESLA-based technique in Section 6.4.3 is used to avoid attaching a digital signature to each message. We divide the battery’s energy capacity by the result to compute lifetime. As shown in Table 6.3, JEDI decreases battery life by about 40-60%. Battery life is several years even with JEDI, acceptable for IoT sensor platforms.

JEDI’s overhead depends primarily on the granularity of expiry times (one hour, for these estimates), *not* the sample interval. To improve power consumption, one could use a time tree with larger leaves, allowing principals to perform WKD-IBE encryptions and anonymous signatures less often. This would, of course, make expiry times coarser.

6.7.3.3 Memory Budget

Performing WKD-IBE operations requires only 6.5 KiB of data memory, which fits comfortably within the 32 KiB of data memory (RAM) available on the SAMR21. The code space required for our implementation of WKD-IBE and BLS12-381 is about 74 KiB, which fits comfortably in the 256 KiB of code memory (ROM) provided by the SAMR21.

A related question is whether storing a hash chain in memory (as required for authenticated broadcast, Section 6.4.3) is practical. If we use a granularity of 1 minute for authenticated broadcast, the length of the hash chain is 60. At the start of an hour, one computes the entire chain, storing 10 hashes equally spaced along the chain, each separated by 5 hashes. As one progresses along the hash chain, one re-computes each set of 5 hashes one additional time. This requires storage for only 15 hashes (< 4 KiB memory) and computation of only 105 hashes *per hour*, which is practical. One could possibly optimize performance further using *hierarchical hash chains* [HJP05].

6.7.3.4 Impact of JEDI’s Optimizations

JEDI’s cryptographic optimizations (Section 6.3.6.2, Section 6.4.2.2, Section 6.4.3), which use WKD-IBE in a non-black-box manner, provide a 2-3x performance improvement. Our assembly

optimizations (Section 6.6) provide an additional 4-5x improvement. Without both of these techniques, JEDI would not be practical on low-power sensors. Hybrid encryption and key reuse (Section 6.3.6.1), which let JEDI use WKD-IBE *rarely*, are also crucial.

6.7.4 Comparison to Other Systems

Crypto Scheme / System	Avoids Central Trust?	Expressivity	Performance
Trusted Key Server (Section 6.2)	- No	+ Supports arbitrary policies (beyond hierarchies) - No delegation	+ $\approx 10 \mu\text{s}$ to encrypt 1 KiB message (same as JEDI in common case, faster for first message after key rotation) - Trusted party generates one key <i>per resource</i>
PRE (Lattice-Based), as used in PICADOR [BGPRR17]	- No	+ Supports arbitrary policies (beyond hierarchies) - No delegation	+ $\approx 5 \text{ ms}$ encrypt, $\approx 3 \text{ ms}$ decrypt (similar to JEDI: 3-4 ms) - Trusted party must generate one key per sender-receiver pair
PRE (Pairing-Based), as used in Pilatus [SHBFD17]	+ Yes	- Delegation is single-hop - Delegation is coarse (all-or-nothing) + Can compute aggregates on encrypted data	+ 0.6 ms encrypt, 1.3 ms re-encrypt, 0.5 ms decrypt (faster than JEDI: 3-4 ms) + Practical on constrained IoT device with crypto accelerator
CP-ABE [BSW07]	+ Yes	+ Good fit for RBAC policies - Cannot support JEDI's hierarchy abstraction with delegation	+ Only symmetric crypto in common case - 14 ms encrypt for first time after key rotation (4-5x slower than JEDI: 3 ms)

KP-ABE, as used in Sieve [WMZV16]	+ Yes	+ Succinct delegation based on attributes - Delegation is single-hop	+ Only symmetric crypto in common case - 25 ms encrypt for first time after key rotation (8-9x slower than JEDI: 3 ms)
Delegable Large Univ. KP-ABE [GPSW06] (used in Alternative JEDI Design)	+ Yes	+ Generalizes beyond hierarchies and supports multi-hop delegation (subsumes JEDI)	+ Only symmetric crypto in common case - 60 ms encrypt for first time after key rotation (20x slower than JEDI: 3 ms) - Impractical for low-power sense-and-send
This paper: WKD-IBE [AKN] with Optimizations, as used in JEDI	+ Yes	+ Delegation is multi-hop + Succinct delegation of <i>sub-trees</i> of resources (or more complex sets, Section 6.3.7) + Non-interactive expiry	+ After key rotation (e.g., once per hour), 3 ms encrypt, 4 ms decrypt (Figure 6.6a) + Only symmetric crypto in common case + Practical for ultra low-power “sense-and-send” <i>without crypto accelerator</i>

Table 6.4: Comparison of JEDI with other crypto-based IoT/cloud systems

Table 6.4 compares JEDI to other systems and cryptographic approaches, particularly those geared toward IoT, in regard to security, expressivity and performance. We treat these existing systems as they would be used in a messaging system for smart buildings (Section 6.1). Table 6.4 contains quantitative comparisons to the cryptography used by these systems; for those schemes based on bilinear groups, we re-implemented them using our JEDI crypto library (Section 6.6.1) for a fair comparison.

Security. The owner of a resource is considered *trusted* for that resource, in the sense that an adversary who compromises a principal can read all of that principal’s resources. In Table 6.4, we focus on whether a single component is trusted for *all* resources in the system. Note that, although Trusted Key Server (Section 6.2) and PICADOR [BGPRR17] encrypt data in flight, granting or revoking access to a principal requires participation of an *online trusted party* to generate new keys.

Expressivity. PRE-based approaches, which associate public keys with users and support delegation via proxy re-encryption, are fundamentally coarse-grained—a re-encryption key allows *all* of a user’s data to be re-encrypted. PICADOR [BGPRR17] allows more fine-grained semantics, but does

not enforce them cryptographically. ABE-based approaches typically do not support delegation beyond a single hop, whereas JEDI achieves multi-hop delegation. In ABE-based schemes, however, attributes/policies attached to keys can describe more complex sets of resources than JEDI. That said, a hierarchical resource representation is sufficient for JEDI's intended use case, namely smart cities; existing syndication systems for smart cities, which do not encrypt data and are unconstrained by the expressiveness of crypto schemes, choose a hierarchical rather than attribute-based representation (Section 6.1).

Performance. The Trusted Key Server (Section 6.2) is the most naïve approach, requiring an online trusted party to enforce all policy. Even so, JEDI's performance in the common case is the same as the Trusted Key Server (Figure 6.7), because of JEDI's hybrid encryption—JEDI invokes WKD-IBE *rarely*. Even when JEDI invokes WKD-IBE, its performance is not significantly worse than PRE-based approaches. An alternative design for JEDI uses the GPSW KP-ABE construction instead of WKD-IBE, but it is significantly more expensive. Based Table 6.3, the power cost of a WKD-IBE operation *even when only invoked once per hour* contributes significantly to the overall energy consumption on the low-power IoT device; using KP-ABE instead of WKD-IBE would increase this power consumption by an order of magnitude, reducing battery life significantly.

In summary, existing systems fall into one of three categories. (1) The Trusted Key Server allows access to resources to be managed by arbitrary policies, but relies on a *central trusted party* who must be online whenever a user is granted access or is revoked. (2) PRE-based approaches, which permit sharing via re-encryption, cannot cryptographically enforce fine-grained policies or support multi-hop delegation. (3) ABE-based approaches, if carefully designed, *can* achieve the same expressivity as JEDI, but are substantially less performant and are not suitable for low-power embedded devices.

6.8 Related Work

We organize related work into the following categories.

Traditional Public-Key Encryption. SiRiUS [GSMB03b] and Plutus [KRSWF03] are encrypted filesystems based on traditional public-key cryptography, but they do not support delegable and qualifiable keys like JEDI. Akl et al. [AT83] and further work [CFGJP15; CMW06] propose using key assignment schemes for access control in a hierarchy. A line of work [Tze02; HC04; ABF07; AFB05] builds on this idea to support both hierarchical structure and temporal access. Key assignment approaches, however, require the full hierarchy to be known at setup time, which is not flexible in the IoT setting. JEDI does not require this, allowing different subtrees of the hierarchy to be managed separately (Section 6.1.1, “Delegation”).

Identity-Based Encryption. Tariq et al. [TKR14] use Identity-Based Encryption (IBE) [BF01] to achieve end-to-end encryption in publish-subscribe systems, without the router’s participation in the protocol. However, their approach does not support hierarchical resources. Further, encryption and private keys are on a credential-basis, so each message is encrypted multiple times according to the credentials of the recipients.

Wu et al. [WTSB16] use a prefix encryption scheme based on IBE for mutual authentication in IoT. Their prefix encryption scheme is different from JEDI, in that users with keys for identity $a/b/c$ can decrypt messages encrypted with prefix identity a , a/b and $a/b/c$, but not identities like $a/b/c/d$.

Hierarchical Identity-Based Encryption. Since the original proposal of Hierarchical Identity-Based Encryption (HIBE) [GS02], there have been multiple HIBE constructions [GS02; BB04; BBG05; GH09] and variants of HIBE [YFDL04; AKN]. Although seemingly a good match for resource hierarchies, HIBE cannot be used as a black box to efficiently instantiate JEDI. We considered alternative designs of JEDI based on existing variants of HIBE, but as we elaborate in the appendix of our extended paper [KHAPC19], each resulting design is either less expressive or significantly more expensive than JEDI.

Attribute-Based Encryption. A line of work [YWRL10; WMZV16] uses Attribute-Based Encryption (ABE) [GPSW06; BSW07] to delegate permission. For example, Yu et al. [YWRL10] and Sieve [WMZV16] use Key-Policy ABE (KP-ABE) [GPSW06] to control which principals have access to encrypted data in the cloud. Some of these approaches also provide a means to revoke users, leveraging proxy re-encryption to safely perform re-encryption in the cloud. Our work additionally supports hierarchically-organized resources and decentralized delegation of keys, which [YWRL10] and [WMZV16] do not address. As discussed in Section 6.7.4, WKD-IBE is substantially more efficient than KP-ABE and provides enough functionality for JEDI. WKD-IBE could be a lightweight alternative to KP-ABE for some applications.

Other approaches prefer Ciphertext-Policy ABE (CP-ABE) [BSW07]. Existing work [WLW10; WLWG11] combines HIBE with CP-ABE to produce Hierarchical ABE (HABE), a solution for sharing data on untrusted cloud servers. The “hierarchical” nature of HABE, however, corresponds to the hierarchical organization of domain managers in an enterprise, not a hierarchical organization of *resources* as in our work.

Proxy Re-Encryption. NuCypher KMS [EW17] allows a user to store data in the cloud encrypted

under her public key, and share it with another user using Proxy Re-Encryption (PRE) [BBS98]. While NuCypher assumes limited collusion among cloud servers and recipients (e.g., m of n secret sharing) to achieve properties such as expiry, JEDI enforces expiry via cryptography, and therefore remains secure against *any* amount of collusion. Furthermore, NuCypher's solution for resource hierarchies requires a keypair for each node in the hierarchy, meaning that the creation of resources is centralized. Finally, keys in NuCypher are not qualifiable. Given a key for $a/*$, one cannot generate a key for $a/b/*$ to give to another principal.

PICADOR [BGPRR17], a publish-subscribe system with end-to-end encryption, uses a lattice-based PRE scheme. However, PICADOR requires a central Policy Authority to specify access control, by creating a re-encryption key for every permitted pair of publisher and subscriber. In contrast, JEDI's access control is decentralized.

Revocation Schemes. Broadcast encryption (BE) [NNL01; DF02; BGW05; BW06; LSW10; BWZ14; BZ17] is a mechanism to achieve revocation, by encrypting messages such that they are only decryptable by a specific set of users. However, these existing schemes do not support key qualification and delegation, and therefore, cannot be used in JEDI directly. Another line of work builds revocation directly into the underlying cryptography primitive, achieving Revocable IBE [BGK08; LV09; SE13b; WES17], Revocable HIBE [SE13a; SE15; LLWQNF] and Revocable KP-ABE [AI09]. These papers use a notion of revocation in which URIs are revoked. In contrast, JEDI supports revocation at the level of keys. If multiple principals have access to a URI, and one of their keys is revoked, then the other principal can still use its key to access the resource. Some systems [EW17; BCAR18] rely on the participation of servers or routers to achieve revocation.

Secure Reliable Multicast Protocol. Secure Reliable Multicast [MMR00; MR97] also uses a many-to-many communication model, and ensures correct data transfer in the presence of malicious routers. JEDI, as a protocol to *encrypt* messages, is complementary to those systems.

Authorization Services. JEDI is complementary to authorization services for IoT, such as bw2 [AKCCK17], Vanadium [TS16], WAVE [And+19], and AoT [Net+16], which focus on expressing authorization policies and enabling principals to prove they are authorized, rather than on encrypting data. Droplet [SBDHR18] provides encryption for IoT, but does not support delegation beyond one hop and does not provide hierarchical resources.

An authorization service that provides secure in-band permission exchange, like WAVE [And+19], can be used for key distribution in JEDI. JEDI can craft keys with various permissions, while WAVE can distribute them without a centralized party by including them in its attestations.

Chapter 7

Non-Interactive Differentially Anonymous Router

Anonymous routing is designed to protect privacy for communication and plays a fundamental role in online applications. Almost all existing approaches for anonymous routing (e.g., mix-nets, DC-nets, and others) rely on some *interactive* protocol among multiple servers or routers; and their anonymity is guaranteed in the *threshold model*, i.e., they must assume that one or more of the servers/routers behave honestly.

A recent work by Shi and Wu (Eurocrypt'21) suggested a new, non-interactive abstraction for anonymous routing, coined Non-Interactive Anonymous Router (NIAR). They show how to construct a NIAR scheme with succinct communication from bilinear groups. Unfortunately, the router needs to perform quadratic computation (in the number of senders/receivers) to perform each routing.

In this paper, we show that if one is willing to relax the security notion to (ϵ, δ) -differential privacy, henceforth also called (ϵ, δ) -differential anonymity, then, a non-interactive construction exists with subquadratic router computation, also assuming standard hardness assumptions in bilinear groups. Moreover, even when $1 - 1/\text{poly} \log n$ fraction of the senders are corrupt, we can attain strong privacy parameters where $\epsilon = O(1/\text{poly} \log n)$ and $\delta = \text{negl}(n)$.

This work was previously published in [BHMS21].

7.1 Introduction

Anonymous communication systems allow users to communicate in an insecure channel without leaking information about their identities or message contents. Interest in anonymous communication systems has increased in recent years because they promise a foundation for building anonymous data-sharing systems [BHKP16; HOWW19b; HKP20] and anonymous cryptocurrency systems [CFN90; Cha82; BS+14; HABSG17; HMPS14; RMSK14; RMSK17; Dia21; LYKGKM19]. Various abstractions and techniques for anonymous communication systems have been proposed [DD08; EY09; SSABD18], including mix-nets [Cha81b; Abe99; BG12], the Dining Cryptographers’ nets [Cha88; CGF10; APY20], onion routing [DMS04b; GRS99], multi-server PIR-write [CBM15; OS97; GIKM00], as well as other variants [ZZZR05; HLZZ15b; TGLZZ17b]. Notably, almost all of these anonymous routing schemes are *interactive* in nature, where multiple players or routers engage in some interactive protocol to accomplish the routing. Further, security typically relies on *threshold*-type assumptions, e.g., either majority or at least one of the routers must be honest for security to hold.

Interestingly, the very recent work by Shi and Wu [SW21b] suggest a new, non-interactive abstraction called Non-Interactive Anonymous Router (NIAR), where routing is accomplished on a *single, untrusted* router. In their scheme, there are n senders and n receivers, and each sender wants to talk to a unique receiver. A trusted setup takes in the routing permutation π and generates secret sender and receiver keys for everyone, as well as a token tk for the untrusted router that secretly “encrypts” the routing permutation π . From this moment on, the n senders and n receivers can engage in multiple rounds of communication. In each time step, each sender uses its sender key to encrypt its message. When the router collects all n ciphertexts from the senders, it can apply the token tk and transform the n incoming ciphertexts into n *transformed and permuted* ciphertexts. At this moment, the receivers can each use their receiver key to decrypt its corresponding transformed ciphertext. Importantly, the router is applying the permutation π encoded in the token tk in an oblivious manner without learning what π is.

Shi and Wu show how to construct such a NIAR scheme assuming standard bilinear group assumptions. Their scheme achieves succinct communication: the total communication in each time step is only linear in n , i.e., each sender or receiver sends or receives only $O(1)$ amount of data. Unfortunately, although their NIAR scheme guarantees succinctness in communication, it is not so succinct in terms of router computation. Specifically, the router needs to perform $\Theta(n^2)$ computation to perform the routing in each time step. In this paper, we raise the following natural question:

Can we achieve privacy-preserving non-interactive routing with sub-quadratic router computation?

7.1.1 Our Results and Contributions

We show that if we are willing to relax the security notion to differential privacy [DMNS06], then indeed, we can achieve sub-quadratic computation. We call our construction Non-Interactive Differentially Anonymous Router (NIDAR). A NIDAR scheme has exactly the same syntax as the

NIAR scheme of Shi and Wu [SW21b], except that we now define a more relaxed security notion called (ϵ, δ) -computational differential anonymity (CDA), i.e., the analogy of (ϵ, δ) -differential privacy in the context of anonymous routing.

To define differential anonymity, we first define a notion of neighboring for permutations. Two permutations on the domain $[n]$ are said to be *neighboring*, iff their only difference is that two honest senders' destinations are swapped. Informally speaking, a NIDAR scheme is said to be differentially anonymous, iff no computationally bounded adversary can “effectively” distinguish two neighboring routing permutations. The actual definition is a little more technical since we need to take into account the fact that the adversary can corrupt a subset of the senders and receivers. Like in the work of Shi and Wu [SW21b], we assume that each sender knows their own destinations, so if the adversary corrupts some senders, it will learn their destinations. Further, if the adversary corrupts a subset of the receivers, it naturally learns the messages received by the corrupt receivers in each time step, and this is inherent. Therefore, in our actual definition of (ϵ, δ) -CDA which is formalized in Section 7.2.2, we effectively consider two worlds. The only difference between the two worlds is that an honest pair of senders swap their destinations. Our security definition requires that for a computationally bounded adversary controlling the router and a subset of corrupt senders and receivers, these two worlds are (ϵ, δ) -indistinguishable (by the standard distance notion in the differential privacy literature [DMNS06; DR14; Vad17]), as long as the following conditions are respected: 1) each corrupt sender has the same receiver in both worlds, and 2) each corrupt receiver always receives the same message in both worlds. Note that these conditions are necessary to make sure that the adversary cannot trivially distinguish between the two worlds.

Specifically, we prove the following theorem — we will first give the version with the most general parameters, and then we will help the reader interpret the parameters with some typical choices.

Theorem 7.1.1. *Let $\rho \in (0, 1)$ be the fraction of corrupt senders. Fix any desired $\delta \in (0, 1)$ and any constant $L = O(1)$, fix any Z such that $(1 - \rho)Z \geq C \cdot (\log \frac{1}{\delta} + \log n)$ where C is a sufficiently large constant. Then, assuming the hardness of the Decisional Linear assumption in suitable bilinear groups, there exists a NIDAR scheme that satisfy (ϵ, δ) -CDA for $\epsilon = O(1) \cdot \frac{\sqrt{Z \cdot \min((1-\rho), \rho)(\log \frac{1}{\delta} + \log n) + (\log \frac{1}{\delta} + \log n)}}{(1-\rho)Z}$, that satisfies the following asymptotical performance bounds where κ is a security parameter related to the strength of the hardness assumption:*

- the router's computation per time step is $O(\kappa^L \cdot n^{1+1/L} \cdot Z^{1-1/L})$,
- the per-sender communication and encryption time is $O(k^L \cdot \text{len})$ where len is the length of the plaintext message;
- each sender key is of length $O(\kappa \cdot n^{1/L} \cdot Z^{1-1/L})$; and
- each receiver key is of length $O(\kappa)$.

7.1.1.0.1 Typical parameters choices. We now give a typical example to help the reader interpret the parameters. Below, we use poly_1 and poly_2 to denote potentially different polynomial functions.

Suppose that up to $1 - \frac{1}{\text{poly}_1 \log n}$ fraction of the senders are corrupt, then, there exists a NIDAR scheme that is (ϵ, δ) -CDA for $\epsilon = \frac{1}{\text{poly}_2 \log n}$ and $\delta = \text{negl}(n)$, with the following asymptotical bounds where L denotes an any arbitrary constant and \tilde{O} hides polylogarithmic factors:

- the router's computation per time step is $\tilde{O}(\kappa^L \cdot n^{1+1/L})$,
- the per-sender ciphertext size and encryption time is $O(k^L \cdot \text{len})$ where len is the length of the plaintext message;
- each sender key is of length $\tilde{O}(\kappa \cdot n^{1/L})$; and
- each receiver key is of length $O(\kappa)$.

Intuitively, this means that even when a very large fraction of senders are corrupt, we can still achieve $(\frac{1}{\text{poly} \log n}, \text{negl}(n))$ -CDA. Recall that in the standard differential privacy literature, we typically want $\epsilon = O(1)$ and $\delta = \text{negl}(n)$. Here we achieve something even better because our $\epsilon = o(1)$; and the smaller the ϵ , the more private it is.

7.1.2 Technical Overview

For simplicity, let us first consider a two-layer scheme that makes use of two onion layers of NIAR. We will assume that up to 99% fraction of the senders are corrupt for the time being, although in our technical sections later, we remove this assumption and give the most general parametrization.

7.1.2.0.1 Reducing routing to shuffling. Instead of constructing NIDAR directly, it is sufficient to construct a primitive where the router outputs the transformed ciphertexts destined for the n receivers in a random order, where the permutation is hidden in the (ϵ, δ) -CDA sense. Suppose that π is the actual permutation we want to realize, and π_{mid} is the secret random permutation applied by the shuffler, then the complement permutation π' is defined to be a permutation that satisfies $\pi' \circ \pi_{\text{mid}} = \pi$. We can give the complement permutation π' to the router in the clear, such that after the router applies the shuffler to the incoming ciphertexts, and obtains the randomly permuted and transformed ciphertexts, it can next apply π' to the permuted and transformed ciphertexts. The outcome will be in the order to be received by the n receivers, respectively.

Therefore, below we may focus on how to construct the differentially anonymous shuffler.

7.1.2.0.2 A two-step, matrix permutation algorithm. To overcome the quadratic router computation, we would like to break up the task of permuting n elements to roughly $O(\sqrt{n})$ permutations each of size only $\tilde{O}(\sqrt{n})$. To achieve this, our idea is to arrange the elements in a square matrix, where each entry in the matrix is a bucket of polylogarithmic size. For technical reasons needed later, we need to introduce filler elements. Specifically, assume that initially, each bucket has exactly half real elements and half filler elements. Our goal is to output a random permutation of the real elements.

We perform the permutation in the following two steps:

1. *Row-wise permutations.* We permute each row in the matrix as follows. First, throw each real element into a random bucket (and if the bucket is full, simply abort throwing an overflow exception). Next, for each bucket in the row, pad each empty slot with a random, unconsumed filler element.
2. *Column-wise permutations.* For each column, output a random permutation of the real elements in the column, and throw away all the filler elements in the output.

Now, if we concatenate the outputs of all column-wise permutations, we obtain a permutation of the real elements in the input matrix. This permutation has negligible statistical distance from a uniform random one. In particular, it is not hard to show that if the buckets are of infinite size, then the output permutation would indeed be random. In our case, however, we can prove through standard measure concentration arguments that none of the buckets overflow except with negligible probability, as long as the buckets are polylogarithmic in size.

7.1.2.0.3 Two onion layers of NIAR. Our differentially anonymous shuffler scheme employs two onion layers of NIAR to realize the permutation in the above two-step manner. We invoke a NIAR instance for each row-wise and each column-wise permutation. Each sender will be encrypting two elements, a real element that encodes the message it wants to send, and a filler element that encodes no information but is necessary for security. For each of the real and filler element of the sender, the sender obtains two encryption keys, one corresponding to the column-wise NIAR instance, and one corresponding to the row-wise NIAR instance. When a sender encrypts its real or filler element, the encryption is performed in the reverse order: the inner encryption uses the sender key corresponding to the column-wise NIAR instance, and the inner ciphertext will be encrypted again using the sender key corresponding to the row-wise NIAR instance. The router's routing operation, on the other hand, is done in the forward order: it applies the NIAR tokens corresponding to all row-wise permutations first, and then applies the NIAR tokens corresponding to all column-wise permutations.

7.1.2.0.4 Performance analysis. With the above scheme, the router only needs to perform the routing operation for $O(\sqrt{n})$ many NIAR instances, each of size $\tilde{O}(\sqrt{n})$. Recall that the routing cost of each NIAR instance is quadratic in the size of the instance. Thus, the total routing cost is $\tilde{O}(n^{1.5})$. The two onion layers of encryption, however, incurs an additional blowup: in each layer, the plaintext message is encrypted bit by bit, and each bit encrypts to $O(1)$ bilinear group elements. Therefore, the ciphertext to plaintext ratio in each layer is some security parameter κ that is related to the length of a bilinear group element. With two layers of encryption, we will incur κ^2 blowup in the ciphertext size as well as the router computation.

7.1.2.0.5 Understanding the leakage. The key technical challenge in our proof is how to bound the leakage of the above two-layer onion construction. The issue is that the adversary can learn which buckets the corrupt sender's elements land in during the row-wise permutations, and as we explain below, this leaks a little information about how many honest real elements choose each bucket during the row-wise permutations.

Recall that in our random process, all the real elements choose their destinations first in the row-wise permutations, and then random filler elements are used to pad each bucket to its full capacity. If a bucket has fewer real elements, it will demand more fillers. Since fillers are randomly drawn among the set of real and corrupt fillers belonging to this row, chances are more corrupt fillers will choose that bucket. Since the adversary knows how many corrupt fillers land in each bucket during the row-wise permutations, it has a little leakage on the total number of real elements in that bucket. Since the adversary also knows how many corrupt real elements land in the bucket, it gets a little information about how many honest real elements land in the bucket.

Fortunately, despite this bit of leakage, we can prove that the scheme still satisfies a very strong notion of (ϵ, δ) -differential anonymity. As we explained earlier, our parameters give strong privacy guarantees particularly because we can achieve $\epsilon = 1/\text{poly } \log n = o(1)$ even when $1 - 1/\text{poly } \log n$ fraction of the senders may be corrupt, assuming a negligibly small δ .

7.1.2.0.6 Proof of differential anonymity. To show our scheme differentially anonymous, imprecisely speaking, we need to prove the following statements:

1. the adversary’s view is simulatable, solely based on the leakage how many corrupt fillers go into each bucket — henceforth, we call this leakage the “corrupt filler load vector”; and
2. the corrupt filler load vector is (ϵ, δ) -differentially private for appropriate choices of ϵ and δ .

For the former statement, we go through a sequence of hybrids and rely on the security of the underlying NIAR scheme. Moreover, we need to rewrite the randomized experiment and change the order in which the events are sampled, without changing the nature of the randomized process. The actual proof is somewhat technical, so we defer the detailed explanation to subsequent technical sections.

For proving the second statement, we draw an interesting connection to a *database sampling mechanism* proposed by Chaudhuri and Mishra [CM06]. In their work, they consider a database where each element has an attribute. A random fraction of the database elements are sampled, and the frequency of each attribute is tallied and released. Chaudhuri and Mishra [CM06] show that in this sampling mechanism, the released histogram (i.e., frequency of all attributes) satisfies (ϵ, δ) -differential privacy for reasonable choices of ϵ and δ , as long as no individual attribute is too rare by some technical definition of “rare”.

In our case, we can view the empty slots in each row (after throwing the real elements) as the database elements. The attribute of each element is the bucket it belongs to. Imagine we are sampling k elements where k corresponds to the number of corrupt fillers in this row. The adversary is then able to see the attributes of these sampled k elements. In other words, the adversary can see the number of corrupt fillers that go into each bucket in each row.

Unfortunately, we cannot directly use the analysis of Chaudhuri and Mishra [CM06]. They aim to make their proof work for the most general parameters, and as a result, their parameters are too loose for the special case we are interested in. Furthermore, their analysis makes some undesirable assumptions, e.g., the sampling probability must be at most $1/2$, and in our case, this roughly translates to the requirement that the majority of senders must be honest. Finally, jumping

ahead a little, their analysis also does not exactly match the random process for the multi-layer scheme to be described later.

Instead of using their analysis, we present a new differential privacy analysis that is particularly optimized for the parameters we are interested in. In this way, we avoid the restrictions on the corruption threshold, and our analysis gives tighter bounds on the final ϵ and δ parameters. Again, we defer the detailed proof to the subsequent technical sections.

7.1.2.0.7 Extension: $O(1)$ layers of onions. We can further extend our construction to multiple layers. However, since each onion layer will incur a κ blowup in the ciphertext size, we can only support a constant number of layers.

To formally describe the L -layer scheme where $L = O(1)$, we use a recursive formulation. When the recursion is fully expanded out, it corresponds to routing on an R -way butterfly network $R = O(n^{1/L})$, and each atomic unit in this butterfly network is again a bucket of polylogarithmic size — see Figure 7.1 of Section 7.4 for a graphical illustration. Again, the senders perform encryption in the reverse order of the network, whereas the router’s evaluation is performed in the forward order. We defer the technical details to Section 7.4.

7.1.3 Additional Related Work

Besides our work, differentially anonymous routing was also considered in private messaging systems such as Vuvuzela [HLZZ15b], Stadium [TGLZZ17b], and Karaoke [LGZ18]. All of these prior works, however, are in the interactive setting, whereas we propose a non-interactive abstraction. In this sense, our security definitions are new. A few works on differential private information retrieval [TDG16; AIVG20] are also remotely related to our work, but again their abstractions are incomparable.

7.2 Definitions and Preliminaries

7.2.1 Syntax

A Non-Interactive Differentially Anonymous Router (NIDAR) has the same syntax as a Non-Interactive Anonymous Router (NIAR) which was first proposed by Shi and Wu [SW21b] — the main difference from NIAR is in the security definition which we shall present in Section 7.2.2. Concretely, NIDAR is a cryptographic scheme consisting of the following, possibly randomized algorithms:

- $(\{\mathbf{ek}_u\}_{u \in [n]}, \{\mathbf{rk}_u\}_{u \in [n]}, \mathbf{tk}_\pi) \leftarrow \mathbf{Setup}(1^\lambda, n, \pi, \text{len})$: the trusted **Setup** algorithm takes the security parameter 1^λ , the number of senders/receivers n , a permutation $\pi \in \text{Perm}([n])$ that represents the mapping between the senders and the receivers, and the length of a plaintext message len . The **Setup** algorithm outputs a sender key for each sender denoted $\{\mathbf{ek}_u\}_{u \in [n]}$, a receiver key for each receiver denoted $\{\mathbf{rk}_u\}_{u \in [n]}$, and a token for the router denoted \mathbf{tk}_π .

- $\text{CT}_{u,t} \leftarrow \mathbf{Enc}(\text{ek}_u, x_{u,t}, t)$: sender u uses its sender key ek_u to encrypt the message $x_{u,t}$ where $t \in \mathbb{N}$ denotes the current time step. The \mathbf{Enc} algorithm produces a ciphertext $\text{CT}_{u,t}$.
- $(\text{CT}'_{1,t}, \text{CT}'_{2,t}, \dots, \text{CT}'_{n,t}) \leftarrow \mathbf{Rte}(\text{tk}_\pi, \text{CT}_{1,t}, \text{CT}_{2,t}, \dots, \text{CT}_{n,t})$: the routing algorithm \mathbf{Rte} takes its token tk_π , and n ciphertexts received from the n senders denoted $\text{CT}_{1,t}, \text{CT}_{2,t}, \dots, \text{CT}_{n,t}$, and produces *transformed ciphertexts* $\text{CT}'_{1,t}, \text{CT}'_{2,t}, \dots, \text{CT}'_{n,t}$ where $\text{CT}'_{u,t}$ is destined for the receiver $u \in [n]$.
- $x \leftarrow \mathbf{Dec}(\text{rk}_u, \text{CT}'_{u,t})$: the decryption algorithm \mathbf{Dec} takes a receiver key rk_u , a transformed ciphertext $\text{CT}'_{u,t}$, and outputs a decrypted message x .

In the above formulation, the permutation π is known a-priori at **Setup** time. Once **Setup** has been run, the senders can communicate with the receivers over an unbounded number of time steps t .

7.2.1.0.1 Correctness. Correctness requires that with probability $1 - \text{negl}(\lambda)$, the following holds for any $\lambda \in \mathbb{N}$, any $\text{len} \in \mathbb{N}$, any $(x_1, x_2, \dots, x_n) \in (\{0, 1\}^{\text{len}})^n$, any $t \in \mathbb{N}$, and any permutation $\pi \in \text{Perm}([n])$: let $(\{\text{ek}_u\}_{u \in [n]}, \{\text{rk}_u\}_{u \in [n]}, \text{tk}_\pi) \leftarrow \mathbf{Setup}(1^\lambda, n, \pi, \text{len})$, let $\text{CT}_{u,t} \leftarrow \mathbf{Enc}(\text{ek}_u, x_u, t)$ for $u \in [n]$, let $(\text{CT}'_{1,t}, \text{CT}'_{2,t}, \dots, \text{CT}'_{n,t}) \leftarrow \mathbf{Rte}(\text{tk}_\pi, \text{CT}_{1,t}, \text{CT}_{2,t}, \dots, \text{CT}_{n,t})$, and let $x'_u \leftarrow \mathbf{Dec}(\text{rk}_u, \text{CT}'_{u,t})$ for $u \in [n]$; it must be that

$$x'_{\pi(u)} = x_u \text{ for every } u \in [n].$$

7.2.1.0.2 Communication compactness. We say that a NIDAR scheme satisfies communication compactness, iff the total communication cost per time step is upper bounded by $\text{poly}(\lambda) \cdot O(n) \cdot \text{len}$.

7.2.2 Computational Differential Anonymity

We now define computational differential anonymity (CDA). Consider the following experiment $\text{NIDAR-Expt}^{b, \mathcal{A}}$ parametrized by a bit $b \in \{0, 1\}$:

- $n, \mathcal{K}_S, \mathcal{K}_R, \pi^{(0)}, \pi^{(1)}, \text{len} \leftarrow \mathcal{A}(1^\lambda)$
- $(\{\text{ek}_u\}_{u \in [n]}, \{\text{rk}_u\}_{u \in [n]}, \text{tk}) \leftarrow \mathbf{Setup}(1^\lambda, n, \pi^{(b)}, \text{len})$
- For $t = 1, 2, \dots$:
 - if $t = 1$ then $\{x_{u,t}^{(0)}\}_{u \in \mathcal{H}_S}, \{x_{u,t}^{(1)}\}_{u \in \mathcal{H}_S} \leftarrow \mathcal{A}(\text{tk}, \{\text{ek}_u\}_{u \in \mathcal{K}_S}, \{\text{rk}_u\}_{u \in \mathcal{K}_R})$;
else $\{x_{u,t}^{(0)}\}_{u \in \mathcal{H}_S}, \{x_{u,t}^{(1)}\}_{u \in \mathcal{H}_S} \leftarrow \mathcal{A}(\{\text{CT}_{u,t-1}\}_{u \in \mathcal{H}_S})$;
 - for $u \in \mathcal{H}_S$, $\text{CT}_{u,t} \leftarrow \mathbf{Enc}(\text{ek}_u, x_{u,t}^{(b)}, t)$
- The adversary \mathcal{A} outputs $b' \in \{0, 1\}$, and the experiment also outputs b' .

We say that \mathcal{A} is admissible iff with probability 1, it guarantees that

1. there exist $u \in \mathcal{H}_S$ and $v \in \mathcal{H}_S$ such that

- $u \neq v$;
 - $\pi^{(0)}(u) = \pi^{(1)}(v)$ and $\pi^{(0)}(v) = \pi^{(1)}(u)$;
 - for any $k \in [n]$, where $k \neq u$ and $k \neq v$, $\pi^{(0)}(k) = \pi^{(1)}(k)$;
2. for any $u \in \mathcal{K}_R \cap \pi^{(0)}(\mathcal{H}_S) = \mathcal{K}_R \cap \pi^{(1)}(\mathcal{H}_S)$, $x_{v_0,t}^{(0)} = x_{v_1,t}^{(1)}$ where for $b \in \{0, 1\}$, $v_b := (\pi^{(b)})^{-1}(u)$. In other words, here we require that in the two alternate worlds $b = 0$ or 1 , every corrupt receiver receiving from an honest sender must receive the same message.

Definition 7.2.1 (Computational differential anonymity). *Let $\epsilon > 0$ and $\delta \in (0, 1)$ be functions of the security parameter λ . We say that a NIDAR scheme satisfies (ϵ, δ) -computational differential anonymity (or (ϵ, δ) -CDA for short), iff for every nonuniform p.p.t. adversary \mathcal{A} , for every $\lambda \in \mathbb{N}$, it holds that*

$$\Pr \left[\text{NIDAR-Expt}^{0,\mathcal{A}}(1^\lambda) = 1 \right] \leq e^{\epsilon(\lambda)} \times \Pr \left[\text{NIDAR-Expt}^{1,\mathcal{A}}(1^\lambda) = 1 \right] + \delta(\lambda) .$$

7.2.3 Background on NIAR

In our NIDAR construction, we will use two or more onion layers of NIAR. A NIAR scheme has the same syntax as NIDAR (see Section 7.2.1), but satisfies a stronger, simulation-based security notion as defined below.

We consider static corruption where the set of corrupt players are chosen prior to the **Setup** algorithm.

Real-world experiment $\text{Real}^{\mathcal{A}}(1^\lambda)$. The real-world experiment is described below where $\mathcal{K}_S \subseteq [n]$ denotes the set of corrupt senders, and $\mathcal{K}_R \subseteq [n]$ denotes the set of corrupt receivers. Let $\mathcal{H}_S = [n] \setminus \mathcal{K}_S$ be the set of honest senders and $\mathcal{H}_R = [n] \setminus \mathcal{K}_R$ be the set of honest receivers. Let \mathcal{A} be a *stateful* adversary:

- $n, \pi, \mathcal{K}_S, \mathcal{K}_R, \text{len} \leftarrow \mathcal{A}(1^\lambda)$
- $(\{\text{ek}_u\}_{u \in [n]}, \{\text{rk}_u\}_{u \in [n]}, \text{tk}) \leftarrow \text{Setup}(1^\lambda, n, \pi, \text{len})$
- For $t = 1, 2, \dots$:
 - if $t = 1$ then $\{x_{u,t}\}_{u \in \mathcal{H}_S} \leftarrow \mathcal{A}(\text{tk}, \{\text{ek}_u\}_{u \in \mathcal{K}_S}, \{\text{rk}_u\}_{u \in \mathcal{K}_R})$; else $\{x_{u,t}\}_{u \in \mathcal{H}_S} \leftarrow \mathcal{A}(\{\text{CT}_{u,t-1}\}_{u \in \mathcal{H}_S})$;
 - for $u \in \mathcal{H}_S$, $\text{CT}_{u,t} \leftarrow \text{Enc}(\text{ek}_u, x_{u,t}, t)$

Ideal-world experiment $\text{Ideal}^{\mathcal{A}, \text{Sim}}(1^\lambda)$. The ideal-world experiment involves not just \mathcal{A} , but also a p.p.t. (stateful) simulator denoted Sim , who is in charge of simulating \mathcal{A} 's view knowing essentially only what corrupt senders and receivers know. Further, the $\text{Ideal}^{\mathcal{A}, \text{Sim}}(1^\lambda)$ experiment is parametrized by a leakage function denoted Leak to be defined later. Henceforth for $\mathcal{C} \subseteq [n]$, we use $\pi(\mathcal{C})$ to denote the set $\{\pi(u) : u \in \mathcal{C}\}$.

- $n, \pi, \mathcal{K}_S, \mathcal{K}_R, \text{len} \leftarrow \mathcal{A}(1^\lambda)$

- $(\{\mathbf{ek}_u\}_{u \in [n]}, \{\mathbf{rk}_u\}_{u \in [n]}, \mathbf{tk}) \leftarrow \text{Sim}(1^\lambda, n, \text{len}, \mathcal{K}_S, \mathcal{K}_R, \text{Leak}(\pi, \mathcal{K}_S, \mathcal{K}_R))$
- For $t = 1, 2, \dots$:
 - if $t = 1$ then $\{x_{u,t}\}_{u \in \mathcal{H}_S} \leftarrow \mathcal{A}(\mathbf{tk}, \{\mathbf{ek}_u\}_{u \in \mathcal{K}_S}, \{\mathbf{rk}_u\}_{u \in \mathcal{K}_R})$; else $\{x_{u,t}\}_{u \in \mathcal{H}_S} \leftarrow \mathcal{A}(\{\mathbf{CT}_{u,t-1}\}_{u \in \mathcal{H}_S})$;
 - $\{\mathbf{CT}_{u,t}\}_{u \in \mathcal{H}_S} \leftarrow \text{Sim}(\{\forall u \in \mathcal{K}_R \cap \pi(\mathcal{H}_S) : (u, x_{v,t}) \text{ for } v = \pi^{-1}(u)\})$.

In the above the function $\text{Leak}(\pi, \mathcal{K}_S, \mathcal{K}_R)$ contains the destination of each corrupt sender, as defined below:

$$\text{Leak}(\pi, \mathcal{K}_S, \mathcal{K}_R) := \{\forall u \in \mathcal{K}_S : (u, \pi(u))\}$$

Definition 7.2.2 (NIAR simulation security). *We say that a NIAR scheme satisfies simulation security (with receiver insider protection), iff there exists a p.p.t. simulator Sim such that for any non-uniform p.p.t. adversary \mathcal{A} , \mathcal{A} 's view in $\text{Real}^{\mathcal{A}}(1^\lambda)$ and $\text{Ideal}^{\mathcal{A}, \text{Sim}}(1^\lambda)$ are computationally indistinguishable.*

Note that the above simulation-secure definition is equivalent to $(0, \text{negl}(\lambda))$ -CDA. In particular, Shi and Wu [SW21b] proved that the simulation-based security notion is equivalent to a natural indistinguishability-based definition; and their indistinguishability-based notion is equivalent to $(0, \text{negl}(\lambda))$ -CDA due to a simple hybrid argument, since given any two permutations $\pi^{(0)}$ and $\pi^{(1)}$, we can transform $\pi^{(0)}$ to $\pi^{(1)}$ in polynomially many steps, each time swapping the destinations of two honest senders.

7.3 Two-Layer NIDAR Construction

7.3.1 A Two-Step Permutation Algorithm

We want to anonymously permute n elements. However, recall that a NIAR scheme on n elements would incur at least n^2 computational cost to route n messages in each time step. To reduce the computational cost, our idea is to break up the big permutation on n elements into roughly \sqrt{n} permutations each of size \sqrt{n} .

7.3.1.0.1 Strawman attempt. A strawman attempt is to write the n elements as a matrix — we first permute each row, and then permute each column. Unfortunately, it turns out that this does not result in a random permutation. More specifically, elements originally in the same row must all go to distinct columns. Based on this, a polynomial time adversary could easily distinguish the resulting permutation from a completely random one.

7.3.1.0.2 Our approach. Our approach is inspired by this strawman attempt. However, we make two critical modifications. First, we introduce as many filler elements as there are real elements. Second, we will work on a matrix of buckets, i.e., each entry in the matrix is now a bucket of size Z . We will prove that if Z is at least polylogarithmic in size, then our algorithm produces a permutation

that has negligible statistical distance from a uniform random one. We will describe our `MatrixPerm` algorithm for an input of $2n$ elements rather than n , since later, in our NIDAR scheme, we always invoke `MatrixPerm` on $2n$ elements, where each of the n senders contributes one *real* element and one filler element. For simplicity, the reader may first imagine that the matrix is a square one, i.e., $R = C$ and $R \cdot C \cdot Z = 2n$, although in the case when $2n/Z$ is not a perfect square, R and C do not need to be strictly equal.

Algorithm `MatrixPerm`

7.3.1.0.3 Input. The input \mathbf{I} is an array of $2n$ elements among which at least half are fillers, and the remaining are real elements. View the input as an $(R \times C)$ -sized matrix (also denoted \mathbf{I}) where each entry in the matrix $\mathbf{I}[i, j]$ is a bucket consisting of Z elements, such that *at least half of the elements in each bucket are fillers*. For simplicity, we assume that $R \cdot C \cdot Z = 2n$ (we will explain how to deal with the case when $2n$ is not divisible by $R \cdot Z$ later).

7.3.1.0.4 Algorithm.

1. **Permute rows.** For each row $i \in [R]$, let $\mathbf{I}[i, :] = \text{RowPerm}(\mathbf{I}[i, :])$.
2. **Permute columns.** For each column $j \in [C]$, let $\mathbf{I}[:, j] = \text{ColPerm}(\mathbf{I}[:, j])$.
3. **Output.** For each column $j \in [C]$, output all the real elements in column j in lexicographical ordering of their offset β within the column (and ignore all the filler elements).

Subroutine `RowPerm`

7.3.1.0.5 Input. A list of C buckets each of size Z , and each bucket contains at least half filler elements.

7.3.1.0.6 Algorithm.

1. Initialize C output buckets each of capacity Z . Initially, all output buckets are empty.
2. For each real element in the input list, place it into a random output bucket.
3. If any bucket's load exceeds Z , return overflow; else, for each empty slot in each bucket, choose a random unconsumed filler element from the input array to fill the slot.
4. Randomly permute the elements within each bucket.
5. Return the list of output buckets.

Subroutine ColPerm

7.3.1.0.7 Input. A list of R buckets each of size Z .

7.3.1.0.8 Algorithm.

1. Let X be the list of all real elements contained in the input, and let Y be the list of all filler elements in the input.
2. Return $\text{RandPerm}(X) \parallel \text{RandPerm}(Y)$ where $\text{RandPerm}(\cdot)$ outputs a random permutation of the input array.

Lemma 7.3.1 (MatrixPerm \approx uniform random permutation). *The output of the MatrixPerm(\mathbf{I}) algorithm outputs a permutation of the real elements contained in the input \mathbf{I} , and moreover, the resulting permutation has statistical distance at most $O(n) \cdot \exp(-\Omega(Z))$ from a uniform random permutation.*

Proof. First, pretend that in our algorithm, even if some bucket receives more than Z real elements during the row-wise permutations, we do not abort throwing overflow, but instead continue with the algorithm allowing the buckets to contain arbitrarily many elements. In this case, it is not hard to see that the algorithm must output a uniform random permutation. Therefore, it suffices to prove that the probability of overflow is upper bounded by $O(n) \cdot \exp(-\Omega(Z))$. This follows from a simple application of the Chernoff bound for each fixed bucket, and then taking a union bound over all buckets. \square

7.3.1.0.9 More general parameters. So far, we have assumed that $R \cdot C \cdot Z = 2n$. However, in some cases, $2n$ may not be divisible by $R \cdot Z$. In this case, we may distribute the input elements as evenly as possible across all rows, and as evenly as possible across the buckets in the same row. Later in our application of MatrixPerm, each of the n senders contributes a real and a filler element, and we want that for the same sender, its real and filler elements be assigned to the same bucket in the input. Therefore, when $2n$ is not divisible by $R \cdot Z$, we may assume each row has either Q or $Q + 2$ elements for some Q , and each bucket has either Z elements or $Z + 2$ elements. Like before, we still require that each bucket has at least as many filler elements as there are real elements. Lemma 7.3.1 would still hold for this indivisible case as well.

7.3.2 Two-Layer NIDAR

In our construction, we will implement a permutation using the two-step MatrixPerm algorithm described in Section 7.3.1. Moreover, each sender encrypts one *real* element and one *filler* element, in a two-layer onion fashion as we explain in more detail below. The real element corresponds to the message the sender actually wants to send, whereas the filler element does not encode any useful content, and is only introduced for security. Recall that in MatrixPerm, we divide the input elements,

symbolically denoted $(1R, 1F, 2R, 2F, \dots, nR, nF)$ into a matrix of $R \times C$ buckets, each of size Z — here, for $u \in [n]$, uR denotes sender u 's real element, and uF denotes sender u 's real element. We first randomly permute each row; then, we randomly permute each column while moving all the real elements within each column to the front. In our NIDAR construction, we will invoke an NIAR instance for each of the R row-wise permutations, and similarly, invoke an NIAR instance for each of the C column-wise permutations. The column instances of NIAR will be used to encrypt the actual messages, whereas the row instances of NIAR will be used to encrypt the ciphertexts produced by the column instances, thus creating a two-layer onion.

7.3.2.0.1 Notational conventions. We always use the variable $i \in [R]$ to index into rows, and the variable $j \in [C]$ to index into columns. When we write (i, α) where $i \in [R]$ and $\alpha \in [C \cdot Z]$, we refer to the α -th *element* (as opposed to bucket) of the i -th row, when the C buckets in row i is flattened out as a one-dimensional array. Similarly, when we write (j, β) where $j \in [C]$ and $\beta \in [R \cdot Z]$, we refer to the β -th *element* (as opposed to bucket) of the j -th column. We often use the superscripts “ $-$ ” and “ $|$ ” to differentiate between variables of the row instances and variables of the column instances.

7.3.2.0.2 Our two-layer NIDAR construction. We now describe our two-layer NIDAR construction below.

Two-Layer NIDAR Construction

Parameters: let Z be the bucket size such that $(1 - \rho)Z \geq \Theta(\log \frac{1}{\delta})$ where ρ is the fraction of corrupt senders, and $\Theta(\cdot)$ hides an appropriately large constant. For simplicity, assume that $2n = R \cdot C \cdot Z$ and $R = C$. We will deal with the case when $2n/Z$ is not a perfect square or $2n$ is not divisible by Z later.

7.3.2.0.3 Assume: after the adversary chooses which users to corrupt and before the **Setup** algorithm is first invoked, all senders are randomly permuted, and we renumber the senders from 1 to n after this initial permutation. Throughout the following algorithms, we refer to senders by these randomly renumbered identities.

- **Setup** $(1^\lambda, n, \pi, \text{len})$:

1. *Simulate MatrixPerm.* Simulate a random run of the MatrixPerm algorithm on the symbolic array $(1R, 1F, 2R, 2F, \dots, nR, nF)$, where uR and uF denote the real and filler elements corresponding to sender $u \in [n]$, respectively. Let π_i^- denote the permutation applied to row i in RowPerm, and let $\pi_j^|$ denote the permutation applied to column j in ColPerm. Let π_{mid} be the permutation output by MatrixPerm on the real elements in the input. Let m_1, m_2, \dots, m_C denote the number of real elements in each column after the row-wise permutations. Let len' denote the ciphertext length of a NIAR scheme (with $R \cdot Z$ senders) when the message length is len .

2. *Set up row instances of NIAR.* For each row $i \in [R]$, let

$$\left(\{\mathbf{ek}_{i,\alpha}^-\}_{\alpha \in [C \cdot Z]}, \{\mathbf{rk}_{i,\alpha}^-\}_{\alpha \in [C \cdot Z]}, \mathbf{tk}_i^- \right) \leftarrow \text{NIAR.Setup}(1^\lambda, C \cdot Z, \pi_i^-, \text{len}')$$

3. *Set up column instances of NIAR.* For each column $j \in [C]$, let

$$\left(\{\mathbf{ek}_{j,\beta}^|\}_{\beta \in [R \cdot Z]}, \{\mathbf{rk}_{j,\beta}^|\}_{\beta \in [R \cdot Z]}, \mathbf{tk}_j^| \right) \leftarrow \text{NIAR.Setup}(1^\lambda, R \cdot Z, \pi_j^|, \text{len})$$

4. *Output sender keys.*

- suppose that in the MatrixPerm algorithm earlier, the symbolic elements uR and uF are initially in positions (i, α) and $(i, \alpha + 1)$, respectively^a; moreover, after the row-wise permutations, they are in positions (j, β) and $(\tilde{j}, \tilde{\beta})$, respectively.
- give user u the encryption key $\mathbf{ek}_u := (\mathbf{ek}_{i,\alpha}^-, \mathbf{ek}_{j,\beta}^|, \mathbf{ek}_{i,\alpha+1}^-, \mathbf{ek}_{\tilde{j},\tilde{\beta}}^|)$.

5. *Output receiver keys.* Let $(\mathbf{rk}_1, \dots, \mathbf{rk}_n) := \pi' \left(\{\mathbf{rk}_{j,\beta}^|\}_{j \in [C], \beta \in [m_j]} \right)$ where $\{\mathbf{rk}_{j,\beta}^|\}_{j \in [C], \beta \in [m_j]}$ is flattened to a 1-dimensional array based on lexicographical order of (j, β) .

6. *Output token.* Let π' be the “complement permutation” such that $\pi' \circ \pi_{\text{mid}} = \pi$; output

$$\mathbf{tk} := (\pi', \{m_j\}_{j \in [C]}, \{\mathbf{tk}_i^-\}_{i \in [R]}, \{\mathbf{tk}_j^|\}_{j \in [C]}, \{\mathbf{rk}_{i,\alpha}^-\}_{i \in [R], \alpha \in [C \cdot Z]})$$

• **Enc**($\mathbf{ek}_u, x_{u,t}, t$):

1. parse $\mathbf{ek}_u := (\mathbf{ek}^-, \mathbf{ek}^|, \mathbf{fek}^-, \mathbf{fek}^|)$;
2. let $\mathbf{ict} \leftarrow \text{NIAR.Enc}(\mathbf{ek}^|, x_{u,t}, t)$; and let $\mathbf{ct} \leftarrow \text{NIAR.Enc}(\mathbf{ek}^-, \mathbf{ict}, t)$;
3. let $\tilde{\mathbf{ict}} \leftarrow \text{NIAR.Enc}(\mathbf{fek}^|, \mathbf{0}, t)$; and let $\tilde{\mathbf{ct}} \leftarrow \text{NIAR.Enc}(\mathbf{fek}^-, \tilde{\mathbf{ict}}, t)$; and
4. output $\mathbf{CT} := (\mathbf{ct}, \tilde{\mathbf{ct}})$.

• **Rte**($\mathbf{tk}, \mathbf{CT}_{1,t}, \dots, \mathbf{CT}_{n,t}$):

1. for each $u \in [n]$, parse $\mathbf{CT}_{u,t} := (\mathbf{ct}_{u,t}, \tilde{\mathbf{ct}}_{u,t})$; view $\{\mathbf{CT}_{u,t}\}_{u \in [n]}$ as an $(R \times C)$ -matrix where each entry is a bucket of size Z . Henceforth, we use $\mathbf{CT}[i :]$ to denote the i -th row of this ciphertext matrix.
2. parse $\mathbf{tk} := (\pi', \{m_j\}_{j \in [C]}, \{\mathbf{tk}_i^-\}_{i \in [R]}, \{\mathbf{tk}_j^|\}_{j \in [C]}, \{\mathbf{rk}_{i,\alpha}^-\}_{i \in [R], \alpha \in [C \cdot Z]})$;
3. for each row $i \in [R]$, let $\tilde{\mathbf{ict}}_{i,1}, \dots, \tilde{\mathbf{ict}}_{i,C \cdot Z} \leftarrow \text{NIAR.Rte}(\mathbf{tk}_i^-, \mathbf{CT}[i :])$, and for each $\alpha \in [C \cdot Z]$, let $\mathbf{ict}_{i,\alpha} \leftarrow \text{NIAR.Dec}(\mathbf{rk}_{i,\alpha}^-, \tilde{\mathbf{ict}}_{i,\alpha})$;
4. view $\{\mathbf{ict}_{i,\alpha}\}_{i \in [R], \alpha \in [C \cdot Z]}$ also as a $(R \times C)$ -matrix where each entry is a bucket of size Z — we shall use $\mathbf{ict}[: j]$ to denote the j -th column of this matrix;
5. for each column $j \in [C]$, let $\mathbf{CT}'_{j,1}, \dots, \mathbf{CT}'_{j,R \cdot Z} \leftarrow \text{NIAR.Rte}(\mathbf{tk}_j^|, \mathbf{ict}[: j])$;

6. view $\{\text{CT}'_{j,\beta}\}_{j \in [C], \beta \in [m_j]}$ as an array, apply π' to the array, and output the result.

• **Dec**($\text{rk}_u, \text{CT}'_u$): output **NIAR.Dec**($\text{rk}_u, \text{CT}'_u$).

^aWe may assume that uR and uF must be in adjacent positions in the same row initially.

7.3.2.0.4 More general parameters. So far, we assumed that $R \cdot C \cdot Z = 2n$ and $R = C$. In the more general case, $2n$ may not be divisible by Z , or $2n/Z$ may not be a perfect square. In this case, we can choose $R = \lceil \sqrt{2n/Z} \rceil$, and the above algorithm would still work, as long as when we assign the elements $1R, 1F, \dots, nR, nF$ to the initial matrix of buckets, the following constraints are respected:

1. For any $u \in [n]$, uR and uF are always assigned to the same bucket;
2. All rows' total capacities (of real and filler elements) differ by at most 2;
3. All buckets' total capacities (of real and filler elements) differ by at most 2. In other words, not all buckets are of equal capacity Z ; the capacity could be either Z or $Z + 2$.

The latter two constraints basically says that the loads across all rows and the loads across all buckets within the same row should be as even as possible, subject to the first constraint.

When we have fixed the capacities of all buckets in this $R \times C$ matrix as mentioned above, we can adjust the size parameter of each row-wise and column-wise NIAR instance accordingly. Our proof can easily be modified to make this case work as well.

7.3.2.0.5 Correctness. Fix any security parameter $\lambda \in \mathbb{N}$, any message length $\text{len} \in \mathbb{N}$, any plaintext messages $(x_1, x_2, \dots, x_n) \in (\{0, 1\}^{\text{len}})^n$, any timestamp $t \in \mathbb{N}$, and any permutation $\pi \in \text{Perm}([n])$. Let $(\{\text{ek}_u\}_{u \in [n]}, \{\text{rk}_u\}_{u \in [n]}, \text{tk}_\pi)$ be any key tuples output by the algorithm **Setup**($1^\lambda, n, \pi, \text{len}$), let CT_u be the ciphertext of x_u output by the algorithm **Enc**(ek_u, x_u) for $u \in [n]$, let $(\text{CT}'_1, \text{CT}'_2, \dots, \text{CT}'_n)$ be the shuffled ciphertext output by the algorithm **Rte**($\text{tk}_\pi, \text{CT}_1, \text{CT}_2, \dots, \text{CT}_n$), and let x'_u be the decryption result output by the algorithm **Dec**($\text{rk}_u, \text{CT}'_u$) for $u \in [n]$.

We need to show that $x_u = x'_\pi(u)$ for $u \in [n]$. According to the proof of Lemma 7.3.1, with probability $1 - \text{negl}(\lambda)$, the algorithm **Setup** succeeds in simulating **MatrixPerm** algorithm and produces corresponding key tuples. We first consider the correctness of the row-wise permutation. For a user $u \in [n]$, suppose the symbolic elements uR and uF in the **MatrixPerm** are initially in positions (i, α) and $(i, \alpha + 1)$ ($i \in [R]$ and $\alpha \in [C \cdot Z]$), and are in positions (j, β) and $(\tilde{j}, \tilde{\beta})$ ($j, \tilde{j} \in [C]$ and $\beta, \tilde{\beta} \in [R \cdot Z]$), after the row-wise permutation. The user u receives the encryption key $\text{ek}_{i,\alpha}^-$ and $\text{ek}_{i,\alpha+1}^-$ for the outer encryption in the two-layer onion. Due to the correctness of the NIAR instance for the row i , after running the algorithms **NIAR.Rte** and **NIAR.Dec**, with probability 1, the inner ciphertext of the user u 's real and filler messages will be in positions (j, β) and $(\tilde{j}, \tilde{\beta})$, respectively.

Now, we consider the correctness of the column-wise permutation. Similarly, the user u receives the encryption key $\text{ek}_{j,\beta}^\dagger$ and $\text{ek}_{\tilde{j},\tilde{\beta}}^\dagger$ for the inner encryption in the two-layer onion. Suppose the

symbolic element uR is in the position (j', β') ($j' \in [C]$ and $\beta' \in [m_j]$) after the column-wise permutation. Due to the correctness of the NIAR instance for the column j , after running the algorithm **NIAR.Rte**, with probability 1, the intermediate result of the inner ciphertext will be in the position (j', β') .

Let $\{\text{CT}'_{j,\beta}\}_{j \in [C], \beta \in [m_j]}$ be the array of ciphertexts after the column permutation. Then the $\pi_{\text{mid}}(u)$ -th element is the ciphertext of the user u 's real message and will be send to the receiver $\pi' \circ \pi_{\text{mid}}(u)$. Due to the correctness of the NIAR instance for the column j , after running the algorithm **NIAR.Dec**, with probability 1, the decryption result $x'_{\pi' \circ \pi_{\text{mid}}(u)}$ equals to the message x_u . Because $\pi' \circ \pi_{\text{mid}} = \pi$, we have $x'_{\pi(u)} = x_u$.

7.3.2.0.6 Efficiency. We now analyze the efficiency of our NIDAR scheme assuming that the underlying NIAR is instantiated with the construction of Shi and Wu [SW21b]. We first review the efficiency of the NIAR scheme by Shi and Wu. We will use the notation $O_\lambda(\cdot)$ to hide $\text{poly}(\lambda)$ factors. In the underlying NIAR scheme by Shi and Wu [SW21b], the per-sender ciphertext length and per-sender computation in each time step are upper bounded by $O_\lambda(\text{len})$; each sender's key is at most $O_\lambda(n)$ in size; each receiver's key is $O_\lambda(1)$ in size; the router's token has length $O_\lambda(n^2)$; and finally, the computational overhead for performing the **Rte** operation is $O_\lambda(n^2)$.

In our NIDAR scheme, each sender needs to compute $O(1)$ many NIAR ciphertexts in every time step, which takes $O_\lambda(\text{len})$ amount of time, and moreover, the ciphertext size (per sender) is upper bounded by the same expression. Note that the row instances of NIAR have message lengths that are polynomially larger than the column instances, and this polynomial blowup is accounted for in the $O_\lambda(\cdot)$ notation. It is also not hard to verify that each sender's key is $O_\lambda(R \cdot Z + C \cdot Z) = O_\lambda(\sqrt{nZ})$ in size, and each receiver's key is $O_\lambda(1)$ in size.

We now analyze the computational overhead of the **Rte** operation as well as the router's token size. To perform the **Rte** operation, the router needs to evaluate the underlying NIAR's **Rte** function for R row instances each with $C \cdot Z$ senders, and for C column instances each with $R \cdot Z$ senders. Therefore, the router's total work is upper bounded by $O_\lambda(R \cdot (C \cdot Z)^2 + C \cdot (R \cdot Z)^2) = O_\lambda(\sqrt{n/Z} \cdot (\sqrt{n/Z} \cdot Z)^2) = O_\lambda(n^{\frac{3}{2}} \cdot Z^{\frac{1}{2}})$. Again, the $O_\lambda(\cdot)$ notation accounts for the polynomial blowup in the plaintext size for the row instances. The router's token size is also upper bounded by the same expression, that is, $O_\lambda(n^{\frac{3}{2}} \cdot Z^{\frac{1}{2}})$.

7.3.3 Proofs

Recall that in our NIDAR construction, we first randomly permute all the users at the beginning of **Setup**, and reassign their identities after this random permutation. This random permutation step is there only to make sure that corruption choices are random. Therefore, henceforth in the proof, we may equivalently pretend that the corruption choices are randomly made, and we skip this random permutation step in the algorithm.

Suppose we start out with the symbolic vector $(1R, 1F, 2R, 2F, \dots, nR, nF)$ where each index $u \in [n]$ denotes a user, the letter "R" denotes a real message, and "F" denotes a filler message. A random subset of these users are corrupt. We now view this vector as a matrix of $R \times C$ buckets

each of size Z , and we permute this vector using the MatrixPerm algorithm, where we first apply a row-wise permutation to each row of buckets, and then we apply a column-wise permutation to each column of buckets. Each position in this matrix can be denoted either as (i, α) where $i \in [R]$ and $\alpha \in [C \cdot Z]$ meaning it is the α -th position of the i -th row; or as (j, β) where $j \in [C]$ and $\beta \in [R \cdot Z]$ meaning it is the β -th position of the j -th column.

We will use the following notations to denote the “senders” and “receivers” from the perspective of each NIAR instance:

- **Sources and destinations of the row-wise permutations.** Let $\mathcal{K}_{i,S}^-$ (or $\mathcal{H}_{i,S}^-$, resp.) denote all coordinates (i, α) that correspond to a corrupt (or honest, resp.) element before applying the row-wise permutation.

Note that all destinations in every row-wise permutation are considered corrupt, since the adversary receives all of $\{\text{rk}_{i,\alpha}^-\}_{i \in [R], \alpha \in [C \cdot Z]}$ as part of the token. Therefore, we let $\mathcal{K}_{i,R}^- = \{(i, \alpha)\}_{\alpha \in [C \cdot Z]}$.

- **Sources and destinations of the column-wise permutations.** Let $\mathcal{K}_{j,S}^|$ (or $\mathcal{H}_{j,S}^|$, resp.) denote the all coordinates (j, β) that correspond to corrupt (or honest, resp.) elements sources in j -th column-wise permutation.

Let $\mathcal{K}_{j,R}^|$ (or $\mathcal{H}_{j,R}^|$, resp.) denote the all coordinates (j, β) such that $\beta \leq m_j$, and moreover (j, β) corresponds to a corrupt (or honest, resp.) destination in the j -th column-wise permutation — recall that after the column-wise permutation, only the first m_j coordinates of column j contain real elements, and the adversary receives only the receiver keys (of the column instances) corresponding to the corrupt real destinations.

7.3.3.1 Sequence of Hybrids

7.3.3.1.1 Experiment NIDAR-Expt⁰. Same as the original NIDAR-Expt⁰ experiment as defined in Section 7.2.

7.3.3.1.2 Experiment Hyb₁⁰. Almost the same as NIDAR-Expt⁰ except that we replace each the column instance of NIAR with a NIAR simulator. Recall that the NIAR’s simulator only needs to know the destinations of all corrupt sources, as well as what message each corrupt destination receives in each time step. We describe the modifications from NIDAR-Expt⁰ below, where we use $\text{Sim}_j^|$ to denote the (stateful) NIAR simulator corresponding to the j -th column instance:

1. During **Setup** $(1^\lambda, n, \pi^{(0)}, \text{len})$,

instead of calling $(\{\text{ek}_{j,\beta}^|\}_{\beta \in [R \cdot Z]}, \{\text{rk}_{j,\beta}^|\}_{\beta \in [R \cdot Z]}, \text{tk}_j^|) \leftarrow \text{NIAR.Setup}(1^\lambda, R \cdot Z, \pi_j^|, \text{len})$, we now call

$$(\{\text{ek}_{j,\beta}^|\}_{\beta \in [R \cdot Z]}, \{\text{rk}_{j,\beta}^|\}_{\beta \in [R \cdot Z]}, \text{tk}_j^|) \leftarrow \text{Sim}_j^|(1^\lambda, R \cdot Z, \text{len}, \mathcal{K}_{j,S}^|, \mathcal{K}_{j,R}^|, \text{Leak}(\pi_j^|, \mathcal{K}_{j,S}^|, \mathcal{K}_{j,R}^|))$$

2. During $\mathbf{Enc}(\mathbf{ek}_u, x_{u,t}^{(0)}, t)$ for $u \in \mathcal{H}_S$, instead of calling $\text{ict}_{u,t} \leftarrow \text{NIAR.Enc}(\mathbf{ek}_u^l, x_{u,t}, t)$ and $\widetilde{\text{ict}}_{u,t} \leftarrow \text{NIAR.Enc}(\mathbf{fek}_u^l, \mathbf{0}, t)$ for $u \in \mathcal{H}_S$, we now call

$$\{\text{ict}_{u,t}, \widetilde{\text{ict}}_{u,t}\}_{u \in \mathcal{H}_S} \leftarrow \text{Sim}_j^l \left(\left\{ \forall (j, \beta) \in \mathcal{K}_{j,R}^l \cap \pi_j^l(\mathcal{H}_{j,S}^l) : (\beta, y_{j,\beta,t}) \right\} \right)$$

where $y_{j,\beta,t}$ is the correct plaintext finally decrypted at the position β in the j -column in time step t , assuming that $\{x_{u,t}^{(0)}\}_{u \in \mathcal{H}_S}$ is the challenge vector being encrypted.

Claim 7.3.2. *Suppose that the NIAR scheme is SIM-secure. The adversary's views in Hyb_1^0 and NIDAR-Expt^0 are computationally indistinguishable.*

Proof. Follows in a straightforward fashion due to the SIM-security of the underlying NIAR scheme and the hybrid argument. Specifically, we can swap the column instances of NIAR one by one with an NIAR simulator. \square

7.3.3.1.3 Experiment Hyb_2^0 . Almost the same as Hyb_1^0 , except that that we now replace each row instance of NIAR with a NIAR simulator as well, as described below — here we use Sim_i^- to denote the (stateful) NIAR simulator corresponding to the i -th column instance:

1. During $\mathbf{Setup}(1^\lambda, n, \pi^{(0)}, \text{len})$, instead of calling $(\{\mathbf{ek}_{i,\alpha}^-\}_{\alpha \in [C \cdot Z]}, \{\mathbf{rk}_{i,\alpha}^-\}_{\alpha \in [C \cdot Z]}, \mathbf{tk}_i^-) \leftarrow \text{NIAR.Setup}(1^\lambda, R \cdot Z, \pi_i^-, \text{len}')$, we now call

$$\left(\{\mathbf{ek}_{i,\alpha}^-\}_{\alpha \in [C \cdot Z]}, \{\mathbf{rk}_{i,\alpha}^-\}_{\alpha \in [C \cdot Z]}, \mathbf{tk}_i^- \right) \leftarrow \text{Sim}_i^-(1^\lambda, C \cdot Z, \text{len}', \mathcal{K}_{i,S}^-, \mathcal{K}_{i,R}^-, \text{Leak}(\pi_i^-, \mathcal{K}_{i,S}^-, \mathcal{K}_{i,R}^-))$$

where $\mathcal{K}_{i,R}^- := \{(i, \alpha)\}_{\alpha \in [C \cdot Z]}$.

2. During $\mathbf{Enc}(\mathbf{ek}_u, x_{u,t}^{(0)}, t)$ for $u \in \mathcal{H}_S$, instead of calling $\text{ct}_{u,t} \leftarrow \text{NIAR.Enc}(\mathbf{ek}_u^-, \text{ict}_{u,t}, t)$ and $\widetilde{\text{ct}}_{u,t} \leftarrow \text{NIAR.Enc}(\mathbf{fek}_u^-, \widetilde{\text{ict}}_{u,t}, t)$, we now call

$$\{\text{ct}_{u,t}, \widetilde{\text{ct}}_{u,t}\}_{u \in \mathcal{H}_S} \leftarrow \text{Sim}_i^- \left(\left\{ \forall (i, \alpha) \in \pi_i^-(\mathcal{H}_{i,S}^-) : (\alpha, y_{i,\alpha,t}) \right\} \right)$$

where $y_{i,\alpha,t}$ is the simulated inner ciphertext to be routed to position α of row i during the row-wise permutation in time step t .

Claim 7.3.3. *Suppose that the NIAR scheme is SIM-secure. Then, the adversary's views in Hyb_1^0 and Hyb_2^0 are computationally indistinguishable.*

Proof. Follows in a straightforward fashion due to the SIM-security of the underlying NIAR scheme and the hybrid argument. Specifically, we can swap the row instances of NIAR one by one with an NIAR simulator. \square

7.3.3.1.4 Experiment Hyb_3^0 . Hyb_3^0 is a rewrite of Hyb_2^0 where we change how we sample the random coins. In Hyb_3^0 , we introduce an *initial sampling phase* where a subset of the random coins and events are sampled. Then, based on the outcomes of these partial random coins and events, we invoke a *simulator* that completes the rest of the experiment including interactions with the adversary.

Below we first describe the initial sampling phase.

1. Sample m_1, m_2, \dots, m_C , by throwing n balls into C bins, and counting the bin loads.
2. Sample the complement permutation π' at random, and compute $\pi_{\text{mid}} := (\pi')^{-1} \circ \pi^{(0)}$, which is the permutation to be realized by `MatrixPerm`. At this moment, the following random coins are fully determined:
 - which bucket each real element uR (either real or filler) should land in during the row-wise permutations, and if any bucket's load exceeds Z , return overflow just like before. More specifically, suppose that sender u is mapped to $\pi^{(0)}(u)$ as its final destination. Now, let k be the unique integer such that $\sum_{j=1}^k m_j < \pi_{\text{mid}}(u) \leq \sum_{j=1}^{k+1} m_j$, then the real element uR should go into the k -th bucket in its corresponding row; and
 - the destination of each real element uR during each of the column-wise permutations.
3. Sample the number of corrupt filler elements for all the buckets in all rows.

At this point, imagine we run the following simulator which continues to interact with the adversary:

Input:

1. set of corrupt senders $\mathcal{K}_S \subseteq [n]$ and set of corrupt receivers $\mathcal{K}_R \subseteq [n]$;
2. for every $u \in [n]$, if $(\pi^{(0)})^{-1}(u) \in \mathcal{H}_S$, what message the corrupt receiver u receives from some honest sender in each time step, based on the $\{x_{u,t}^{(0)}\}_{u,t}$ values.
3. the destination of every corrupt sender $u \in \mathcal{K}_S \subseteq [n]$ based on $\pi^{(0)}$;
4. m_1, m_2, \dots, m_C ;
5. π' ;
6. how many corrupt filler elements land in each bucket during the row-wise permutations;

7.3.3.1.5 Simulator algorithm. The simulator now performs the following:

- For each row $i \in [R]$, based on how many corrupt filler elements are to be received in each bucket during the i -th row-wise permutation, randomly assign the corrupt filler elements belonging to this row to the buckets;

Recall that which bucket each corrupt real element should go during the row-wise permutations was already determined during the initial sampling phase. Therefore, at this time, the simulator

knows which bucket each corrupt element (including real and filler) lands in during the row-wise permutations.

- For each bucket, the simulator picks a random unconsumed position for each corrupt element that is supposed to go into this bucket during the row-wise permutation. At this moment, it is fully determined where all corrupt elements go during the row-wise permutation.
- For each $j \in [C]$, for all the corrupt filler elements in column j after the row-wise permutation, pick a random (non-overlapping) position among the last $R \cdot Z - m_j$ positions to be its destination during the j -th column-wise permutation. At this moment, the routes of all corrupt elements during the row-wise and column-wise permutations are fully determined.
- At this moment, it is not hard to see that the simulator can accomplish the interactions with the adversary, since it knows all the inputs needed for calling the NIAR's simulators $\{\text{Sim}_j^|\}_{j \in [C]}$ and $\{\text{Sim}_i^-\}_{i \in [R]}$.

Claim 7.3.4. Hyb_3^0 and Hyb_2^0 are identically distributed.

Proof. It is not difficult to check that Hyb_3^0 is simply a rewrite of Hyb_2^0 , where the random coins are sampled in a different manner, by sampling a subset of the random coins and events first in an initial sampling stage, and then having a simulator accomplish the remaining. \square

7.3.3.1.6 Experiment Hyb_3^1 . Hyb_3^1 is almost the same as Hyb_3^0 , except the following modifications. Recall that the adversary submits $\pi^{(0)}$ and $\pi^{(1)}$ that are almost identical except for swapping the destinations of two honest senders. Moreover, recall that in Hyb_3^0 , we first have an initial sampling stage where part of the random coins are sampled; then we invoke a simulator with some input, and the rest of the simulation is completed by this simulator.

1. During the initial sampling stage: let $\pi_{\text{mid}} := (\pi')^{-1} \circ \pi^{(1)}$.
2. Part of the inputs to the simulator is changed to the following:
 - for every $u \in [n]$, if $(\pi^{(1)})^{-1}(u) \in \mathcal{H}_S$, what message the corrupt receiver u receives from some honest sender in each time step, based on the $\{x_{u,t}^{(1)}\}_{u,t}$ values;
 - the destination of every corrupt sender $u \in \mathcal{K}_S \subseteq [n]$ based on $\pi^{(1)}$.

Lemma 7.3.5. *Suppose that $(1 - \rho)Z \geq \Theta(\log \frac{1}{\delta})$ where $\Theta(\cdot)$ hides some appropriately large constant, where ρ is the fraction of corrupt senders. For any S ,*

$$\Pr[\text{view}_{\mathcal{A}}(\text{Hyb}_3^0) \in S] \leq e^\epsilon \cdot \Pr[\text{view}_{\mathcal{A}}(\text{Hyb}_3^1) \in S] + \delta'$$

where $\text{view}_{\mathcal{A}}(\text{Hyb}_3^b)$ denotes the adversary's view in experiment Hyb_3^b for $b \in \{0, 1\}$, and

$$\epsilon = O(1) \cdot \frac{\sqrt{Z \cdot \min((1 - \rho), \rho) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z}, \quad \delta' = O(n\delta)$$

Proof. Observe that due to the admissibility rule on the adversary, for the above inputs to the simulator, it does not matter whether we use $\{x_{u,t}^{(0)}\}_{u,t}, \pi^{(0)}$ or $\{x_{u,t}^{(1)}\}_{u,t}, \pi^{(1)}$ — the outcomes are the same. In this sense, the only real difference in Hyb_3^0 and Hyb_3^1 is that π_{mid} is now computed as $(\pi')^{-1} \circ \pi^{(1)}$.

In both Hyb_3^0 and Hyb_3^1 , the adversary's view depends only on the simulator's input (and the internal random coins tossed by the simulator). Let the simulator's input be Inp^b in Hyb_3^b for $b \in \{0, 1\}$. By the post-processing lemma of differential privacy [Vad17; DR14], it suffices to prove that for any S ,

$$\Pr[\text{Inp}^0 \in S] \leq e^\epsilon \cdot \Pr[\text{Inp}^1 \in S] + \delta' \quad (7.1)$$

Recall that the input to the simulator consists of the following:

1. set of corrupt senders $\mathcal{K}_S \subseteq [n]$ and set of corrupt receivers $\mathcal{K}_R \subseteq [n]$;
2. for every $u \in [n]$, if $(\pi^{(b)})^{-1}(u) \in \mathcal{H}_S$, what message the corrupt receiver u receives from some honest sender in each time step, based on the $\{x_{u,t}^{(b)}\}_{u,t}$ values.
3. the destination of every corrupt sender $u \in \mathcal{K}_S \subseteq [n]$ based on $\pi^{(b)}$;
4. m_1, m_2, \dots, m_C ;
5. π' ;
6. how many corrupt filler elements land in each bucket during the row-wise permutations;

For 1-5, they are the same no matter whether we are in Hyb_3^0 or Hyb_3^1 . Therefore, it suffices to prove that the numbers of corrupt filler elements that land in all buckets satisfy the above Equation (7.1) in the two experiments. This is the most technical step in our proof, we therefore present the full proof of this statement in Lemma 7.3.10 of Section 7.3.3.2. \square

7.3.3.1.7 Experiment Hyb_2^1 . Same as Hyb_2^0 except that $\pi^{(1)}$ and $\{x_{u,t}^{(1)}\}_{u,t}$ are used in place of $\pi^{(0)}$ and $\{x_{u,t}^{(0)}\}_{u,t}$.

Claim 7.3.6. Hyb_2^1 and Hyb_3^1 are identically distributed.

Proof. The proof is the same as Claim 7.3.4, i.e., Hyb_3^1 is a rewrite of Hyb_2^1 where the sampling is performed in a different way by sampling a subset of the random coins and events first, and then invoking a simulator which samples the remaining randomness and completes the interactions with the adversary. \square

7.3.3.1.8 Experiment Hyb_1^1 . Same as Hyb_1^0 except that $\pi^{(1)}$ and $\{x_{u,t}^{(1)}\}_{u,t}$ are used in place of $\pi^{(0)}$ and $\{x_{u,t}^{(0)}\}_{u,t}$.

Claim 7.3.7. Suppose that the NIAR scheme is SIM-secure. Then, the adversary's views in Hyb_1^1 and Hyb_2^1 are computationally indistinguishable.

Proof. The proof follows in the same way as that of Claim 7.3.3. \square

7.3.3.1.9 Experiment NIDAR-Expt¹. Same as the original NIDAR-Expt¹ experiment as defined in Section 7.2.

Claim 7.3.8. *Suppose that the NIAR scheme is SIM-secure. Then, the adversary's views in Hyb₁¹ and NIDAR-Expt¹ are computationally indistinguishable.*

Proof. The proof follows in the same way as that of Claim 7.3.2. □

Theorem 7.3.9 (2-layer NIDAR). *Let \mathcal{A} be an arbitrary non-uniform p.p.t. adversary that controls ρ fraction of the senders, and recall that for $b \in \{0, 1\}$, the experiment NIDAR-Expt^b outputs the adversary \mathcal{A} 's output. Suppose that $(1 - \rho)Z \geq \Theta(\log \frac{1}{\delta})$ where $\Theta(\cdot)$ hides a suitably large constant; further, suppose that the underlying NIAR scheme is SIM-secure. Then, there exists a negligible function $\text{negl}(\cdot)$, for any S ,*

$$\Pr[\text{NIDAR-Expt}^0 \in S] \leq e^\epsilon \cdot \Pr[\text{NIDAR-Expt}^1 \in S] + \delta'$$

where

$$\epsilon = O(1) \cdot \frac{\sqrt{Z \cdot \min((1 - \rho), \rho) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z}, \quad \delta' = O(n\delta) + \text{negl}(\lambda)$$

Proof. Due to the hybrid argument, the theorem follows from Claims 7.3.2, 7.3.3, 7.3.4, Lemma 7.3.5, as well as Claims 7.3.6, 7.3.7, and 7.3.8. □

7.3.3.2 Differential Privacy Lemma

Consider the following experiment DPExpt^b where $b \in \{0, 1\}$, in which the adversary's view captures the simulator's input in the earlier hybrid experiment Hyb₃^b.

DPExpt^b

1. The adversary submits two neighboring permutations $\pi^{(0)}$ and $\pi^{(1)}$, i.e., the two permutations are otherwise identical except for swapping the destinations of two honest senders.
2. The challenger randomly chooses a set of corrupt senders \mathcal{K}_S and tells the adversary the set \mathcal{K}_S .
3. The challenger samples m_1, \dots, m_C at random as mentioned before and tells the adversary these values.
4. The challenger samples a random π' . The challenger computes $\pi_{\text{mid}} = (\pi')^{-1} \circ \pi^{(b)}$. Tell the adversary π' .
5. Each row has C buckets, and each real element $u \in \mathbb{R}$ belongs to some row as mentioned earlier where $u \in [n]$. For each row $i \in [R]$, the challenger throws all real elements belonging to row i into C buckets. Suppose that $\pi_{\text{mid}}(u) = v$ where $\sum_{j=1}^k m_j < v \leq \sum_{j=1}^{k+1} m_j$, then the

element uR should go into the k -th bucket in its respective row. If any bucket exceeds Z real elements, abort throwing overflow.

6. For each row i , let $\mu_{i,1}, \dots, \mu_{i,C}$ denote the remaining empty slots inside the buckets belonging to row i . For each empty slot, fill it with a random filler element belonging to this row. Tell the adversary for each $i \in [R]$ and $j \in [C]$, exactly how many corrupt filler elements go into the j -th bucket of row i .
7. Return the adversary's view.

We want to prove the following lemma, which is the core technical lemma needed the proof of Lemma 7.3.5.

Lemma 7.3.10. *Assume that $(1 - \rho)Z \geq \Theta(\log \frac{1}{\delta})$ where ρ denotes the fraction of corrupt senders and $\Theta(\cdot)$ hides a sufficiently large constant. For any S ,*

$$\Pr[\text{DPExpt}^0 \in S] \leq e^\epsilon \cdot \Pr[\text{DPExpt}^1 \in S] + \delta'$$

where

$$\epsilon = O(1) \cdot \frac{\sqrt{Z \cdot \min((1 - \rho), \rho) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z}, \quad \delta' = O(n\delta)$$

Proof. The experiment DPExpt^b needs to flip various coins. We will use the following names to refer to these coins:

- *Corruption coins:* the coins used to determine the corrupt set of senders \mathcal{K}_S ;
- *Routing coins:* includes the random choice of m_1, \dots, m_C , and π' — these coins determine which bucket each real element will be thrown into;
- *Filler coins:* the coins used to decide which filler elements are used to fill the remaining empty slots in each bucket, after the real elements are thrown into buckets.

Let $\widetilde{M}_{i,j}^b$ denote the total number of filler elements that the j -th bucket of the i -th row wants to receive, assuming we are in DPExpt^b where $b \in \{0, 1\}$. Let u^* and v^* be the two honest senders whose destinations got swapped in $\pi^{(0)}$ and $\pi^{(1)}$. If u^* and v^* belong to the same row initially, then, fixing the same routing coins in DPExpt^0 or DPExpt^1 respectively, it must be that $\widetilde{M}_{i,j}^0 = \widetilde{M}_{i,j}^1$ for all i, j . In this case, the adversary's views are identical in DPExpt^0 and DPExpt^1 .

Below we focus on the case when u^* and v^* do not belong to the same row initially — specifically, suppose u^* belongs to row i_0 and v^* belongs to row i_1 , respectively. Once the routing coins are fixed in both DPExpt^0 and DPExpt^1 , the following must hold:

$\widetilde{M}_{i,j}^0 = \widetilde{M}_{i,j}^1$, for almost all $i \in [R]$ and $j \in [C]$, except for some i_0, j_0 , and i_1, j_1 , where

$$\begin{aligned}\widetilde{M}_{i_1, j_0}^0 &= \widetilde{M}_{i_1, j_0}^1 - 1, & \widetilde{M}_{i_1, j_1}^0 &= \widetilde{M}_{i_1, j_1}^1 + 1 \\ \widetilde{M}_{i_0, j_0}^0 &= \widetilde{M}_{i_0, j_0}^1 + 1, & \widetilde{M}_{i_0, j_1}^0 &= \widetilde{M}_{i_0, j_1}^1 - 1\end{aligned}$$

7.3.3.2.1 Focus on a single row r_0 . Henceforth we shall first focus on the row i_0 , since analyzing the other row i_1 is similar. Suppose we have fixed the routing coins. We use the vector $\{M_1, M_2, \dots, M_{j_b} + 1, \dots, M_C\}_{j \in [C]}$ to denote the total number of filler elements in each bucket of the i_0 -th row, when we are in DPExpt^b .

filler elements each bucket in row i_0 :

$$\text{DPExpt}^0 : (M_1, M_2, \dots, M_{j_0-1}, \underline{M_{j_0} + 1}, M_{j_0+1}, \dots, \dots, M_C)$$

$$\text{DPExpt}^1 : (M_1, M_2, \dots, \dots, M_{j_1-1}, \underline{M_{j_1} + 1}, M_{j_1+1}, \dots, M_C)$$

For $j \in [C]$, we use $\mu_j \leq M_j$ to denote the number of corrupt filler elements in the j -th bucket of the i_0 -th row. In the random process of DPExpt^b , we can imagine that first, the total number of filler elements of each bucket $\{M_1, M_2, \dots, M_{j_b} + 1, \dots, M_C\}_{j \in [C]}$ is determined, and then, the random variables $\{\mu_j\}_{j \in [C]}$ can be determined in the following way. Suppose that the i_0 -th row has ρ' fraction of corrupt senders. Suppose we have a database where exactly $M_1, M_2, \dots, M_{j_b} + 1, \dots, M_C$ elements have the attributes $1, 2, \dots, C$, respectively. We now sample $\rho' \cdot \frac{n}{R}$ elements at random without replacement from this database, and μ_j is the number of elements with attribute j . Note that $\frac{n}{R}$ denotes the total number of filler elements (including honest and corrupt) belonging to any specific row, and this is fixed regardless of whether we are in DPExpt^0 or DPExpt^1 .

Claim 7.3.11. *Suppose that $(1 - \rho)Z \geq \Theta'(\log \frac{1}{\delta})$ where $\Theta'(\cdot)$ hides a sufficiently large constant. No matter whether we are in DPExpt^0 or DPExpt^1 , the following statements hold. For any fixed bucket, with probability at least $1 - \delta$ over the choice of the routing coins, its load of real elements is between $[Z - O(\sqrt{Z} \cdot \log \frac{1}{\delta}), Z + O(\sqrt{Z} \cdot \log \frac{1}{\delta})]$. Further, for any fixed bucket, with probability at least $1 - \delta$ over the choice of routing coins, its load of filler elements is between $[Z - O(\sqrt{Z} \cdot \log \frac{1}{\delta}), Z + O(\sqrt{Z} \cdot \log \frac{1}{\delta})]$.*

As a direct corollary, for any fixed $j \in [C]$, with probability at least $1 - \delta$, $M_j = \Theta(Z)$ where $\Theta(\cdot)$ hides an appropriately large constant.

Proof. With the random process of DPExpt , essentially, every real element is assigned to a random bucket within its row. The expected number of real elements each bucket receives is exactly Z . Consider one fixed bucket. By the Chernoff bound, there are some appropriate constants c and c'

such that

$$\Pr \left[\text{a fixed bucket's real load} \in \left[Z - c \cdot \sqrt{Z \cdot \log \frac{1}{\delta}}, Z + c \cdot \sqrt{Z \cdot \log \frac{1}{\delta}} \right] \right] \\ \geq 1 - \exp \left(-c' \cdot \log \frac{1}{\delta} \right) = 1 - \delta$$

□

Claim 7.3.12. *Suppose that $(1 - \rho)Z \geq \Theta(\log \frac{1}{\delta})$ where $\Theta(\cdot)$ hides an appropriately large constant. Then, with probability $1 - \delta$ over the choice of the corruption coins, it must be that $1 - \rho' = \Theta(1 - \rho)$ and $\rho'Z = O(\rho Z + \log \frac{1}{\delta})$.*

Proof. Follows in a straightforward fashion from the Chernoff bound. □

Lemma 7.3.13. *Let ρ' be the fraction of corrupt senders in row i_0 . For any fixed $j \in [C]$, the following holds regardless of the choice of b , with probability at least $1 - \delta$ over the choice of filler coins (of row i_0):*

$$\mu_j \in \left[\rho' M_j - O \left(\sqrt{\min(\rho', 1 - \rho') M_j \log \frac{1}{\delta}} + \log \frac{1}{\delta} \right), \right. \\ \left. \rho' M_j + O \left(\sqrt{\min(\rho', 1 - \rho') M_j \log \frac{1}{\delta}} + \log \frac{1}{\delta} \right) \right]$$

As a corollary, suppose that $(1 - \rho)Z \geq \Theta(\log \frac{1}{\delta})$ where $\Theta(\cdot)$ hides a sufficiently large constant. For any fixed j , it must be that conditioned on any specific choice of good routing coins and corruption coins that satisfy the good events of Claims 7.3.11 and 7.3.12, with probability at least $1 - \delta$ over the choice of the filler coins (of row i_0),

- $M_j - \mu_j \geq \Theta((1 - \rho)Z)$;
- $\mu_j < M_j - 1$.

Proof. By negative association and the Chernoff bound, with at least $1 - \delta$ probability, the total number of honest filler elements in bucket j of row i_0 is within the range $[(1 - \rho')M_j - O(\sqrt{(1 - \rho')M_j \log \frac{1}{\delta}} + \log \frac{1}{\delta}), (1 - \rho')M_j + O(\sqrt{(1 - \rho')M_j \log \frac{1}{\delta}} + \log \frac{1}{\delta})]$. Observe that the total number of honest and corrupt filler elements of the j -th bucket of row i_0 is exactly M_j . It follows that with at least $1 - \delta$ probability, $\mu_j \in [\rho' M_j - O(\sqrt{(1 - \rho')M_j \log \frac{1}{\delta}} + \log \frac{1}{\delta}), \rho' M_j + O(\sqrt{(1 - \rho')M_j \log \frac{1}{\delta}} + \log \frac{1}{\delta})]$.

Similarly, by negative association and Chernoff bound, we get that with at least $1 - \delta$ probability, $\mu_j \in [\rho' M_j - O(\sqrt{\rho' M_j \log \frac{1}{\delta}} + \log \frac{1}{\delta}), \rho' M_j + O(\sqrt{\rho' M_j \log \frac{1}{\delta}} + \log \frac{1}{\delta})]$. The lemma follows by combining the above.

□

For convenience, we shall use the notation rc to denote the union of the router coins and the corruption coins. We use rc to denote the random variable and use rc to denote any specific choice of these coins. For any good choice rc that satisfies the good events of Claims 7.3.11 and 7.3.12, — note that these coins fix the M_1, \dots, M_C values. Let μ_1, \dots, μ_C be a set of good values that satisfy the good events of Lemma 7.3.13, w.r.t. these M_1, \dots, M_C values. We have the following where Pr_b is taken over the choice of filler coins of row i_0 in DPExt^b .

$$\begin{aligned}
\frac{\text{Pr}_0[\mu_1, \dots, \mu_C | \text{rc} = rc]}{\text{Pr}_1[\mu_1, \dots, \mu_C | \text{rc} = rc]} &\leq \frac{1 - \frac{\mu_{j_1}}{M_{j_1}+1}}{1 - \frac{\mu_{j_0}}{M_{j_0}+1}} = 1 + \frac{\frac{\mu_{j_0}}{M_{j_0}+1} - \frac{\mu_{j_1}}{M_{j_1}+1}}{1 - \frac{\mu_{j_0}}{M_{j_0}+1}} \\
&\leq 1 + \frac{O(1) \cdot \frac{\sqrt{Z \min(1-\rho', \rho') \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{Z}}{1 - \rho} \\
&\leq 1 + O(1) \cdot \frac{\sqrt{Z \min(1 - \rho', \rho') \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z} \\
&\leq 1 + O'(1) \cdot \frac{\sqrt{\min((1 - \rho)Z, \rho Z + \log \frac{1}{\delta}) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z} \\
&\leq 1 + O''(1) \cdot \frac{\sqrt{Z \cdot \min((1 - \rho), \rho) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z} \\
&\leq \exp \left(O'' \left(\frac{\sqrt{Z \cdot \min((1 - \rho), \rho) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z} \right) \right) \quad (\spadesuit)
\end{aligned}$$

In the above, the first step of the derivation is proven in Lemma 7.3.14 and the proof is deferred to later.

Henceforth let $\epsilon := O'' \left(\frac{\sqrt{Z \cdot \min((1 - \rho), \rho) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z} \right)$.

7.3.3.2.2 Back to considering both rows i_0 and i_1 . So far, we have focused only on the row i_0 . Below, we shall complete the proof of Lemma 7.3.10, and we shall now consider both rows i_0 and i_1 . Let $\boldsymbol{\mu}[i_0]$ and $\boldsymbol{\mu}[i_1]$ denote the corrupt filler load vector for buckets in rows i_0 and i_1 , respectively.

Now, consider an arbitrary set $S := \{(\boldsymbol{\mu}[i_0], \boldsymbol{\mu}[i_1])\}$ containing choices of $(\boldsymbol{\mu}[i_0], \boldsymbol{\mu}[i_1])$ values. We also use S to denote the event that the adversary sees corrupt filler load vectors of rows i_0 and i_1 that lie within S .

$$\begin{aligned}
\Pr_0[S] &= \sum_{\mu[i_0], \mu[i_1] \in S} \Pr_0[\mu[i_0], \mu[i_1]] = \sum_{\mu[i_0], \mu[i_1] \in S} \sum_{rc} \Pr_0[\mu[i_0], \mu[i_1], rc] \\
&= \sum_{\mu[i_0], \mu[i_1] \in S} \sum_{rc: \text{good}} \Pr_0[\mu[i_0], \mu[i_1], rc] + \sum_{\mu[i_0], \mu[i_1] \in S} \sum_{rc: \neg \text{good}} \Pr_0[\mu[i_0], \mu[i_1], rc] \\
&= \sum_{\mu[i_0], \mu[i_1] \in S} \sum_{rc: \text{good}} \Pr_0[\mu[i_0] | rc] \cdot \Pr_0[\mu[i_1] | rc] \cdot \Pr_0[rc] + O(C) \cdot \log \frac{1}{\delta} \\
&\leq \sum_{\mu[i_0], \mu[i_1] \in S} \sum_{rc: \text{good}} e^\epsilon \cdot \Pr_1[\mu[i_0] | rc] \cdot e^\epsilon \cdot \Pr_1[\mu[i_1] | rc] \cdot \Pr_1[rc] + O(C) \cdot \log \frac{1}{\delta} \\
&\leq \sum_{\mu[i_0], \mu[i_1] \in S} \sum_{rc} e^{2\epsilon} \cdot \Pr_1[\mu[i_0] | rc] \cdot \Pr_1[\mu[i_1] | rc] \cdot \Pr_1[rc] + O(C) \cdot \log \frac{1}{\delta} \\
&= e^{2\epsilon} \cdot \Pr_1[S] + O(C) \cdot \log \frac{1}{\delta}
\end{aligned}$$

In the above, the subscript “ rc : good” means that the M_1, M_2, \dots, M_C values resulting from the choice of rc satisfy the good events of Claims 7.3.11, 7.3.12, and Lemma 7.3.13, w.r.t. the choice of $\mu[i_0]$ and $\mu[i_1]$ which are already fixed in the outer summation. The notation $\Pr_b[\text{evt}]$ denotes the probability of seeing the event evt in DPExt^b , and if we write $\Pr[\text{evt}]$ without a subscript, it means that the probability of the relevant event evt is the same in both DPExt^0 and DPExt^1 . The $O(C)$ factor before the $\log \frac{1}{\delta}$ is due to taking a union bound over all choice of $j \in [C]$. \square

7.3.3.2.3 Analysis of the sampling mechanism. Imagine that we have a database containing M items, where each item is assigned some attribute from the domain $[C]$, i.e., there are C total attributes. Consider the following sampling mechanism **Samp**.

The sampling mechanism **Samp**

Sample s items at random (without replacement) from this database, and output a vector $(\mu_1, \mu_2, \dots, \mu_C)$ reporting the total number of occurrences for each of the C attributes.

We now prove a useful lemma that will be needed in our differential anonymity proof. Consider two neighboring databases DB and DB' . The only difference in DB and DB' is that the i -th item’s attribute is changed from k to k' . Suppose that in database DB , the total number of occurrences for each of the C attributes is denoted $(M_1, M_2, \dots, M_k + 1, \dots, M_C)$, and in DB' , the total number of occurrences for each of the C attributes is denoted $(M_1, M_2, \dots, M_{k'} + 1, \dots, M_C)$. We use the notations $\Pr_{\text{DB}}[\mu_1, \dots, \mu_C]$ and $\Pr_{\text{DB}'}[\mu_1, \dots, \mu_C]$ to denote the probabilities of encountering the sample (μ_1, \dots, μ_C) , when the database is DB and DB' , respectively.

Lemma 7.3.14. *Given any (μ_1, \dots, μ_C) vector where $\mu_j \leq M_j$ for $j \in [C]$,*

$$\frac{\Pr_{DB}[\mu_1, \dots, \mu_C]}{\Pr_{DB'}[\mu_1, \dots, \mu_C]} = \frac{1 - \frac{\mu_{k'}}{M_{k'}+1}}{1 - \frac{\mu_k}{M_k+1}}$$

Proof. We have

$$\Pr_{DB}[\mu_1, \dots, \mu_C] = \frac{\binom{M_k+1}{\mu_k} \cdot \prod_{j \in [C], j \neq k} \binom{M_j}{\mu_j}}{\binom{M}{s}}$$

$$\Pr_{DB'}[\mu_1, \dots, \mu_C] = \frac{\binom{M_{k'}+1}{\mu_{k'}} \cdot \prod_{j \in [C], j \neq k'} \binom{M_j}{\mu_j}}{\binom{M}{s}}$$

where $M := \sum_{j \in [C]} M_j$ and $s = \sum_{j \in [C]} \mu_j$. Therefore,

$$\frac{\Pr_{DB}[\mu_1, \dots, \mu_C]}{\Pr_{DB'}[\mu_1, \dots, \mu_C]} = \frac{\binom{M_k+1}{\mu_k} \cdot \binom{M_{k'}}{\mu_{k'}}}{\binom{M_k}{\mu_k} \cdot \binom{M_{k'}+1}{\mu_{k'}}} = \frac{M_k + 1}{M_k + 1 - \mu_k} \cdot \frac{M_{k'} + 1 - \mu_{k'}}{M_{k'} + 1} = \frac{1 - \frac{\mu_{k'}}{M_{k'}+1}}{1 - \frac{\mu_k}{M_k+1}}$$

□

7.4 Multi-Layer NIDAR

7.4.1 Multi-Layer NIDAR Construction

Inspired by the two-layer construction, we now suggest a multi-layer variant. To formally describe the scheme, we will use a recursive construction.

L -layer NIDAR where $L \geq 2$

7.4.1.0.1 Assume: (same as before) after the adversary chooses which users to corrupt and before the **Setup** algorithm is first invoked, all senders are randomly permuted, and we renumber the senders from 1 to n after this initial permutation. Throughout the following algorithms, we refer to senders by these randomly renumbered identities.

7.4.1.0.2 Parameters: let $L \geq 2$ be the total number of layers. Let Z be an even number that denotes the bucket size. For simplicity, we will first assume that $R := (2n/Z)^{1/L}$ is an integer, and R is also the fanout in the butterfly network when the recursions are expanded all the way (see Figure 7.1). We will deal with the indivisible case later in this section.

7.4.1.0.3 Main algorithms: we describe the main algorithms below, where each algorithm may in turn call a recursive subroutine, denoted **RecSetup**, **RecEnc**, and **RecRte**, respectively.

We will define these recursive subroutines subsequently in Section 7.4.2.

• **Setup**($1^\lambda, n, \pi, \text{len}$):

– let $(\pi_{\text{mid}}, \{\text{ek}_v\}_{v \in [2n]}, \{\text{rk}_{j,\beta}^\dagger\}_{j \in [C], \beta \in [m_j]}, \text{tk}')$ \leftarrow **RecSetup**($1^\lambda, 2n, L, 1$), where

$$\text{tk}' := \left(\{m_j\}_{j \in [C]}, \{\text{tk}_i^-\}_{i \in [R]}, \{\text{tk}_j^\dagger\}_{j \in [C]}, \{\text{rk}_{i,\alpha}^-\}_{i \in [R], \alpha \in [C \cdot Z]} \right);$$

note that since each sender encrypts a real element and a filler element, we may pretend that each sender acts as two virtual senders, and this is why we pass $2n$ to the recursive call **RecSetup**;

– for each $u \in [n]$, let $\text{ek}_u := \{\text{ek}_{2(u-1)+1}, \text{ek}_{2u}\}$;

– compute the complement permutation π' such that $\pi' \circ \pi_{\text{mid}} = \pi$;

– let $\text{rk}_1, \dots, \text{rk}_n := \pi' \left(\{\text{rk}_{j,\beta}^\dagger\}_{j \in [C], \beta \in [m_j]} \right)$ where $\{\text{rk}_{j,\beta}^\dagger\}_{j \in [C], \beta \in [m_j]}$ is flattened as a 1-dimensional array in the lexicographical ordering of (j, β) ;

– let the router's token $\text{tk} := \left(\pi', \{m_j\}_{j \in [C]}, \{\text{tk}_i^-\}_{i \in [R]}, \{\text{tk}_j^\dagger\}_{j \in [C]}, \{\text{rk}_{i,\alpha}^-\}_{i \in [R], \alpha \in [C \cdot Z]} \right)$;

– output the sender and receiver keys $\{\text{ek}_u, \text{rk}_u\}_{u \in [n]}$, as well as the router token tk .

• **Enc**($\text{ek}_u, x_{u,t}, t$):

– parse $\text{ek}_u = (\text{ek}, \text{ek}')$;

– call $\text{ct} \leftarrow \text{RecEnc}(\text{ek}, x, t, L)$ and $\text{ct}' \leftarrow \text{RecEnc}(\text{ek}', \mathbf{0}, t, L)$; and

– output $\text{CT} := (\text{ct}, \text{ct}')$.

• **Rte**($\text{tk}, \text{CT}_{1,t}, \dots, \text{CT}_{n,t}$):

– let tk' be the same as tk but without the π' term;

– for each $u \in [n]$, parse $\text{CT}_{u,t} := (\text{ct}_{2(u-1)+1}, \text{ct}_{2u})$;

– call $\text{CT}'_1, \dots, \text{CT}'_n \leftarrow \text{RecRte}(\text{tk}', \text{ct}_1, \dots, \text{ct}_{2n}, L, 1)$ and return $\pi'(\text{CT}'_1, \dots, \text{CT}'_n)$.

• **Dec**($\text{rk}_u, \text{CT}'_u$): output **NIAR.Dec**($\text{rk}_u, \text{CT}'_u$).

7.4.2 Recursive Subroutines

7.4.2.0.1 Subroutine RecSetup($1^\lambda, n, \text{len}, \text{nLayer}, \text{bFin}$): If $\text{nLayer} = 1$, then

1. view the symbolic input $1R, 1F, 2R, 2F, \dots, \frac{n}{2}R, \frac{n}{2}F$ as R buckets each of size Z such that each bucket has half real and half filler elements, and simulate a run of the **RowPerm** algorithm resulting in the permutation π — recall that the **RowPerm** algorithm may throw an overflow exception if any bucket receives more real elements than its capacity.

2. let $(\{\text{ek}_v, \text{rk}_v\}_{v \in [n]}, \text{tk}) \leftarrow \text{NIAR.Setup}(1^\lambda, n, \pi, \text{len})$, and return $(\{\text{ek}_v, \text{rk}_v\}_{v \in [n]}, \text{tk})$.

Else, continue with the following:

1. View the symbolic vector $1R, 1F, 2R, 2F, \dots, \frac{n}{2}R, \frac{n}{2}F$ as a matrix containing $R \times C$ buckets each of size Z , where R is a global parameter defined earlier, and $C = n/(R \cdot Z)$.
 - If $\text{bFin} = 1$, then for each column, simulate a run of the ColPerm algorithm which randomly permutes the column and moves real elements to the front; and let m_1, m_2, \dots, m_C be the number of real elements in each column after applying the row-wise permutations;
 - Else if $\text{bFin} = 0$, then for each column, simulate a run of the RowPerm algorithm which assigns each real element to a random bucket, and uses the remaining filler elements at random to pad all buckets to its maximum capacity; furthermore, a random permutation is applied to within each bucket.

Let $\pi_1^\downarrow, \dots, \pi_C^\downarrow$ denote the resulting column-wise permutations.

2. For each $j \in [C]$, call

$$\left(\{\mathbf{ek}_{j,\beta}^\downarrow\}_{\beta \in [R \cdot Z]}, \{\mathbf{rk}_{j,\beta}^\downarrow\}_{\beta \in [R \cdot Z]}, \mathbf{tk}_j^\downarrow \right) \leftarrow \text{NIAR.Setup}(1^\lambda, R \cdot Z, \pi_j^\downarrow, \text{len})$$

3. For each $i \in [R]$, recursively call

$$\left(\pi_i^-, \{\mathbf{ek}_{i,\alpha}^-\}_{\alpha \in [C \cdot Z]}, \{\mathbf{rk}_{i,\alpha}^-\}_{\alpha \in [C \cdot Z]}, \mathbf{tk}_i^- \right) \leftarrow \text{RecSetup}(1^\lambda, C \cdot Z, \kappa \cdot \text{len}, \text{nLayer} - 1, 0)$$

where $1/\kappa$ denotes the rate of NIAR.Enc (i.e., κ is the ciphertext size divided by the plaintext size).

4. Let π_{mid} be the effective permutation after applying the row-wise permutation π_i^- to each row $i \in [R]$, and applying the column-wise permutation π_j^\downarrow to each column $j \in [C]$. In particular, if $\text{bFin} = 1$ then π_{mid} denotes the permutation on only the real elements; else, π_{mid} denotes the permutation on all elements (including real and filler).
5. For $v \in [n]$, suppose that the element v corresponds to the initial position (i, α) , and is routed to position (j, β) after the row-wise permutations¹, then, let $\mathbf{ek}_v := (\mathbf{ek}_{i,\alpha}^-, \mathbf{ek}_{j,\beta}^\downarrow)$.
6. If $\text{bFin} = 1$, then let $\mathbf{tk} := \left(\{m_j\}_{j \in [C]}, \{\mathbf{tk}_i^-\}_{i \in [R]}, \{\mathbf{tk}_j^\downarrow\}_{j \in [C]}, \{\mathbf{rk}_{i,\alpha}^-\}_{i \in [R], \alpha \in [C \cdot Z]} \right)$ and return $\left(\pi_{\text{mid}}, \{\mathbf{ek}_v\}_{v \in [n]}, \{\mathbf{rk}_{j,\beta}^\downarrow\}_{j \in [C], \beta \in [m_j]}, \mathbf{tk} \right)$.
Else, let $\mathbf{tk} := \left(\{\mathbf{tk}_i^-\}_{i \in [R]}, \{\mathbf{tk}_j^\downarrow\}_{j \in [C]}, \{\mathbf{rk}_{i,\alpha}^-\}_{i \in [R], \alpha \in [C \cdot Z]} \right)$ and return $\left(\pi_{\text{mid}}, \{\mathbf{ek}_v\}_{v \in [n]}, \{\mathbf{rk}_{j,\beta}^\downarrow\}_{j \in [C], \beta \in [R \cdot Z]}, \mathbf{tk} \right)$.

¹As before, position (i, α) refers to the α -th position of the i -th row, and position (j, β) refers to the β -th position of the j -th column.

7.4.2.0.2 Subroutine RecEnc(ek, x, t, nLayer): If $n\text{Layer} = 1$, then let $\text{CT} := \text{NIAR.Enc}(\text{ek}, x, t)$ and return $\overline{\text{CT}}$. Else,

1. parse $\text{ek} := (\text{ek}^-, \text{ek}^l)$;
2. let $\text{ict} \leftarrow \text{NIAR.Enc}(\text{ek}^l, x, t)$ and let $\text{CT} \leftarrow \text{RecEnc}(\text{ek}^-, \text{ict}, t, n\text{Layer} - 1)$;
3. return CT .

7.4.2.0.3 Subroutine RecRte(tk, CT₁, ..., CT_n, nLayer, bFin):

- If $b\text{Fin} = 1$, then parse $\text{tk} := (\{m_j\}_{j \in [C]}, \{\text{tk}_i^-\}_{i \in [R]}, \{\text{tk}_j^l\}_{j \in [C]}, \{\text{rk}_{i,\alpha}^-\}_{i \in [R], \alpha \in [C \cdot Z]})$; else parse $\text{tk} := (\{\text{tk}_i^-\}_{i \in [R]}, \{\text{tk}_j^l\}_{j \in [C]}, \{\text{rk}_{i,\alpha}^-\}_{i \in [R], \alpha \in [C \cdot Z]})$.
- Let $R := (n/Z)^{1/n\text{Layer}}$, $C := n/(R \cdot Z)$, and view $\text{CT}_1, \dots, \text{CT}_n$ as an $(R \times C)$ matrix where entries are buckets of size Z . We shall use $\text{CT}[i :]$ to denote the i -th row of the CT matrix.
For each row $i \in [R]$, let $\overline{\text{ict}}_{i,1}, \dots, \overline{\text{ict}}_{i,C \cdot Z} \leftarrow \text{RecRte}(\text{tk}_i^-, \text{CT}[i :], n\text{Layer} - 1, 0)$, and for each $\alpha \in [C \cdot Z]$, let $\text{ict}_{i,\alpha} \leftarrow \text{NIAR.Dec}(\text{rk}_{i,\alpha}^-, \overline{\text{ict}}_{i,\alpha})$;
- View $\{\text{ict}_{i,\alpha}\}_{i \in [R], \alpha \in [C \cdot Z]}$ also as a $(R \times C)$ -matrix where each entry is a bucket of size Z — we shall use $\text{ict}[: j]$ to denote the j -th column of this matrix.
For each column $j \in [C]$, let $\text{CT}'_{j,1}, \dots, \text{CT}'_{j,R \cdot Z} \leftarrow \text{NIAR.Rte}(\text{tk}_j^l, \text{ict}[: j])$;
- If $b\text{Fin} = 1$, then view $\{\text{CT}'_{j,\beta}\}_{j \in [C], \beta \in [m_j]}$ as a 1-dimensional array, and return the result.
Else, then view $\{\text{CT}'_{j,\beta}\}_{j \in [C], \beta \in [R \cdot Z]}$ as a 1-dimensional array, and return the result.

7.4.2.0.4 More general parameters. So far, we have assumed that $(\frac{2n}{Z})^{1/L}$ is an integer. If not, we can let $R := \lceil \lceil \lceil \frac{2n}{Z} \rceil \rceil^{1/L}$, and this determines the structure of the routing network when the recursions are expanded all the way (see Figure 7.1). Moreover, in this indivisible case, each bucket will not all be of uniform capacity. It is not hard to ensure the invariant that every bucket's capacity is either Z or $Z + 2$, and moreover, all buckets' capacities are even. With the slightly modified bucket size, we need to modify the instance size for each NIAR instance accordingly. With this resulting algorithm, all of our analyses would still hold.

7.4.2.0.5 Correctness The correctness of the multi-layer NIDAR can be generalized from the two-layer design. Specifically, by extending the proof of Lemma 7.3.1, with probability $1 - \text{negl}(\lambda)$, RecSetup will succeed in simulating the permutation without overflow and outputting key tuples. Then the user will encrypt the real and filler messages in an onion way according to the path decided by the RecSetup algorithm. Similarly, for each layer of decryption in RecRte, the correctness of NIAR scheme guarantees each element will be routed to match RecSetup algorithm. In the final array of intermediate ciphertexts, the $\pi_{\text{mid}}(u)$ -th element is the ciphertext of the user u 's real message

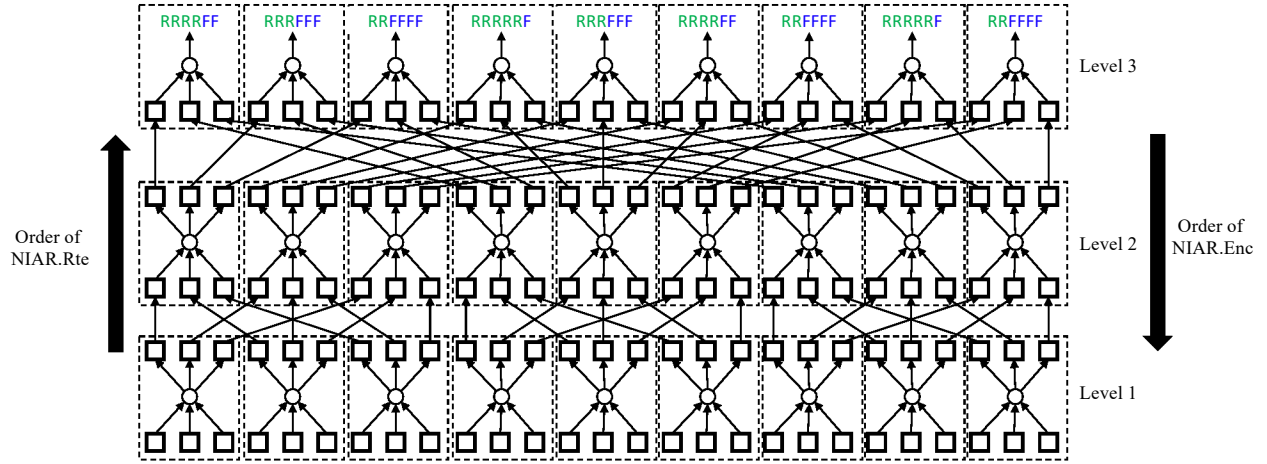


Figure 7.1: An R -way butterfly network when the recursion is fully expanded. In this example, $n_{\text{Layer}} = 3$, $2n/Z = 27$, and $R = (2n/Z)^{1/n_{\text{Layer}}} = 3$. The elements are being routed from level 1 to level L , and the onion layers of encryption are performed in the reverse order where level 1 is the outer-most layer. Each small box \square denotes a bucket, and each dashed big box is either a RowPerm or a ColPerm instance. The last level is special and employs ColPerm instances which route the real elements to the front in a random order.

and will be sent to the receiver $\pi' \circ \pi_{\text{mid}}(u)$. Because $\pi' \circ \pi_{\text{mid}} = \pi$, the $\pi(u)$ -th receiver will obtain the u -th sender's message.

7.4.2.0.6 Efficiency. For our efficiency analysis, suppose that the underlying NIAR is instantiated with the construction of Shi and Wu [SW21b]. We reviewed the asymptotic efficiency of the underlying NIAR scheme in Section 7.3.2. Recall that earlier, we used the notation $O_{\lambda}(\cdot)$ to hide $\text{poly}(\lambda)$ parameters — in fact, this notation hides a multiplicative factor related to the length of each bilinear group element in the underlying NIAR. If we use $\kappa = \text{poly}(\lambda)$ to denote the bit-length of a single bilinear group element in the underlying NIAR, then, $O_{\lambda}(\cdot)$ can be equivalently expressed as $O(\cdot) \cdot \kappa$. Note also that underlying NIAR's coding rate is $\Theta(\frac{1}{\kappa})$. i.e., the ratio of the ciphertext size and the plaintext size is $\Theta(\kappa)$.

Suppose the number of layers $L = O(1)$, and we now analyze the asymptotical performance bounds for our multi-layer NIDAR. Clearly, the receiver key is still $O(\kappa)$ in size like before. The sender key size is $O(\kappa \cdot R \cdot Z \cdot L) = O\left(\kappa \cdot \left(\frac{n}{Z}\right)^{1/L} \cdot Z\right)$. Each of the L layers incur a multiplicative κ factor blowup in the ciphertext size. Therefore, the per-sender ciphertext size as well as encryption runtime are $O(\kappa^L \cdot \text{len})$. The Rte cost is $O(\kappa^L \cdot \frac{n}{R \cdot Z} \cdot (R \cdot Z)^2) = O\left(\kappa^L \cdot \left(\frac{n}{Z}\right)^{1/L} \cdot n \cdot Z\right)$.

7.4.2.0.7 Expanding the recursion. Recall that part of the goal of the recursive RecSetup algorithm is to sample the permutation π_{mid} , and this permutation is realized over multiple layers of routing. For ease of understanding as well as in our proofs, it is often helpful to think about

what actually happens when we fully expand the recursion out. The network structure looks like Figure 7.1 when we fully expand the recursion out. In this example, we assume that $2n/Z = 27$, and $R = 3$ in each level of the recursion. Each little box \square represents a bucket. Each big dashed box represents a RowPerm or ColPerm instance.

The elements are routed from level 1 to level 3. Except for the last level which is a little special and uses ColPerm, for all other levels, each dashed box denotes a RowPerm instance. Inside each RowPerm instance, all real elements are thrown into random buckets, and if any bucket's load exceeds Z , simply throw an overflow exception. Next, for each remaining empty slot inside each bucket, we fill them with a random unconsumed filler element belonging to this instance. In the last level, each dashed box represents a ColPerm instance, which randomly permutes all elements moving all real elements to the front.

Our multi-layer NIDAR scheme is routing real elements as follows. In each RowPerm instance in levels 1 through $L - 1$, the real elements are throw at random into buckets, and an overflow exception is thrown if any bucket receives more real elements than its capacity Z . In the last level, all the real elements are routed to the front arranged in a random order in each ColPerm instance.

The following lemma is a counterpart of Lemma 7.3.1 for the multi-layer case.

Lemma 7.4.1. *The output of the the above random process outputs a permutation of the real elements, and moreover, the resulting permutation has statistical distance at most $O(nL) \cdot \exp(-\Omega(Z))$ from a uniform random permutation.*

Proof. The proof is essentially identical to that of Lemma 7.3.1. □

7.4.3 Proofs

Theorem 7.4.2 (L -layer NIDAR). *Let $L \geq 2 = O(1)$ be the number of layers, and let \mathcal{A} be an arbitrary non-uniform p.p.t. adversary that controls ρ fraction of the senders. Suppose that $(1 - \rho)Z \geq \Theta(\log \frac{1}{\delta})$ where $\Theta(\cdot)$ hides a suitably large constant; further, suppose that the underlying NIAR scheme is SIM-secure. Then, there exists a negligible function $\text{negl}(\cdot)$, for any S ,*

$$\Pr[\text{NIDAR-Expt}^0 \in S] \leq e^\epsilon \cdot \Pr[\text{NIDAR-Expt}^1 \in S] + \delta'$$

where

$$\epsilon = \frac{\sqrt{Z \cdot \min((1 - \rho), \rho) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z}, \quad \delta' = O(n \cdot \delta) + \text{negl}(\lambda)$$

The remainder of this section will be dedicated to the proof of Theorem 7.4.2. The proof of the multi-layer scheme is an extension of the two-layer proof. Recall that our multi-layer construction is recursive. If a NIAR instance is instantiated in a recursive call when the variable $n\text{Layer} = \ell$, we say that this is a *level- ℓ* NIAR instance. Suppose that $R := (n/Z)^{1/L}$ is an integer. If one fully expands the recursion out, then there are R number of level- L instances, R^2 number of level- $(L - 1)$ instances, and so on, and finally, there are R^L number of level-1 instances.

7.4.3.1 Sequence of Hybrids

We first define a sequence of hybrids.

7.4.3.1.1 Experiment NIDAR-Expt⁰. Same as the original NIDAR-Expt⁰ experiment as defined in Section 7.2. Henceforth, we may assume that during the experiment NIDAR-Expt⁰ that interacts with \mathcal{A} , the experiment samples all the randomness needed in all instances of RowPerm and ColPerm upfront for the entire recursion. In this way, it will be determined at the beginning of the experiment where all elements will be routed to in each instance of RowPerm or ColPerm when the recursion is fully expanded.

7.4.3.1.2 Experiment Hyb_L⁰. Almost the same as NIDAR-Expt⁰ except that each level- L NIAR instance is replaced now with with a NIAR simulator. Recall that the NIAR's simulated **Setup** algorithm needs to know the destinations of all corrupt sources, and the NIAR's simulated **Enc** algorithm needs to know what message each corrupt destination receives in each time step. As mentioned earlier, since we assume that the experiment chooses all random coins needed by all RowPerm and ColPerm instances upfront, therefore, it is possible to pass to the NIAR simulator which are the corrupt sources in each level- L NIAR instances, and which are their destinations.

Claim 7.4.3. *Suppose that the NIAR scheme is SIM-secure. The adversary's views in Hyb₁^L and NIDAR-Expt⁰ are computationally indistinguishable.*

Proof. Follows in a straightforward fashion due to the SIM-security of the underlying NIAR scheme and the hybrid argument. \square

7.4.3.1.3 Experiment Hyb_{L-k}⁰ for $k \in [L - 1]$. Hyb_{L-k}⁰ is almost identical to Hyb_{L-k+1}⁰, except that in all level- $(L - k)$ instances, we replace each NIAR instance with a NIAR simulator.

Claim 7.4.4. *Suppose that the NIAR scheme is SIM-secure. The adversary's views in Hyb_{L-k}⁰ and Hyb_{L-k-1}⁰ are computationally indistinguishable for $k \in \{0, 1, \dots, L - 2\}$.*

Proof. Follows in a straightforward fashion due to the SIM-security of the underlying NIAR scheme and the hybrid argument. \square

7.4.3.1.4 Experiment Hyb_{*}⁰. Hyb_{*}⁰ is a rewrite of Hyb₁⁰, where we change how we sample the random coins. In Hyb_{*}⁰, we introduce an *initial sampling phase* where a subset of the random coins and events are sampled. Then, based on the outcomes of these partial random coins and events, we invoke a *simulator* that completes the rest of the experiment including interactions with the adversary.

Henceforth, it is often helpful to think of the recursive algorithm that chooses the permutation π_{mid} as fully expanded out. Earlier in Section 7.4, we described what things look like when the recursion is fully expanded out, and how the permutation π_{mid} is chosen over multiple layers of routing.

Below we first describe the initial sampling phase.

1. Sample m_1, m_2, \dots, m_C , by throwing n balls into $2n/(R \cdot Z)$ bins, and counting the bin loads.
2. Sample the complement permutation π' at random, and compute $\pi_{\text{mid}} := (\pi')^{-1} \circ \pi^{(0)}$, which is the permutation chosen by RecSetup (specifically, the simulated version where all NIAR instances are replaced with NIAR simulators). At this moment, the following random coins are fully determined:
 - which bucket each real element uR (either real or filler) should land in during each RowPerm instance in the expanded recursion, and if any bucket's load exceeds Z , return overflow just like before;
 - and
 - the destination of each real element uR during each ColPerm instance.
3. Sample the number of corrupt filler elements for all the buckets in all RowPerm instances in the expanded recursion. To sample these random variables, we can go from level 1 to level $L - 1$ in the expanded recursion, and in each level, for each RowPerm instance, recall that the destinations of the real elements have already been fixed when we sampled π' . We can now sample the destinations for all the filler elements. After this, we calculate the number of corrupt filler elements that land in each bucket during each RowPerm instance, and throw away the rest of the information we have sampled since they will be resampled again freshly by the simulator, in the next stage.

At this point, imagine we run the following simulator which continues to interact with the adversary:

Input:

1. set of corrupt senders $\mathcal{K}_S \subseteq [n]$ and set of corrupt receivers $\mathcal{K}_R \subseteq [n]$;
2. for every $u \in [n]$, if $(\pi^{(0)})^{-1}(u) \in \mathcal{H}_S$, what message the corrupt receiver u receives from some honest sender in each time step, based on the $\{x_{u,t}^{(0)}\}_{u,t}$ values.
3. the destination of every corrupt sender $u \in \mathcal{K}_S \subseteq [n]$ based on $\pi^{(0)}$;
4. m_1, m_2, \dots, m_C ;
5. π' ;
6. how many corrupt filler elements land in each bucket during each RowPerm instance in the expanded recursion.

7.4.3.1.5 Simulator algorithm. The simulator now performs the following:

- Consider the expanded recursion. Starting from level-1 to $L - 1$, for every RowPerm instance, based on how many corrupt filler elements are to be received in each bucket during the

RowPerm instance, randomly assign the corrupt filler elements belonging this RowPerm instance to the buckets;

Recall that which bucket each corrupt real element should go during each RowPerm instance was already determined during the initial sampling phase. Therefore, at this time, the simulator knows which bucket each corrupt element (including real and filler) lands in during each RowPerm instance.

- For each bucket in each RowPerm instance, the simulator picks a random unconsumed position for each corrupt element that is supposed to go into this bucket during this RowPerm instance. At this moment, it is fully determined where all corrupt elements go during all RowPerm instances.
- For each $j \in [C]$, consider the ColPerm instance of column j in the last level of the recursion: for all the corrupt filler elements in column j belonging to this ColPerm instance, pick a random (non-overlapping) position among the last $R \cdot Z - m_j$ positions to be its destination. At this moment, the routes of all corrupt elements during all RowPerm and ColPerm instances are fully determined.
- At this moment, it is not hard to see that the simulator can accomplish the interactions with the adversary, since it knows all the inputs needed for calling the NIAR's simulators.

Claim 7.4.5. *The adversary's views in Hyb_1^0 and Hyb_*^0 are identically distributed.*

Proof. It is not difficult to check that Hyb_*^0 is simply a rewrite of Hyb_1^0 , where the random coins are sampled in a different manner, by sampling a subset of the random coins and events first in an initial sampling stage, and then having a simulator accomplish the remaining. \square

7.4.3.1.6 Experiment Hyb_*^1 . Hyb_*^1 is almost identical to Hyb_*^0 except the following changes.

1. During the initial sampling stage: let $\pi_{\text{mid}} := (\pi')^{-1} \circ \pi^{(1)}$.
2. Part of the inputs to the simulator is changed to the following:
 - for every $u \in [n]$, if $(\pi^{(1)})^{-1}(u) \in \mathcal{H}_S$, what message the corrupt receiver u receives from some honest sender in each time step, based on the $\{x_{u,t}^{(1)}\}_{u,t}$ values;
 - the destination of every corrupt sender $u \in \mathcal{K}_S \subseteq [n]$ based on $\pi^{(1)}$.

Due to the admissibility rule on the adversary, for the above inputs to the simulator, it does not matter whether we use $\{x_{u,t}^{(0)}\}_{u,t}, \pi^{(0)}$ or $\{x_{u,t}^{(1)}\}_{u,t}, \pi^{(1)}$ — the outcomes are the same. In this sense, the only real difference in Hyb_*^0 and Hyb_*^1 is that π_{mid} is now computed as $(\pi')^{-1} \circ \pi^{(1)}$.

Lemma 7.4.6. *Suppose that $(1 - \rho)Z \geq \Theta(\log \frac{1}{\delta})$ where $\Theta(\cdot)$ hides some appropriately large constant, where ρ is the fraction of corrupt senders. For any S ,*

$$\Pr[\text{view}_{\mathcal{A}}(\text{Hyb}_*^0) \in S] \leq e^\epsilon \cdot \Pr[\text{view}_{\mathcal{A}}(\text{Hyb}_*^1) \in S] + \delta'$$

where $\text{view}_{\mathcal{A}}(\text{Hyb}_3^b)$ denotes the adversary's view in experiment Hyb_*^b for $b \in \{0, 1\}$, and

$$\epsilon = O(1) \cdot \frac{\sqrt{Z \cdot \min((1 - \rho), \rho) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z}, \quad \delta' = O(n\delta)$$

Proof. The full proof of this lemma is deferred to Section 7.4.3.2. □

7.4.3.1.7 Experiment Hyb_{L-k}^1 for $k \in [L - 1] \cup \{0\}$. Same as Hyb_{L-k}^0 except that $\pi^{(1)}$ and $\{x_{u,t}^{(1)}\}_{u,t}$ are used in place of $\pi^{(0)}$ and $\{x_{u,t}^{(0)}\}_{u,t}$.

Claim 7.4.7. Suppose that the NIAR scheme is SIM-secure. The adversary's views in Hyb_{L-k}^1 and Hyb_{L-k-1}^1 are computationally indistinguishable for $k \in \{0, 1, \dots, L - 2\}$.

Proof. Symmetric to the proof of Claim 7.4.4. □

7.4.3.1.8 Experiment NIDAR-Expt^1 . Same as the original NIDAR-Expt^1 experiment as defined in Section 7.2.

Claim 7.4.8. Suppose that the NIAR scheme is SIM-secure. The adversary's views in Hyb_L^1 and NIDAR-Expt^1 are computationally indistinguishable.

Proof. Symmetric to the proof of Claim 7.4.3. □

7.4.3.1.9 Proof of Theorem 7.4.2. The proof of Theorem 7.4.2 due to the standard hybrid lemma and Claims 7.4.3, 7.4.4, 7.4.5, 7.4.7, and 7.4.8, and Lemma 7.3.5.

7.4.3.2 Differential Privacy Lemma

Let $R = (2n/Z)^{1/n_{\text{Layer}}}$, and consider an R -way butterfly network like the one depicted earlier in Figure 7.1. Now, consider the following experiment DPExpt^b where $b \in \{0, 1\}$, in which the adversary's view captures the simulator's input in the earlier hybrid experiment Hyb_*^b .

DPExpt^b

1. The adversary submits two neighboring permutations $\pi^{(0)}$ and $\pi^{(1)}$, i.e., the two permutations are otherwise identical except for swapping the destinations of two honest senders.
2. The challenger randomly chooses a set of corrupt senders \mathcal{K}_S and tells the adversary the set \mathcal{K}_S .
3. The challenger samples m_1, \dots, m_C at random as mentioned before and tells the adversary these values.

4. The challenger samples a complement random π' . The challenger computes $\pi_{\text{mid}} = (\pi')^{-1} \circ \pi^{(b)}$. Tell the adversary the complement permutation π' .
5. At this moment, it is fully determined that in the each RowPerm instance in R -way butterfly network, which bucket each real element will land. If any bucket exceeds Z real elements, abort throwing overflow.
6. For all RowPerm instances, simulate how many corrupt filler elements land in each bucket during this instance. Tell the adversary the number of corrupt filler elements that land in each bucket for all RowPerm instances.
7. Return the adversary's view.

Lemma 7.4.9. *Suppose that the total number of levels $L = O(1)$. Lemma 7.3.10 still holds for the new definition of DPExpt^0 and DPExpt^1 .*

Proof. Like in the proof of Lemma 7.3.10, we define three types of coins, corruption coins, routing coins, and filler coins. Their definitions are the same as before.

Once we fix the routing coins and corruption coins, the number filler elements in all buckets are fixed. Henceforth let $\widetilde{\mathbf{M}}_{\ell,r}^b$ denote the filler loads in all buckets of the r -th RowPerm instance in level ℓ , after this RowPerm instance finishes its routing in DPExpt^b . For each level $\ell \in [L-1]$, it must be that there are at most four r 's where $\widetilde{\mathbf{M}}_{\ell,r}^0 \neq \widetilde{\mathbf{M}}_{\ell,r}^1$. Henceforth, for each RowPerm instance indexed by (ℓ, r) such that $\widetilde{\mathbf{M}}_{\ell,r}^0 \neq \widetilde{\mathbf{M}}_{\ell,r}^1$, we call such an instance a *distinguishing* instance. For each distinguishing instance (ℓ, r) , there are the following possible scenarios:

1. there exists j_0 and j_1 such that the filler load vector becomes $\{M_1, M_2, \dots, M_{j_b} + 1, \dots, M_R\}_{j \in [R]}$ in this RowPerm instance in DPExpt^b for $b \in \{0, 1\}$;
2. there exists j_0 the filler load vector becomes $\{M_1, M_2, \dots, M_{j_0} + 1, \dots, M_R\}_{j \in [R]}$ in this RowPerm instance in DPExpt^0 , and becomes $\{M_1, M_2, \dots, M_{j_0}, \dots, M_R\}_{j \in [R]}$ in this RowPerm instance in DPExpt^1 ;
3. there exists j_1 the filler load vector becomes $\{M_1, M_2, \dots, M_{j_1} + 1, \dots, M_R\}_{j \in [R]}$ in this RowPerm instance in DPExpt^1 , and becomes $\{M_1, M_2, \dots, M_{j_1}, \dots, M_R\}_{j \in [R]}$ in this RowPerm instance in DPExpt^0 .

Henceforth, we may assume that for every (ℓ, r) , $\widetilde{\mathbf{M}}_{\ell,r}^b = \mathbf{M}_{\ell,r} + \Delta_{\ell,r}^b$ for some $\mathbf{M}_{\ell,r}$ and $\Delta_{\ell,r}^b$ such that

1. for every RowPerm instance (ℓ, r) that is not distinguishing, $\Delta_{\ell,r}^0 = \Delta_{\ell,r}^1 = \mathbf{0}$;
2. for a distinguishing RowPerm instance (ℓ, r) , it must be one of the following cases:
 - a) for $b \in \{0, 1\}$, there is a j_b such that $\Delta_{\ell,r,j_b}^b = 1$; all other coordinates in $\Delta_{\ell,r}^b$ are 0;
 - b) $\Delta_{\ell,r}^1 = \mathbf{0}$; moreover, there is a j_0 such that $\Delta_{\ell,r,j_0}^0 = 1$, and all other coordinates in $\Delta_{\ell,r}^0$ are 0;

c) $\Delta_{\ell,r}^0 = \mathbf{0}$; moreover, there is a j_1 such that $\Delta_{\ell,r,j_1}^1 = 1$, and all other coordinates in $\Delta_{\ell,r}^1$ are 0.

Below is the new counterpart of Claim 7.3.11. Henceforth we will often index a bucket by a tuple (ℓ, r, j) meaning that it is the j -th bucket in the RowPerm instance indexed by (ℓ, r) .

Claim 7.4.10. *Suppose $(1 - \rho)Z \geq \Theta'(\log \frac{1}{\delta})$ where $\Theta'(\cdot)$ hides a sufficiently large constant. It must be that for any fixed bucket indexed by (ℓ, r, j) , with probability at least $1 - \delta$, $M_{\ell,r,j} = \Theta(Z)$ where $\Theta(\cdot)$ hides an appropriately large constant.*

Proof. Due to a straightforward application of the Chernoff bound. \square

The following claim will be used as a counterpart of Claim 7.3.12 and Lemma 7.3.13 in the multi-layer case.

Claim 7.4.11. *Suppose that $(1 - \rho)Z \geq \Theta_1(\log \frac{1}{\delta})$ where $\Theta_1(\cdot)$ hides a sufficiently large constant and $L = O(1)$. For every fixed bucket in level 1, the fraction of filler elements that are honest among fillers is at least $\Theta_2(1 - \rho)$, except with $\frac{1}{\delta}$ probability over the choice of the corruption coins.*

Further, suppose that in some RowPerm instance denoted (ℓ, r) , the fraction of filler elements that are honest is $\Theta_3(1 - \rho)$, and suppose that every coordinate in $\mathbf{M}_{\ell,r}$ satisfies the good event of Claim 7.4.10. Then, for any fixed bucket within this RowPerm instance denoted (ℓ, r, j) , with $1 - \frac{1}{\delta}$ probability over the choice of the filler coins of this RowPerm instance,

- $M_{\ell,r,j} - \mu_{\ell,r,j} \geq \Theta_4((1 - \rho)Z)$ where $\mu_{\ell,r,j}$ is the number of corrupt filler elements that land in the bucket indexed by ℓ, r, j during the RowPerm instance indexed by (ℓ, r) ;
- among the filler elements that land in the bucket indexed by ℓ, r, j during the RowPerm instance indexed by (ℓ, r) , the fraction of honest elements is at least $\Theta_5(1 - \rho)$;
- $\mu_{\ell,r,j} < M_{\ell,r,j} - 1$.

Proof. The statement about the first level follows due to a straightforward application of the Chernoff bound. The rest of the claim can be proven in a similar fashion as that of Lemma 7.3.13 due to negative association and the Chernoff bound, and observing that since the constants are blown up over only $O(1)$ levels, they remain constants. \square

Henceforth, we use rc to denote some specific choice of the routing and corruption coins. We use $\boldsymbol{\mu}_{1:\ell}$ to denote some specific choice of the corrupt filler load vectors of all levels from 1 to ℓ .

Lemma 7.4.12. *Fix some good choice of rc that satisfies the good events of Claims 7.4.10 for all buckets. Further, consider an arbitrary $\ell \in [L - 1]$ and fix some good choice of $\boldsymbol{\mu}_{1:\ell}$ such that given rc and $\boldsymbol{\mu}_{1:\ell}$, the good events of Claim 7.4.11 hold for all buckets in levels 1 to ℓ . Now, consider some distinguishing RowPerm instance indexed by (ℓ, r) , taking probability over the filler coins of the instance (ℓ, r) , we have that*

$$\frac{\Pr_0[\boldsymbol{\mu}_{\ell,r} \mid rc, \boldsymbol{\mu}_{1:\ell-1}]}{\Pr_1[\boldsymbol{\mu}_{\ell,r} \mid rc, \boldsymbol{\mu}_{1:\ell-1}]} \leq e^\epsilon \quad \text{where } \epsilon := O'' \left(\frac{\sqrt{Z \cdot \min((1 - \rho), \rho) \log \frac{1}{\delta} + \log \frac{1}{\delta}}}{(1 - \rho)Z} \right)$$

Proof. Recall that there are three cases for a distinguishing instance.

For case (a), the lemma follows due to the same analysis as Lemma 7.3.14 and the proof of Lemma 7.3.10. For cases (b) and (c), we will instead use Lemma 7.4.13 (to be proven later) in place of Lemma 7.3.10. For case (b), we have

$$\frac{\Pr_0[\boldsymbol{\mu}_{\ell,r} \mid rc, \boldsymbol{\mu}_{1:\ell-1}]}{\Pr_1[\boldsymbol{\mu}_{\ell,r} \mid rc, \boldsymbol{\mu}_{1:\ell-1}]} \leq \frac{1 - \frac{\sum_j \mu_{\ell,r,j}}{\sum_j M_{\ell,r,j} + 1}}{1 - \frac{\mu_{\ell,r,j_0}}{M_{\ell,r,j_0} + 1}}$$

For case (c), we have

$$\frac{\Pr_0[\boldsymbol{\mu}_{\ell,r} \mid rc, \boldsymbol{\mu}_{1:\ell-1}]}{\Pr_1[\boldsymbol{\mu}_{\ell,r} \mid rc, \boldsymbol{\mu}_{1:\ell-1}]} \leq \frac{1 - \frac{\mu_{\ell,r,j_1}}{M_{\ell,r,j_1} + 1}}{1 - \frac{\sum_j \mu_{\ell,r,j}}{\sum_j M_{\ell,r,j} + 1}}$$

In both cases (b) and (c), plugging in Claims 7.4.10 and 7.4.11, it is not hard to see that the same calculation steps in the proof of Lemma 7.3.10 — specifically, Equation (♠) — still hold here. Thus we arrive at the statement claimed. \square

Now, consider an arbitrary set $S = \{\boldsymbol{\mu}\}$ of choices of $\boldsymbol{\mu}$'s, where $\boldsymbol{\mu}$ denotes the vector of corrupt filler loads of all buckets. Let rc be some choice of routing and corruption coins. We use $\mathbb{G}(rc, \boldsymbol{\mu})$ to denote the event that given rc the resulting $\mathbf{M}_{\ell,r}$'s and $\boldsymbol{\mu}$ satisfy the good events of Claims 7.4.10 and 7.4.11.

We have

$$\begin{aligned} \Pr_0[S] &= \sum_{\boldsymbol{\mu} \in S} \sum_{rc} \Pr_0[\boldsymbol{\mu}, rc] \\ &= \sum_{\boldsymbol{\mu} \in S} \sum_{rc: \mathbb{G}(rc, \boldsymbol{\mu})} \Pr_0[\boldsymbol{\mu}, rc] + \sum_{\boldsymbol{\mu} \in S} \sum_{rc: \neg \mathbb{G}(rc, \boldsymbol{\mu})} \Pr_0[\boldsymbol{\mu}, rc] \\ &= \sum_{\boldsymbol{\mu} \in S} \sum_{rc: \mathbb{G}(rc, \boldsymbol{\mu})} \Pr[rc] \cdot \Pr_0[\boldsymbol{\mu}_1 \mid rc] \cdot \Pr_0[\boldsymbol{\mu}_2 \mid rc, \boldsymbol{\mu}_{1:1}] \cdot \Pr_0[\boldsymbol{\mu}_3 \mid rc, \boldsymbol{\mu}_{1:2}] \cdots \Pr_0[\boldsymbol{\mu}_{L-1} \mid rc, \boldsymbol{\mu}_{1:L-2}] \\ &\quad + O(n) \cdot \log \frac{1}{\delta} \\ &\leq \sum_{\boldsymbol{\mu} \in S} \sum_{rc: \mathbb{G}(rc, \boldsymbol{\mu})} e^{2(L-1)\epsilon} \cdot \Pr[rc] \cdot \Pr_1[\boldsymbol{\mu}_1 \mid rc] \cdot \Pr_1[\boldsymbol{\mu}_2 \mid rc, \boldsymbol{\mu}_{1:1}] \cdots \Pr_1[\boldsymbol{\mu}_{L-1} \mid rc, \boldsymbol{\mu}_{1:L-2}] \\ &\quad + O(n) \cdot \log \frac{1}{\delta} \\ &\leq e^{4(L-1)\epsilon} \cdot \Pr_1[S] + O(n) \cdot \log \frac{1}{\delta} \end{aligned}$$

Note that in the last but second inequality, the $e^{4(L-1)\epsilon}$ term comes from the fact that there are $L - 1$ levels, and moreover, for each level, there are at most 4 distinguishing instances. Furthermore, the $O(n)$ factor in the $O(n) \cdot \log \frac{1}{\delta}$ term comes from taking a union bound over all buckets. \square

7.4.3.2.1 Analysis of the sampling mechanism (variant). We consider the same sampling mechanism as before, except that 1) the attributes are chosen from $[R]$ rather than $[C]$; and 2) the two neighboring database are now $DB := (M_1, M_2, \dots, M_k, \dots, M_R)$ and $DB' := (M_1, M_2, \dots, M_k + 1, \dots, M_R)$.

Lemma 7.4.13. *Given any (μ_1, \dots, μ_R) vector where $\mu_j \leq M_j$ for $j \in [R]$,*

$$\frac{\Pr_{DB}[\mu_1, \dots, \mu_R]}{\Pr_{DB'}[\mu_1, \dots, \mu_R]} = \frac{1 - \frac{\mu_k}{M_k + 1}}{1 - \frac{s}{M + 1}}$$

Proof. We have

$$\Pr_{DB}[\mu_1, \dots, \mu_R] = \frac{\binom{M_k}{\mu_k} \cdot \prod_{j \in [R], j \neq k} \binom{M_j}{\mu_j}}{\binom{M}{s}}$$

$$\Pr_{DB'}[\mu_1, \dots, \mu_R] = \frac{\binom{M_k + 1}{\mu_k} \cdot \prod_{j \in [R], j \neq k} \binom{M_j}{\mu_j}}{\binom{M + 1}{s}}$$

where $M := \sum_{j \in [R]} M_j$ and $s = \sum_{j \in [R]} \mu_j$.

Therefore,

$$\frac{\Pr_{DB}[\mu_1, \dots, \mu_R]}{\Pr_{DB'}[\mu_1, \dots, \mu_R]} = \frac{\binom{M_k}{\mu_k} \cdot \binom{M + 1}{s}}{\binom{M_k + 1}{\mu_k} \cdot \binom{M}{s}} = \frac{M + 1}{M + 1 - s} \cdot \frac{M_k + 1 - \mu_k}{M_k + 1} = \frac{1 - \frac{\mu_k}{M_k + 1}}{1 - \frac{s}{M + 1}}$$

□

Bibliography

- [Aad] *Libaad*. github.com/alinush/libaad-ccs2019/tree/master/experiments.
- [ABCSSS12] P. Arjunan, N. Batra, H. Choi, A. Singh, P. Singh, and M. B. Srivastava. “SensorAct: a privacy and security aware federated middleware for building management”. In: *Proceedings of the 4th ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. BuildSys ’12. 2012, pp. 80–87.
- [Abe99] M. Abe. “Mix-Networks on Permutation Networks”. In: *Proceedings of the 5th International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology*. ASIACRYPT ’99. 1999, pp. 258–273.
- [ABF07] M. J. Atallah, M. Blanton, and K. B. Frikken. “Incorporating temporal capabilities in existing key management schemes”. In: *Proceedings of the 12th European Symposium On Research In Computer Security*. ESORICS ’07. 2007, pp. 515–530.
- [ABLSZ19] B. Abdolmaleki, K. Baghery, H. Lipmaa, J. Siim, and M. Zajac. “UC-Secure CRS Generation for SNARKs”. In: *Proceedings of the 11th International Conference on Cryptology in Africa*. AFRICACRYPT ’19. 2019, pp. 99–117.
- [Adk] H. Adkins. *An update on attempted man-in-the-middle attacks*. security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html.
- [AEVL16] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman. “MedRec: Using Blockchain for Medical Data Access and Permission Management”. In: *Proceedings of the 2nd International Conference on Open and Big Data*. OBD ’16. 2016, pp. 25–30.
- [AFB05] M. J. Atallah, K. B. Frikken, and M. Blanton. “Dynamic and efficient key management for access hierarchies”. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS ’05. 2005, pp. 190–202.
- [AFC16] M. P. Andersen, G. Fierro, and D. E. Culler. “System Design for a Synergistic, Low Power Mote/BLE Embedded Platform”. In: *Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN ’16. 2016, 17:1–17:12.
- [AHIV17] S. Ames, C. Hazay, Y. Ishai, and M. Venkatasubramanian. “Ligero: Lightweight Sublinear Arguments Without a Trusted Setup”. In: *Proceedings of the 24th ACM Conference on Computer and Communications Security*. CCS ’17. 2017, pp. 2087–2104.
- [AI09] N. Attrapadung and H. Imai. “Conjunctive Broadcast and Attribute-Based Encryption”. In: *Proceedings of the 3rd Pairing-Based Cryptography*. ICPBC ’09. 2009, pp. 248–265.

- [AIVG20] K. D. Albab, R. Issa, M. Varia, and K. Graffi. *Batched Differentially Private Information Retrieval*. Cryptology ePrint Archive, Report 2020/1596. <https://ia.cr/2020/1596>. 2020.
- [AKC17] M. P. Andersen, H. Kim, and D. E. Culler. “Hamilton: a cost-effective, low power networked sensor for indoor environment monitoring”. In: *Proceedings of the 4th ACM International Conference on Systems for Energy-Efficient Built Environments*. BuildSys ’17. 2017, 36:1–36:2.
- [AKCCK17] M. P. Andersen, J. Kolb, K. Chen, D. E. Culler, and R. H. Katz. “Democratizing authority in the built environment”. In: *Proceedings of the 4th ACM International Conference on Systems for Energy-Efficient Built Environments*. BuildSys ’17. 2017, 23:1–23:10.
- [AKN] M. Abdalla, E. Kiltz, and G. Neven. *Generalized Key Delegation for Hierarchical Identity-Based Encryption*. Cryptology ePrint Archive, Report 2007/221.
- [AL99] C. Adams and S. Lloyd. *Understanding public-key infrastructure: concepts, standards, and deployment considerations*. 1999.
- [Ale] Aleo Inc. *Aleo: Where Applications Become Private*. <https://www.aleo.org/>.
- [ALMSS98] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. “Proof verification and the hardness of approximation problems”. In: *Journal of the ACM* 45.3 (1998). Preliminary version in FOCS ’92., pp. 501–555.
- [AM18] M. Al-Bassam and S. Meiklejohn. “Contour: A Practical System for Binary Transparency”. In: *Proceedings of the 23rd Data Privacy Management, Cryptocurrencies and Blockchain Technology International Workshops*. ESORICS workshops ’18. 2018, pp. 94–110.
- [And+19] M. P. Andersen et al. “WAVE: A Decentralized Authorization Framework with Transitive Delegation”. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security’19. 2019, pp. 1375–1392.
- [ANSF16] M. Ali, J. Nelson, R. Shea, and M. J. Freedman. “Blockstack: A global naming and storage system secured by blockchains”. In: *USENIX ATC*. 2016.
- [APY20] I. Abraham, B. Pinkas, and A. Yanai. “Blinder: MPC Based Scalable and Robust Anonymous Committed Broadcast”. In: *ACM CCS*. 2020.
- [ark] arkworks. *arkworks: an ecosystem for developing and programming with zkSNARKs*. URL: <https://github.com/arkworks-rs>.
- [AS98] S. Arora and S. Safra. “Probabilistic checking of proofs: a new characterization of NP”. In: *Journal of the ACM* 45.1 (1998). Preliminary version in FOCS ’92., pp. 70–122.
- [AT83] S. G. Akl and P. D. Taylor. “Cryptographic solution to a problem of access control in a hierarchy”. In: *TOCS* (1983).
- [Bab85] L. Babai. “Trading group theory for randomness”. In: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*. STOC ’85. 1985, pp. 421–429.
- [Bac02] A. Back. “Hashcash - a denial of service counter-measure”. In: (2002).
- [BB04] D. Boneh and X. Boyen. “Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles”. In: *EUROCRYPT*. 2004.

- [BBBPWM18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. “Bulletproofs: Short Proofs for Confidential Transactions and More”. In: *Proceedings of the 39th IEEE Symposium on Security and Privacy*. S&P '18. 2018, pp. 315–334.
- [BBDP01] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. “Key-privacy in public-key encryption”. In: *ASIACRYPT*. 2001.
- [BBF19] D. Boneh, B. Bünz, and B. Fisch. “Batching techniques for accumulators with applications to iops and stateless blockchains”. In: *Crypto*. 2019.
- [BBG05] D. Boneh, X. Boyen, and E.-J. Goh. “Hierarchical identity based encryption with constant size ciphertext”. In: *EUROCRYPT and Cryptology ePrint Archive*. 2005.
- [BBHR18] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. “Fast Reed–Solomon Interactive Oracle Proofs of Proximity”. In: *Proceedings of the 45th International Colloquium on Automata, Languages and Programming*. ICALP '18. 2018, 14:1–14:17.
- [BBHR19] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. “Scalable Zero Knowledge with No Trusted Setup”. In: *Proceedings of the 39th Annual International Cryptology Conference*. CRYPTO '19. 2019, pp. 733–764.
- [BBS98] M. Blaze, G. Bleumer, and M. Strauss. “Divertible protocols and atomic proxy cryptography”. In: *EUROCRYPT* (1998).
- [BBSBS09] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. “Persona: an online social network with user-defined privacy”. In: *CCR*. 2009.
- [BC12] N. Bitansky and A. Chiesa. “Succinct Arguments from Multi-Prover Interactive Proofs and their Efficiency Benefits”. In: *Proceedings of the 32nd Annual International Cryptology Conference*. CRYPTO '12. 2012, pp. 255–272.
- [BCAR18] S. Belguith, S. Cui, M. R. Asghar, and G. Russello. “Secure Publish and Subscribe Systems with Efficient Revocation”. In: *SAC*. 2018.
- [BCC88] G. Brassard, D. Chaum, and C. Crépeau. “Minimum disclosure proofs of knowledge”. In: *Journal of Computer and System Sciences* 37.2 (1988), pp. 156–189.
- [BCCGP16] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. “Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting”. In: *Proceedings of the 35th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT '16. 2016, pp. 327–357.
- [BCCT13] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. “Recursive Composition and Bootstrapping for SNARKs and Proof-Carrying Data”. In: *Proceedings of the 45th ACM Symposium on the Theory of Computing*. STOC '13. 2013, pp. 111–120.
- [BCG20] J. Bootle, A. Chiesa, and J. Groth. “Linear-Time Arguments with Sublinear Verification from Tensor Codes”. In: *Proceedings of the 18th Theory of Cryptography Conference*. TCC '20. 2020, pp. 19–46.
- [BCGGHJ17] J. Bootle, A. Cerulli, E. Ghadafi, J. Groth, M. Hajiabadi, and S. K. Jakobsen. “Linear-Time Zero-Knowledge Proofs for Arithmetic Circuit Satisfiability”. In: *Proceedings of the 23rd International Conference on the Theory and Applications of Cryptology and Information Security*. ASIACRYPT '17. 2017, pp. 336–365.

- [BCGGRS19] E. Ben-Sasson, A. Chiesa, L. Goldberg, T. Gur, M. Riabzev, and N. Spooner. “Linear-Size Constant-Query IOPs for Delegating Computation”. In: *Proceedings of the 17th Theory of Cryptography Conference*. TCC ’19. 2019.
- [BCGJM18] J. Bootle, A. Cerulli, J. Groth, S. K. Jakobsen, and M. Maller. “Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution”. In: *Proceedings of the 24th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT ’18. 2018, pp. 595–626.
- [BCGRS17] E. Ben-Sasson, A. Chiesa, A. Gabizon, M. Riabzev, and N. Spooner. “Interactive Oracle Proofs with Constant Rate and Query Complexity”. In: *Proceedings of the 44th International Colloquium on Automata, Languages and Programming*. ICALP ’17. 2017, 40:1–40:15.
- [BCGTV15] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. “Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs”. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy*. S&P ’15. 2015, pp. 287–304.
- [BCGV16] E. Ben-Sasson, A. Chiesa, A. Gabizon, and M. Virza. “Quasilinear-Size Zero Knowledge from Linear-Algebraic PCPs”. In: *Proceedings of the 13th Theory of Cryptography Conference*. TCC ’16-A. 2016, pp. 33–64.
- [BCHO22] J. Bootle, A. Chiesa, Y. Hu, and M. Orrù. “Gemini: Elastic SNARKs for Diverse Environments”. In: *Proceedings of the 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT’22. 2022.
- [BCIOP13] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. “Succinct Non-Interactive Arguments via Linear Interactive Proofs”. In: *Proceedings of the 10th Theory of Cryptography Conference*. TCC ’13. 2013, pp. 315–333.
- [BCKPSS14] D. Basin, C. Cremers, T. Kim, A. Perrig, R. Sasse, and P. Szalachowski. “ARPKI: Attack resilient public-key infrastructure”. In: *CCS*. 2014.
- [BCL22] J. Bootle, A. Chiesa, and S. Liu. “Zero-Knowledge Succinct Arguments with a Linear-Time Prover”. In: *Proceedings of the 42nd Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT ’22. 2022.
- [BCPR16] N. Bitansky, R. Canetti, O. Paneth, and A. Rosen. “On the Existence of Extractable One-Way Functions”. In: *SIAM Journal on Computing* 45.5 (2016). Preliminary version appeared in STOC ’14., pp. 1910–1952.
- [BCQAS13] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. “DepSky: dependable and secure storage in a cloud-of-clouds”. In: *TOS* (2013).
- [BCRSVW19] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. “Aurora: Transparent Succinct Arguments for R1CS”. In: *Proceedings of the 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’19. Full version available at <https://eprint.iacr.org/2018/828>. 2019, pp. 103–128.
- [BCS16] E. Ben-Sasson, A. Chiesa, and N. Spooner. “Interactive Oracle Proofs”. In: *Proceedings of the 14th Theory of Cryptography Conference*. TCC ’16-B. 2016, pp. 31–60.

- [BCS21] J. Bootle, A. Chiesa, and K. Sotiraki. “Sumcheck Arguments and Their Applications”. In: *Proceedings of the 41st Annual International Cryptology Conference*. CRYPTO ’15. 2021, pp. 742–773.
- [BCTV14a] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Scalable Zero Knowledge via Cycles of Elliptic Curves”. In: *Proceedings of the 34th Annual International Cryptology Conference*. CRYPTO ’14. Extended version at <http://eprint.iacr.org/2014/595>. 2014, pp. 276–294.
- [BCTV14b] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture”. In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security ’14. Extended version at <http://eprint.iacr.org/2013/879>. 2014, pp. 781–796.
- [BD] R. Barbulescu and S. Duquesne. *Updating key size estimations for pairings*. Cryptology ePrint Archive, Report 2017/334.
- [BDFG20] D. Boneh, J. Drake, B. Fisch, and A. Gabizon. *Efficient polynomial commitment schemes for multiple points and polynomials*. Cryptology ePrint Archive, Report 2020/081. 2020.
- [BDLSY12] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang. “High-speed high-security signatures”. In: *JCEN* (2012).
- [BEGKN91] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. “Checking the correctness of memories”. In: *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*. FOCS ’91. 1991, pp. 90–99.
- [Ben+14] E. Ben-Sasson et al. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *Proceedings of the 35th IEEE Symposium on Security and Privacy*. SP ’14. 2014, pp. 459–474.
- [Ben14] J. Benet. “IPFS: Content Addressed, Versioned, P2P File System”. In: *CoRR* (2014). URL: <http://arxiv.org/abs/1407.3561>.
- [Ben+17] E. Ben-Sasson et al. “Computational integrity with a public random string from quasi-linear PCPs”. In: *Proceedings of the 36th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT ’17. 2017, pp. 551–579.
- [BF01] D. Boneh and M. Franklin. “Identity-based encryption from the Weil pairing”. In: *CRYPTO*. 2001.
- [BFLS91] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. “Checking computations in polylogarithmic time”. In: *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*. STOC ’91. 1991, pp. 21–32.
- [BFS20] B. Bünz, B. Fisch, and A. Szepieniec. “Transparent SNARKs from DARK Compilers”. In: *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’20. 2020, pp. 677–706.
- [BG12] S. Bayer and J. Groth. “Efficient Zero-Knowledge Argument for Correctness of a Shuffle”. In: *Proceedings of the 31st Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’12. 2012, pp. 263–280.

- [BGG17] S. Bowe, A. Gabizon, and M. Green. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*. Cryptology ePrint Archive, Report 2017/602. 2017.
- [BGK08] A. Boldyreva, V. Goyal, and V. Kumar. “Identity-based Encryption with Efficient Revocation”. In: *CCS*. 2008.
- [BGM17] S. Bowe, A. Gabizon, and I. Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. Cryptology ePrint Archive, Report 2017/1050. 2017.
- [BGPRR17] C. Borcea, A. B. D. Gupta, Y. Polyakov, K. Rohloff, and G. Ryan. “PICADOR: End-to-end encrypted Publish-Subscribe information distribution with proxy re-encryption”. In: *FGCS (2017)*.
- [BGW05] D. Boneh, C. Gentry, and B. Waters. “Collusion resistant broadcast encryption with short ciphertexts and private keys”. In: *CRYPTO*. 2005.
- [BHKP16] M. Backes, A. Herzberg, A. Kate, and I. Pryvalov. “Anonymous RAM”. In: *ESORICS*. 2016.
- [BHMS21] B. Bünz, Y. Hu, S. Matsuo, and E. Shi. “Non-Interactive Differentially Anonymous Router”. In: *IACR Cryptol. ePrint (2021)*, p. 1242.
- [BHRRS20] A. R. Block, J. Holmgren, A. Rosen, R. D. Rothblum, and P. Soni. “Public-Coin Zero-Knowledge Arguments with (almost) Minimal Time and Space Overheads”. In: *Proceedings of the 18th Theory of Cryptography Conference*. TCC ’20. 2020, pp. 168–197.
- [BHRRS21] A. R. Block, J. Holmgren, A. Rosen, R. D. Rothblum, and P. Soni. “Time- and Space-Efficient Arguments from Groups of Unknown Order”. In: *Proceedings of the 41st Annual International Cryptology Conference*. CRYPTO ’21. 2021, pp. 123–152.
- [BISW17] D. Boneh, Y. Ishai, A. Sahai, and D. J. Wu. “Lattice-Based SNARGs and Their Application to More Efficient Obfuscation”. In: *Proceedings of the 36th Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’17. 2017, pp. 247–277.
- [BISW18] D. Boneh, Y. Ishai, A. Sahai, and D. J. Wu. “Quasi-Optimal SNARGs via Linear Multi-Prover Interactive Proofs”. In: *Proceedings of the 37th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT ’18. 2018, pp. 222–255.
- [Bita] *Bitcoin*. <https://bitcoin.org/>.
- [Bitb] *BitInfoCharts*. <https://bitinfocharts.com/zcash/>.
- [Bla93] M. Blaze. “A cryptographic file system for UNIX”. In: *CCS*. 1993.
- [BLNN15] F. Buccafurri, G. Lax, S. Nicolazzo, and A. Nocera. “Accountability-preserving anonymous delivery of cloud services”. In: *TrustBus*. 2015.
- [BMPR14] D. Brunelli, I. Minakov, R. Passerone, and M. Rossi. “POVOMON: An Ad-hoc Wireless Sensor Network for indoor environmental monitoring”. In: *EESMS*. 2014.

- [BMRS21] C. Baum, A. J. Malozemoff, M. B. Rosen, and P. Scholl. “Mac’n’Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions”. In: *Proceedings of the 41st Annual International Cryptology Conference*. CRYPTO ’21. 2021, pp. 92–122.
- [Bon16] J. Bonneau. “EthIKS: Using Ethereum to audit a CONIKS key transparency log”. In: *FC*. 2016.
- [Bow18] S. Bowe. *BLS12-381: New zk-SNARK Elliptic Curve Construction*. <https://z.cash/blog/new-snark-curve/>. 2018.
- [BP15] E. Boyle and R. Pass. “Limits of Extractability Assumptions with Distributional Auxiliary Input”. In: *Proceedings of the 21st International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT ’15. 2015, pp. 236–261.
- [BPETVL14] A. Birgisson, J. G. Politz, Ú. Erlingsson, A. Taly, M. Vrable, and M. Lenczner. “Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud”. In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium*. NDSS. 2014.
- [BPH15] A. Baumann, M. Peinado, and G. Hunt. “Shielding applications from an untrusted cloud with haven”. In: *TOCS* (2015).
- [Bra+12] A. Brandt et al. *RPL: IPv6 routing protocol for low-power and lossy networks*. RFC. RFC Editor, 2012.
- [BS08] E. Ben-Sasson and M. Sudan. “Short PCPs with Polylog Query Complexity”. In: *SIAM Journal on Computing* 38.2 (2008). Preliminary version appeared in STOC ’05., pp. 551–607.
- [BS+14] E. Ben-Sasson et al. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *S&P*. 2014.
- [BSCTV17] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Scalable zero knowledge via cycles of elliptic curves”. In: *Algorithmica* (2017).
- [BSW07] J. Bethencourt, A. Sahai, and B. Waters. “Ciphertext-Policy Attribute-Based Encryption”. In: *S&P*. 2007.
- [But+13] V. Buterin et al. “Ethereum white paper”. In: *GitHub repository* (2013).
- [BW06] D. Boneh and B. Waters. “A fully collusion resistant broadcast, trace, and revoke system”. In: *CCS*. 2006.
- [Bw2] *bw2*. <https://github.com/immesys/bw2>.
- [BWZ14] D. Boneh, B. Waters, and M. Zhandry. “Low overhead broadcast encryption from multilinear maps”. In: *CRYPTO*. 2014.
- [BZ17] D. Boneh and M. Zhandry. “Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation”. In: *Algorithmica* (2017).
- [Cam17] B. Campbell. *Introducing Hail*. <https://www.tockos.org/blog/2017/introducing-hail/>. 2017.
- [CBM15] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. “Riposte: An Anonymous Messaging System Handling Millions of Users”. In: *S & P*. 2015.

- [CD16] V. Costan and S. Devadas. “Intel sgx explained.” In: *IACR Cryptol. ePrint* 2016 (2016).
- [CD98] R. Cramer and I. Damgård. “Zero-Knowledge Proofs for Finite Field Arithmetic; or: Can Zero-Knowledge be for Free?” In: *Proceedings of the 18th Annual International Cryptology Conference*. CRYPTO ’98. 1998, pp. 424–441.
- [CDDGS03] D. Clarke, S. Devadas, M. v. Dijk, B. Gassend, and G. E. Suh. “Incremental multiset hash functions and their application to memory integrity checking”. In: *International conference on the theory and application of cryptology and information security*. Springer. 2003, pp. 188–207.
- [CDGM19] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai. “Seemless: Secure end-to-end encrypted messaging with less trust”. In: *CCS*. 2019.
- [CEEFMR01] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. “Certificate chain discovery in SPKI/SDSI”. In: *Journal of Computer Security* (2001).
- [CFGJP15] J. Crampton, N. Farley, G. Gutin, M. Jones, and B. Poettering. “Cryptographic enforcement of information flow policies without public information”. In: *ACNS*. 2015.
- [CFN90] D. Chaum, A. Fiat, and M. Naor. “Untraceable Electronic Cash”. In: *CRYPTO*. 1990.
- [CFS17] A. Chiesa, M. A. Forbes, and N. Spooner. *A Zero Knowledge Sumcheck and its Applications*. Cryptology ePrint Archive, Report 2017/305. 2017.
- [CGBM15] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. “Riposte: An Anonymous Messaging System Handling Millions of Users”. In: *S&P*. 2015.
- [CGF10] H. Corrigan-Gibbs and B. Ford. “Dissent: Accountable Anonymous Group Messaging”. In: *CCS*. 2010, 340–350.
- [Cha81a] D. Chaum. “Untraceable electronic mail, return addresses, and digital pseudonyms”. In: *CACM* (1981).
- [Cha81b] D. L. Chaum. “Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms”. In: *Communications of the ACM* 24.2 (Feb. 1981), 84–90.
- [Cha82] D. Chaum. “Blind Signatures for Untraceable Payments”. In: *CRYPTO*. Ed. by D. Chaum, R. L. Rivest, and A. T. Sherman. 1982.
- [Cha83] D. Chaum. “Blind signatures for untraceable payments”. In: *EUROCRYPT*. 1983.
- [Cha84] D. Chaum. “Blind signature system”. In: *EUROCRYPT*. 1984.
- [Cha88] D. L. Chaum. “The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability”. In: *Journal of Cryptology* 1.1 (Mar. 1988), 65–75. ISSN: 0933-2790.
- [Che06] J. H. Cheon. “Security analysis of the strong Diffie-Hellman problem”. In: *EUROCRYPT*. 2006.
- [Che+19] R. Cheng et al. “Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts”. In: *EuroS&P*. 2019.
- [CHMMVW] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. P. Ward. *A Rust library for the Marlin preprocessing zkSNARK*. <https://github.com/arkworks-rs/marlin>.

- [CHMMVW20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. P. Ward. “Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS”. In: *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT’20. 2020, pp. 738–768.
- [Chr] *Certificate Transparency in Chrome*. <https://sites.google.com/site/certificate-transparency/certificate-transparency-in-chrome>.
- [Cis14] Cisco. *The Internet of Things Reference Model*. Tech. rep. Cisco, 2014.
- [CL99] M. Castro and B. Liskov. “Practical Byzantine fault tolerance”. In: *OSDI*. 1999.
- [CM06] K. Chaudhuri and N. Mishra. “When Random Sampling Preserves Privacy”. In: *CRYPTO*. 2006.
- [CM16] M. Chase and S. Meiklejohn. “Transparency overlays and applications”. In: *CCS*. 2016.
- [CMT12] G. Cormode, M. Mitzenmacher, and J. Thaler. “Practical Verified Computation with Streaming Interactive Proofs”. In: *Proceedings of the 4th Symposium on Innovations in Theoretical Computer Science*. ITCS ’12. 2012, pp. 90–112.
- [CMW06] J. Crampton, K. Martin, and P. Wild. “On key assignment for hierarchical access control”. In: *CSFW*. 2006.
- [Coh] W. W. Cohen. *Enron Email Dataset*. www.cs.cmu.edu/~enron.
- [Con] *CONIKS Go*. github.com/coniks-sys/coniks-go.
- [COS20a] A. Chiesa, D. Ojha, and N. Spooner. “Fractal: Post-quantum and transparent recursive proofs from holography”. In: *Eurocrypt*. 2020.
- [COS20b] A. Chiesa, D. Ojha, and N. Spooner. “Fractal: Post-Quantum and Transparent Recursive Proofs from Holography”. In: *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’20. 2020.
- [CPS94] J. L. Camenisch, J. Piveteau, and M. A. Stadler. “Blind signatures based on the discrete logarithm problem”. In: *EUROCRYPT*. 1994.
- [Cry] Crypho. *Enterprise communications with end-to-end encryption*. <https://www.crypho.com/>.
- [CSPLM15] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri. “Efficient gossip protocols for verifying the consistency of certificate logs”. In: *CNS*. 2015.
- [CSPZKA16] R. Cheng, W. Scott, B. Parno, I. Zhang, A. Krishnamurthy, and T. Anderson. *Talek: A Private Publish-Subscribe Protocol*. Tech. rep. University of Washington CSE, 2016.
- [CTY11] G. Cormode, J. Thaler, and K. Yi. “Verifying computations with streaming interactive proofs”. In: *Proceedings of the VLDB Endowment* 5.1 (2011), pp. 25–36.
- [CVPPFR19] S. Contiu, S. Vaucher, R. Pires, M. Pasin, P. Felber, and L. Réveillère. “Anonymous and confidential file sharing over untrusted clouds”. In: *SRDS* (2019).
- [CWWZ10] S. Chen, R. Wang, X. Wang, and K. Zhang. “Side-channel leaks in web applications: A reality today, a challenge tomorrow”. In: *S&P*. 2010.

- [Dav] J. Davis. *The 10 Biggest Healthcare Data Breaches of 2019, So Far*. <https://healthitsecurity.com/news/the-10-biggest-healthcare-data-breaches-of-2019-so-far>. Sep. 12, 2019.
- [DCS07] P. Dutta, D. E. Culler, and S. Shenker. “Procrastination Might Lead to a Longer and More Useful Life”. In: *HotNets*. 2007.
- [DD08] G. Danezis and C. Diaz. *A Survey of Anonymous Communication Channels*. Technical Report MSR-TR-2008-35, Microsoft Research. 2008.
- [DF02] Y. Dodis and N. Fazio. “Public key broadcast encryption for stateless receivers”. In: *DRM*. 2002.
- [DGHS16] B. Dowling, F. Günther, U. Herath, and D. Stebila. “Secure logging schemes and certificate transparency”. In: *ESORICS*. 2016.
- [DH+13] S. Dawson-Haggerty et al. “BOSS: Building Operating System Services”. In: *NSDI*. 2013.
- [DHJTOC10] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. E. Culler. “sMAP: A Simple Measurement and Actuation Profile for Physical Information”. In: *SenSys*. 2010.
- [Dia21] B. E. Diamond. “Many-out-of-Many Proofs and Applications to Anonymous Zether”. In: *IEEE S & P*. 2021.
- [DLB59] R. De La Briandais. “File searching using variable length keys”. In: *WJCC*. 1959.
- [DMNS06] C. Dwork, F. McSherry, K. Nissim, and A. Smith. “Calibrating Noise to Sensitivity in Private Data Analysis”. In: *TCC*. 2006.
- [DMS04a] R. Dingledine, N. Mathewson, and P. Syverson. *Tor: The second-generation onion router*. Tech. rep. 2004.
- [DMS04b] R. Dingledine, N. Mathewson, and P. Syverson. “Tor: The Second-Generation Onion Router”. In: *USENIX Security Symposium*. 2004.
- [DPP16] R. Dahlberg, T. Pulls, and R. Peeters. “Efficient sparse merkle trees”. In: *NordSec*. 2016.
- [DPVHJK19] R. Dahlberg, T. Pulls, J. Vestin, T. Høiland-Jørgensen, and A. Kassler. “Aggregation-Based Gossip for Certificate Transparency”. In: (2019).
- [DR14] C. Dwork and A. Roth. “The Algorithmic Foundations of Differential Privacy.” In: *Foundations and Trends in Theoretical Computer Science* 9.3-4 (2014), pp. 211–407. URL: <http://dblp.uni-trier.de/db/journals/fttcs/fttcs9.html#DworkR14>.
- [Dra20] J. Drake. *PLONK without FFTs*. 2020. URL: <https://www.youtube.com/watch?v=ffXgxv1CBvo>.
- [DSST86] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. “Making data structures persistent”. In: *STOC*. 1986.
- [Dwo15] M. J. Dworkin. *SHA-3 standard: Permutation-based hash and extendable-output functions*. Tech. rep. 2015.
- [ELC] A. Eijdenberg, B. Laurie, and A. Cutter. *Verifiable Data Structures*. <https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf>.
- [Ele] D. Electronics. *ATSAMR21E18A-MU Microchip Technology*. Feb. 8, 2019.
- [ema] P. F. encrypted email. <https://protonmail.com/>.

- [Eth] *Ethereum*. <https://ethereum.org/>.
- [EW17] M. Egorov and M. Wilkison. “NuCypher KMS: Decentralized key management system”. In: *CoRR* (2017). URL: <http://arxiv.org/abs/1707.06140>.
- [EY09] M. Edman and B. Yener. “On Anonymity in an Electronic Society: A Survey of Anonymous Communication Systems”. In: *ACM Comput. Surv.* 42.1 (Dec. 2009).
- [FC15] G. Fierro and D. E. Culler. *XBOS: An Extensible Building Operating System*. Tech. rep. EECS Department, University of California, Berkeley, 2015. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-197.html>.
- [FDPFSS14] S. Fahl, S. Dechand, H. Perl, F. Fischer, J. Smrcek, and M. Smith. “Hey, nsa: Stay away from my market! future proofing app markets against powerful attackers”. In: *CCS*. 2014.
- [Fel09] M. C. Feldmeier. “Personalized Building Comfort Control”. PhD thesis. MIT, 2009.
- [Fil] *Filecoin*. <https://filecoin.io>. Jan. 19, 2018.
- [FKL18] G. Fuchsbauer, E. Kiltz, and J. Loss. “The Algebraic Group Model and its Applications”. In: *Proceedings of the 38th Annual International Cryptology Conference*. CRYPTO ’18. 2018, pp. 33–62.
- [FMBBSH16] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. “How Secure is TextSecure?” In: *EuroS&P*. 2016.
- [FS86] A. Fiat and A. Shamir. “How to prove yourself: practical solutions to identification and signature problems”. In: *Proceedings of the 6th Annual International Cryptology Conference*. CRYPTO ’86. 1986, pp. 186–194.
- [FZFF10] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. “SPORC: Group Collaboration using Untrusted Cloud Resources.” In: *OSDI*. 2010.
- [Gab19] A. Gabizon. *Improved prover efficiency and SRS size in a Sonic-like system*. Cryptology ePrint Archive, Report 2019/601. 2019.
- [Gab20] A. Gabizon. *Lineal Protocol*. Available at <https://hackmd.io/aWxth2dASPaGVrXiGg1Cmg?view>. 2020.
- [GCL] W. Gordon, A. Chopra, and A. Landman. *Patient-Led Data Sharing — A New Paradigm for Electronic Health Data*. <https://catalyst.nejm.org/patient-led-health-data-paradigm/>. Sep. 12, 2019.
- [GD] S. Ghemawat and J. Dean. *LevelDB*. <https://github.com/google/leveldb>.
- [GGPR13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. “Quadratic Span Programs and Succinct NIZKs without PCPs”. In: *Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT ’13. 2013, pp. 626–645.
- [GH09] C. Gentry and S. Halevi. “Hierarchical Identity Based Encryption with Polynomially Many Levels”. In: *TCC*. 2009.
- [GIKM00] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. “Protecting Data Privacy in Private Information Retrieval Schemes”. In: *J. Comput. Syst. Sci.* 60.3 (2000).

- [GKMMM18] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. “Updatable and Universal Common Reference Strings with Applications to zk-SNARKs”. In: *Proceedings of the 38th Annual International Cryptology Conference*. CRYPTO ’18. 2018, pp. 698–728.
- [GKR15] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. “Delegating Computation: Interactive Proofs for Muggles”. In: *Journal of the ACM* 62.4 (2015), 27:1–27:64.
- [GLSTW21] A. Golovnev, J. Lee, S. T. V. Setty, J. Thaler, and R. S. Wahby. *Brakedown: Linear-time and post-quantum SNARKs for RICS*. Cryptology ePrint Archive, Report 2021/1043. 2021.
- [GMNRS16] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. “Breaking Web Applications Built On Top of Encrypted Data”. In: *CCS*. 2016.
- [GMR89] S. Goldwasser, S. Micali, and C. Rackoff. “The knowledge complexity of interactive proof systems”. In: *SIAM Journal on Computing* 18.1 (1989). Preliminary version appeared in *STOC ’85.*, pp. 186–208.
- [GO96] O. Goldreich and R. Ostrovsky. “Software protection and simulation on oblivious RAMs”. In: *JACM* (1996).
- [Gol07] O. Goldreich. *Foundations of cryptography: volume 1, basic tools*. 2007.
- [Gooa] Google. *HTTPS encryption on the web*. transparencyreport.google.com/https/certificates.
- [Goob] Google. *Key Transparency Contributions*. github.com/google/keytransparency/blob/master/docs/design-improvements.md.
- [Gooc] Google. *Key Transparency New Design*. github.com/google/keytransparency/blob/master/docs/design_new.md.
- [Good] Google. *New Design Doc*. github.com/google/keytransparency/pull/1469.
- [GPSW06] V. Goyal, O. Pandey, A. Sahai, and B. Waters. “Attribute-based encryption for fine-grained access control of encrypted data”. In: *CCS*. 2006.
- [Gro10] J. Groth. “Short Pairing-Based Non-interactive Zero-Knowledge Arguments”. In: *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT ’10. 2010, pp. 321–340.
- [Gro16] J. Groth. “On the Size of Pairing-Based Non-interactive Arguments”. In: *Proceedings of the 35th Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’16. 2016, pp. 305–326.
- [GRS99] D. Goldschlag, M. Reed, and P. Syverson. “Onion Routing for Anonymous and Private Internet Connections”. In: *Communications of the ACM* 42 (1999), pp. 39–41.
- [GS02] C. Gentry and A. Silverberg. “Hierarchical ID-Based Cryptography”. In: *ASIACRYPT*. 2002.
- [GSMB03a] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. “SiRiUS: Securing remote untrusted storage”. In: *NDSS*. 2003.
- [GSMB03b] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. “SiRiUS: Securing Remote Untrusted Storage”. In: *NDSS*. 2003.

- [GSML16] W. C. Garrison, A. Shull, S. Myers, and A. J. Lee. “On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud”. In: *S&P*. 2016.
- [GW20] A. Gabizon and Z. J. Williamson. *plookup: A simplified polynomial protocol for lookup tables*. Cryptology ePrint Archive, Report 2020/315. 2020.
- [GWC19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Report 2019/953. 2019.
- [HABSG17] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg. “TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub”. In: *NDSS*. 2017.
- [HAJSS14] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. “Shadowcrypt: Encrypted web applications for everyone”. In: *CCS*. 2014.
- [Ham] *Hamilton IoT*. <https://hamiltoniot.com/>.
- [HB17] R. Hurst and G. Belvin. *Security Through Transparency*. <https://security.googleblog.com/2017/01/security-through-transparency.html>. 2017.
- [HC04] H.-F. Huang and C.-C. Chang. “A new cryptographic key assignment scheme with time-constraint access control in a hierarchy”. In: *Computer Standards & Interfaces* (2004).
- [HC17] B. Hof and G. Carle. “Software distribution transparency and auditability”. In: *arXiv:1711.07278* (2017).
- [HHKYP] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa. *A Go library for MerkleSquare: A Low-Latency Transparency Log System*. <https://github.com/ucbrise/MerkleSquare>.
- [HHKYP21] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa. “Merkle²: A Low-Latency Transparency Log System”. In: *Proceedings of the 42nd IEEE Symposium on Security and Privacy*. SP ’21. 2021, pp. 285–303.
- [HJP05] Y.-C. Hu, M. Jakobsson, and A. Perrig. “Efficient Constructions for One-Way Hash Chains”. In: *ACNS*. 2005.
- [HK18] J. Hviid and M. B. Kjaergaard. “Activity-Tracking Service For Building Operating Systems”. In: *PerCom*. 2018.
- [HKP20] Y. Hu, S. Kumar, and R. A. Popa. “Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust”. In: *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. NSDI’20. 2020, pp. 851–877.
- [HLZZ15a] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. “Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis”. In: *SOSP*. 2015.
- [HLZZ15b] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. “Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis”. In: *SOSP*. 2015.
- [HMPS14] S. Hohenberger, S. A. Myers, R. Pass, and A. Shelat. “ANONIZE: A Large-Scale Anonymous Survey System”. In: *IEEE S & P*. 2014.

- [Hop] D. Hoppe. *Blockchain Use Cases: Electronic Health Records*. https://gammalaw.com/blockchain_use_cases_electronic_health_records/. Sep. 12, 2019.
- [HOWW19a] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs. “Private Anonymous Data Access”. In: 2019.
- [HOWW19b] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs. “Private anonymous data access”. In: *Eurocrypt*. 2019.
- [HR18] J. Holmgren and R. Rothblum. “Delegating Computations with (Almost) Minimal Time and Space Overhead”. In: *Proceedings of the 59th Annual IEEE Symposium on Foundations of Computer Science*. FOCS ’18. 2018, pp. 124–135.
- [HS90] S. Haber and W. S. Stornetta. “How to time-stamp a digital document”. In: *EUROCRYPT*. 1990.
- [HW90] M. P. Herlihy and J. M. Wing. “Linearizability: A correctness condition for concurrent objects”. In: *TOPLAS* (1990).
- [Ide18] Identity Theft Resource Center. “At Mid-Year, U.S. Data Breaches Increase at Record Pace”. In: *ITRC*. 2018.
- [IKK12] M. S. Islam, M. Kuzu, and M. Kantarcioglu. “Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.” In: *NDSS*. 2012.
- [IKVR88] S. S. Iyengar, R. L. Kashyap, V. K. Vaishnavi, and N. S. V. Rao. “Multidimensional data structures: Review and outlook”. In: *ADV COMPUT* (1988).
- [Imi] *imix: Low-Power IoT Research Platform*. <https://github.com/helena-project/imix>. 2017.
- [Inc] T. Inc. *End-to-end encrypted cloud storage*. tresorit.com.
- [KB16] T. Kim and R. Barbulescu. “Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case”. In: *CRYPTO*. 2016.
- [Kep15] B. Kepes. *Some scary (for some) statistics around file sharing usage*. <https://www.computerworld.com/article/2991924/some-scary-for-some-statistics-around-file-sharing-usage.html>. 2015.
- [Key] Keybase.io. *End-to-end encryption for things that matter*. <https://keybase.io/>.
- [KFKC12] A. Krioukov, G. Fierro, N. Kitaev, and D. E. Culler. “Building Application Stack (BAS)”. In: *BuildSys*. 2012.
- [KFPC16] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. “Verena: End-to-end integrity protection for web applications”. In: *S&P*. 2016.
- [KH12] S. M. Khan and K. W. Hamlen. “AnonymousCloud: A data ownership privacy provider framework in cloud computing”. In: *TrustCom*. 2012.
- [KHAPC] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler. *Golang implementation of JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT*. <https://github.com/ucbrise/jedi-protocol-go>.

- [KHAPC19] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler. “JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT”. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security. 2019, pp. 1519–1536.
- [KHPJG13] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor. “Accountable key infrastructure (AKI) a proposal for a public-key validation infrastructure”. In: *WWW*. 2013.
- [Kil92] J. Kilian. “A note on efficient zero-knowledge proofs and arguments”. In: *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*. STOC ’92. 1992, pp. 723–732.
- [Kim+18] H.-S. Kim et al. “System Architecture Directions for Post-SoC/32-bit Networked Sensors”. In: *SenSys*. 2018.
- [KKKP] J. Kalidhindi, A. Kazorian, A. Khera, and C. Pari. “Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree”. In: ().
- [KKSK16] Y. Kawahara, T. Kobayashi, M. Scott, and A. Kato. *Barreto-Naehrig Curves*. Tech. rep. <https://datatracker.ietf.org/doc/html/draft-kasamatsu-bncurves-02>. Internet-Draft draft-kasamatsu-bncurves-02. Internet Engineering Task Force., 2016.
- [KL15] B. H. Kim and D. Lie. “Caelus: Verifying the consistency of cloud services with battery-powered devices”. In: *S&P*. 2015.
- [KMP20] A. Kothapalli, E. Masserova, and B. Parno. *A Direct Construction for Asymptotically Optimal zkSNARKs*. Cryptology ePrint Archive, Report 2020/1318. 2020.
- [KPK17] N. P. Karvelas, A. Peter, and S. Katzenbeisser. “Using Oblivious RAM in Genomic Studies”. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. 2017.
- [KRSWF03] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. “Plutus: Scalable Secure File Sharing on Untrusted Storage”. In: *FAST*. 2003.
- [Kub+00] J. Kubiatowicz et al. “OceanStore: An Architecture for Global-scale Persistent Storage”. In: *ASPLOS*. 2000.
- [KZFMSG18] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais. *Commit-Chains: Secure, Scalable Off-Chain Payments*. <https://eprint.iacr.org/2018/642>. 2018.
- [KZG10] A. Kate, G. M. Zaverucha, and I. Goldberg. “Constant-Size Commitments to Polynomials and Their Applications”. In: *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT ’10. 2010, pp. 177–194.
- [Lau] B. Laurie. *Improving SSL certificate security*. security.googleblog.com/2011/04/improving-ssl-certificate-security.html.
- [Lau14] B. Laurie. “Certificate transparency”. In: *CACM* (2014).
- [LCSJLB14] B. Lau, S. P. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva. “Mimesis Aegis: A Mimicry Privacy Shield—A System’s Approach to Data Privacy on Public Cloud.” In: *USENIX Security*. 2014.
- [Lee+01] W. Lee et al. “Real time data mining-based intrusion detection”. In: *DISCEX*. 2001.

- [Lee20] J. Lee. *Dory: Efficient, Transparent arguments for Generalised Inner Products and Polynomial Commitments*. Cryptology ePrint Archive, Report 2020/1274. 2020.
- [Lem] R. Lemos. *Home Depot estimates data on 56 million cards stolen by cybercriminals*. <https://arstechnica.com/information-technology/2014/09/home-depot-estimates-data-on-56-million-cards-stolen-by-cybercriminals/>. Apr. 21, 2019.
- [Lev84] H. M. Levy. *Capability-based computer systems*. 1984.
- [LFKN92] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. “Algebraic Methods for Interactive Proof Systems”. In: *Journal of the ACM* 39.4 (1992), pp. 859–868.
- [LGZ18] D. Lazar, Y. Gilad, and N. Zeldovich. “Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI ’18. 2018, 711–725.
- [Li+19] B. Li et al. “Certificate transparency in the wild: Exploring the reliability of monitors”. In: *CCS*. 2019.
- [Lip12] H. Lipmaa. “Progression-Free Sets and Sublinear Pairing-Based Non-Interactive Zero-Knowledge Arguments”. In: *Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography*. TCC ’12. 2012, pp. 169–189.
- [LK12] B. Laurie and E. Kasper. “Revocation transparency”. In: *Google Research* (2012).
- [LKL13] A. Langley, E. Kasper, and B. Laurie. *RFC 6962-Certificate Transparency*. 2013.
- [LKMS04] J. Li, M. Krohn, D. Mazières, and D. Shasha. “Secure Untrusted Data Repository (SUNDR)”. In: *OSDI*. 2004.
- [LLK13a] B. Laurie, A. Langley, and E. Kasper. *Certificate transparency*. Tech. rep. 2013.
- [LLK13b] B. Laurie, A. Langley, and E. Kasper. “Certificate Transparency”. In: *RFC* (2013), pp. 1–27.
- [LLLMSL14] C. Li, Z. Li, M. Li, F. Meggers, A. Schlueter, and H. B. Lim. “Energy Efficient HVAC System with Distributed Sensing and Control”. In: *ICDCS*. 2014.
- [LLWQNF] W. Liu, J. Liu, Q. Wu, B. Qin, D. Naccache, and H. Ferradi. *Compact CCA2-secure Hierarchical Identity-Based Broadcast Encryption for Fuzzy-entity Data Sharing*. Cryptology ePrint Archive, Report 2016/634.
- [Lok+19] M. Lokhava et al. “Fast and secure global payments with Stellar”. In: *SOSP*. 2019.
- [LPY12a] B. Libert, T. Peters, and M. Yung. “Group Signatures with Almost-for-Free Revocation”. In: *CRYPTO*. 2012.
- [LPY12b] B. Libert, T. Peters, and M. Yung. “Scalable Group Signatures with Revocation”. In: *EUROCRYPT*. 2012.
- [LSW10] A. Lewko, A. Sahai, and B. Waters. “Revocation Systems with Very Small Private Keys”. In: *S&P*. 2010.
- [LV09] B. Libert and D. Vergnaud. “Adaptive-ID Secure Revocable Identity-Based Encryption”. In: *CT-RSA*. 2009.

- [LYKGM19] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller. “Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication”. In: *CCS*. 2019.
- [Mah+11] P. Mahajan et al. “Depot: Cloud storage with minimal trust”. In: *TOCS* (2011).
- [Man] R. Mandalia. *SECURITY BREACH IN CA NETWORKS -COMODO, DIGINOTAR, GLOBALSIGN*. blog.isc2.org/isc2_blog/2012/04/test.html.
- [Mau96] U. Maurer. “Modelling a public-key infrastructure”. In: *ESORICS*. 1996.
- [MB02] P. Maniatis and M. Baker. “Secure History Preservation Through Timeline Entanglement”. In: *USENIX Security*. 2002.
- [MBBF15] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. “CONIKS: Bringing Key Transparency to End Users”. In: *Proceedings of the 24th USENIX Security Symposium*. USENIX Security 15. 2015, pp. 383–398.
- [MBKM19] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. “Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings”. In: *Proceedings of the 26th ACM Conference on Computer and Communications Security*. CCS ’19. 2019, pp. 2111–2128.
- [Med] *Medicalchain - Blockchain for electronic health records*. <https://medicalchain.com>. Sep. 12, 2019.
- [Mer79] R. Merkle. *Merkle tree patent*. 1979.
- [Mer87] R. C. Merkle. “A digital signature based on a conventional encryption function”. In: *Eurocrypt*. 1987.
- [MHWK16] M. Milutinovic, W. He, H. Wu, and M. Kanwal. “Proof of luck: An efficient blockchain consensus protocol”. In: *SysTEX*. 2016.
- [MMR00] D. Malkhi, M. Merritt, and O. Rodeh. “Secure reliable multicast protocols in a WAN”. In: *Dist. Computing* (2000).
- [MMRS15] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. “Privacy and access control for outsourced personal records”. In: *S&P*. 2015.
- [MMRS17] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. “Maliciously secure multi-client ORAM”. In: *ACNS*. 2017.
- [MR17] S. Matsumoto and R. M. Reischuk. “IKP: Turning a PKI around with decentralized automated incentives”. In: *S&P*. 2017.
- [MR97] D. Malkhi and M. Reiter. “A high-throughput secure reliable multicast protocol”. In: *Computer Security* (1997).
- [MRK18] A. Mehanovic, T. H. Rasmussen, and M. B. Kjærgaard. “Brume - A Horizontally Scalable and Fault Tolerant Building Operating System”. In: *IoTDI*. 2018.
- [MRV99] S. Micali, M. Rabin, and S. Vadhan. “Verifiable random functions”. In: *FOCS*. 1999.
- [MS02] D. Mazières and D. Shasha. “Building secure file systems out of Byzantine storage”. In: *PODC*. 2002.

- [MWC10] A. Mettler, D. A. Wagner, and T. Close. “Joe-E: A Security-Oriented Subset of Java.” In: *NDSS*. 2010.
- [Nah04] F. F. Nah. “A study on tolerable waiting time: how long are web users willing to wait?” In: *Behaviour & Information Technology* (2004).
- [Nak08] S. Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: (2008).
- [Nak19] S. Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. 2019.
- [NB13] A. Niemann and J. Brendel. *A survey on ca compromises*. 2013.
- [Net+16] A. L. M. Neto et al. “Aot: Authentication and access control for the entire iot device life-cycle”. In: *SenSys*. 2016.
- [Ngu05] L. Nguyen. “Accumulators from bilinear pairings and applications”. In: *CT-RSA*. 2005.
- [Nik+17] K. Nikitin et al. “CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds”. In: *USENIX Security*. 2017.
- [NNL01] D. Naor, M. Naor, and J. Lotspiech. “Revocation and tracing schemes for stateless receivers”. In: *CRYPTO*. 2001.
- [Ope] *Open mHealth*. <http://www.openmhealth.org/>. Sep. 19, 2019.
- [OS97] R. Ostrovsky and V. Shoup. “Private Information Storage (Extended Abstract)”. In: *STOC*. 1997, pp. 294–303.
- [Par] *Particle Mesh*. <https://www.particle.io/mesh>. Feb. 2, 2019.
- [Per99] R. Perlman. “An overview of PKI trust models”. In: *IEEE network* (1999).
- [PLS19] R. d. Pino, V. Lyubashevsky, and G. Seiler. “Short Discrete Log Proofs for FHE and Ring-LWE Ciphertexts”. In: *Proceedings of the 22nd International Conference on Practice and Theory of Public-Key Cryptography*. PKC ’19. 2019, pp. 344–373.
- [Pop+14] R. A. Popa et al. “Building Web Applications on Top of Encrypted Data Using Mylar.” In: *NSDI*. 2014.
- [PP12] V. Pacheco and R. Puttini. “SaaS Anonymous Cloud Service Consumption Structure”. In: *ICDCS*. 2012.
- [Pre] PreVeil Inc. *PreVeil: Encrypted email and file sharing for the enterprise*. www.preveil.com.
- [Pri] Privly Inc. *Privly*. priv.ly.
- [PS18] R. Pass and E. Shi. “Thunderella: Blockchains with optimistic instant confirmation”. In: *Eurocrypt*. 2018.
- [PST13] C. Papamanthou, E. Shi, and R. Tamassia. “Signatures of Correct Computation”. In: *Proceedings of the 10th Theory of Cryptography Conference*. TCC ’13. 2013, pp. 222–242.
- [PSWCT01] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. “SPINS: Security Protocols for Sensor Networks”. In: *MobiCom*. 2001.
- [Rad] S. Radicati. *Email Statistics Report, 2014-2018*. www.radicati.com/wp/wp-content/uploads/2014/01/Email-Statistics-Report-2014-2018-Executive-Summary.pdf.

- [RD01] A. Rowstron and P. Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In: *Middleware*. 2001.
- [RFHKS01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. “A Scalable Content-addressable Network”. In: *SIGCOMM*. 2001.
- [RMSK14] T. Ruffing, P. Moreno-Sanchez, and A. Kate. “Coinshuffle: Practical decentralized coin mixing for bitcoin”. In: *ESORICS*. 2014.
- [RMSK17] T. Ruffing, P. Moreno-Sanchez, and A. Kate. “P2P Mixing and Unlinkable Bitcoin Transactions.” In: *NDSS*. 2017.
- [RR22] N. Ron-Zewi and R. D. Rothblum. “Proving as Fast as Computing: Succinct Arguments with Constant Prover Overhead”. In: *Proceedings of the 54th Annual ACM Symposium on Theory of Computing*. STOC ’22. 2022.
- [RRR16] O. Reingold, R. Rothblum, and G. Rothblum. “Constant-Round Interactive Proofs for Delegating Computation”. In: *Proceedings of the 48th ACM Symposium on the Theory of Computing*. STOC ’16. 2016, pp. 49–62.
- [Rsa] <https://github.com/cryptoballot/rsablind>.
- [RWV13] S. Raza, L. Wallgren, and T. Voigt. “SVELTE: Real-time intrusion detection in the Internet of Things”. In: *Ad hoc networks* (2013).
- [Rya14] M. D. Ryan. “Enhanced Certificate Transparency and End-to-End Encrypted Mail.” In: *NDSS*. 2014.
- [SATJ18] S. T. V. Setty, S. Angel, G. Trinabh, and L. Jonathan. “Proving the correct execution of concurrent services in zero-knowledge”. In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’18. 2018, pp. 339–356.
- [SBDHR18] H. Shafagh, L. Burkhalter, S. Duquenooy, A. Hithnawi, and S. Ratnasamy. “Droplet: Decentralized Authorization for IoT Data Streams”. In: *CoRR* (2018).
- [SCCKMS10] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. “Venus: Verification for untrusted cloud storage”. In: *CCSW*. 2010.
- [SCSL11] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li. “Oblivious RAM with $O((\log N)^3)$ worst-case cost”. In: *ASIACRYPT*. 2011.
- [SE13a] J. H. Seo and K. Emura. “Efficient Delegation of Key Generation and Revocation Functionalities in Identity-Based Encryption”. In: *CT-RSA*. 2013.
- [SE13b] J. H. Seo and K. Emura. “Revocable Identity-Based Encryption Revisited: Security Model and Construction”. In: *PKC*. 2013.
- [SE15] J. H. Seo and K. Emura. “Revocable Hierarchical Identity-Based Encryption: History-Free Update, Security Against Insiders, and Short Ciphertexts”. In: *CT-RSA*. 2015.
- [Sea18] T. Seals. *17% of Workers Fall for Social Engineering Attacks*. 2018.
- [Sec] *Secret Double Octopus | Passwordless High Assurance Authentication*. <https://doubleoctopus.com>. Apr. 21, 2019.
- [Set19] S. Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup*. Cryptology ePrint Archive, Report 2019/550. 2019.

- [Set20] S. T. V. Setty. “Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup”. In: *40th Annual International Cryptology Conference*. CRYPTO '20. 2020, pp. 704–737.
- [SH12] M. Srivatsa and M. Hicks. “Deanonymizing mobility traces: Using social network as a side-channel”. In: *CCS*. 2012.
- [Sha] J. Sharp. *Will Healthcare See Ethical Patient Data Exchange?* <https://www.idigitalhealth.com/news/healthcare-ethical-patient-data-exchange-cms-rule>. Sep. 12, 2019.
- [Sha79] A. Shamir. “How to share a secret”. In: *CACM* (1979).
- [SHBFD17] H. Shafagh, A. Hithnawi, L. Burkhalter, P. Fischli, and S. Duquennoy. “Secure Sharing of Partially Homomorphic Encrypted IoT Data”. In: *SenSys*. 2017.
- [Sia] *Sia*. <https://sia.tech>. Apr. 16, 2019.
- [SM12] K. Scarfone and P. Mell. *Guide to intrusion detection and prevention systems (idps)*. Tech. rep. NIST, 2012.
- [SMP14] P. Szalachowski, S. Matsumoto, and A. Perrig. “PoliCert: Secure and flexible TLS certificate management”. In: *CCS*. 2014.
- [Sol] *Solace Cloud*. <https://solace.com>. Jan. 17, 2018.
- [SRSB15] M. Singh, M. Rajan, V. Shivraj, and P. Balamuralidhar. “Secure mqtt for internet of things (iot)”. In: *CSNT*. 2015.
- [SS13] E. Stefanov and E. Shi. “Oblivstore: High performance oblivious cloud storage”. In: *S&P*. 2013.
- [SSABD18] F. Shirazi, M. Simeonovski, M. R. Asghar, M. Backes, and C. Diaz. “A Survey on Routing in Anonymous Communication Protocols”. In: (2018).
- [SSF99] J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS: A Fast Capability System”. In: *SOSP*. 1999.
- [Ste+13] E. Stefanov et al. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: *CCS*. 2013.
- [Sto] *Decentralized Cloud Storage — Storj*. <https://storj.io>. Apr. 16, 2019.
- [SW21a] E. Shi and K. Wu. “Non-Interactive Anonymous Router”. In: *Proceedings of the 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT 2021. 2021, pp. 489–520.
- [SW21b] E. Shi and K. Wu. “Non-Interactive Anonymous Router”. In: *Eurocrypt*. 2021.
- [Syt+16] E. Syta et al. “Keeping authorities" honest or bust" with decentralized witness cosigning”. In: *S&P*. 2016.
- [SZEALT16] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro. “Taostore: Overcoming asynchronicity in oblivious data storage”. In: *S&P*. 2016.
- [TBPPTD19] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas. “Transparency logs via append-only authenticated dictionaries”. In: *CCS*. 2019.

- [TD17] A. Tomescu and S. Devadas. “Catena: Efficient non-equivocation via bitcoin”. In: *S&P*. 2017.
- [TDG16] R. R. Toledo, G. Danezis, and I. Goldberg. “Lower-Cost Epsilon-Private Information Retrieval”. In: *Proc. Priv. Enhancing Technol.* 2016.4 (2016), pp. 184–201.
- [TFJ16] V. Tron, A. Fischer, and N. Johnson. *Smash-proof: Auditable storage for Swarm secured by masked audit secret hash*. Tech. rep. Ethersphere, 2016.
- [TGLZZ17a] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. “Stadium: A Distributed Metadata-Private Messaging System”. In: *SOSP*. 2017.
- [TGLZZ17b] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. “Stadium: A Distributed Metadata-Private Messaging System”. In: *SOSP*. 2017.
- [Tha13] J. Thaler. “Time-Optimal Interactive Proofs for Circuit Evaluation”. In: *Proceedings of the 33rd Annual International Cryptology Conference*. CRYPTO ’13. 2013, pp. 71–89.
- [TKR14] M. A. Tariq, B. Koldehofe, and K. Rothermel. “Securing broker-less publish/subscribe systems using identity-based encryption”. In: *TPDS* (2014).
- [TS16] A. Taly and A. Shankar. “Distributed Authorization in Vanadium”. In: *FOSAD VIII*. 2016.
- [Tze02] W.-G. Tzeng. “A time-bound cryptographic key assignment scheme for access control in a hierarchy”. In: *TKDE* (2002).
- [Vad17] S. Vadhan. “The Complexity of Differential Privacy”. In: 2017.
- [VB+18] J. Van Bulck et al. “Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution”. In: *USENIX Security*. 2018.
- [Vir] Virtru Inc. *Virtru: Email Encryption and Data Protection Solutions*. www.virtru.com.
- [Vol] VOLTTTRON. <https://voltttron.org/>. Jan. 23, 2019.
- [WBMLM06] S. A Weil, S. A Brandt, E. L Miller, D. D. Long, and C. Maltzahn. “Ceph: A scalable, high-performance distributed file system”. In: *OSDI*. 2006.
- [WCS15] X. Wang, H. Chan, and E. Shi. “Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound”. In: *CCS*. 2015.
- [WES17] Y. Watanabe, K. Emura, and J. H. Seo. “New Revocable IBE in Prime-Order Groups: Adaptively Secure, Decryption Key Exposure Resistant, and with Short Public Parameters”. In: *CT-RSA*. 2017.
- [Wha] A. Whalley. *Distusting WoSign and StartCom Certificates*. security.googleblog.com/2016/10/distusting-wosign-and-startcom.html.
- [Wha12] WhatsApp. *WhatsApp’s Privacy Notice*. www.whatsapp.com/legal/?doc=privacy-policy. 2012.
- [WLCWZJ20] Z. Wang, J. Lin, Q. Cai, Q. Wang, D. Zha, and J. Jing. “Blockchain-based certificate transparency and revocation transparency”. In: *TDSC* (2020).
- [WLW10] G. Wang, Q. Liu, and J. Wu. “Hierarchical attribute-based encryption for fine-grained access control in cloud storage services”. In: *CCS*. 2010.

- [WLWG11] G. Wang, Q. Liu, J. Wu, and M. Guo. “Hierarchical attribute-based encryption and scalable user revocation for sharing data in cloud servers”. In: *Computers & Security* (2011).
- [WMZV16] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan. “Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds”. In: *NSDI*. 2016.
- [Woo14] G. Wood. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* (2014).
- [WPGKHK21] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt. “SoK: Decentralized Finance (DeFi)”. In: *CoRR* (2021).
- [WTSB16] D. J. Wu, A. Taly, A. Shankar, and D. Boneh. “Privacy, discovery, and authentication for the Internet of things”. In: *ESORICS*. 2016.
- [WTSTW18] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. “Doubly-Efficient zkSNARKs Without Trusted Setup”. In: *Proceedings of the 39th IEEE Symposium on Security and Privacy*. S&P ’18. 2018, pp. 926–943.
- [WZCPS18] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. “DIZK: A Distributed Zero Knowledge Proof System”. In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Security ’18. 2018, pp. 675–692.
- [XZZPS19] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. “Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation”. In: *Proceedings of the 39th Annual International Cryptology Conference*. CRYPTO ’19. 2019, pp. 733–764.
- [YCR16] J. Yu, V. Cheval, and M. Ryan. “DTKI: A new formalized PKI with verifiable trusted parties”. In: *The Computer Journal* (2016).
- [YFDL04] D. Yao, N. Fazio, Y. Dodis, and A. Lysyanskaya. “ID-based Encryption for Complex Hierarchies with Applications to Forward Security and Broadcast Encryption”. In: *CCS*. 2004.
- [YHE02] W. Ye, J. Heidemann, and D. Estrin. “An energy-efficient MAC protocol for wireless sensor networks”. In: *INFOCOM*. 2002.
- [YMRGA19] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. “Hotstuff: Bft consensus with linearity and responsiveness”. In: *PODC*. 2019.
- [YWRL10] S. Yu, C. Wang, K. Ren, and W. Lou. “Achieving Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing”. In: *INFOCOM*. 2010.
- [Zcaa] Zcash. <https://z.cash/>.
- [Zcab] Zcash. *Zcash*. <https://z.cash/>.
- [ZcashMPC] *The Zcash Ceremony*. <https://z.cash/blog/the-design-of-the-ceremony.html>. 2016.
- [ZEEJVR17] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. Van Renesse. “REM: Resource-efficient mining for blockchains”. In: *USENIX Security*. 2017.
- [Zeta] K. Zetter. *An Unprecedented Look at Stuxnet, the World’s First Digital Weapon*. <https://www.wired.com/2014/11/countdown-to-zero-day-stuxnet/>. Apr. 21, 2019.

- [Zetb] K. Zetter. ‘Google’ Hackers Had Ability to Alter Source Code. <https://www.wired.com/2010/03/source-code-hacks/>. Apr. 21, 2019.
- [ZGKPP17a] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. *A Zero-Knowledge Version of vSQL*. Cryptology ePrint Archive, Report 2017/1146. 2017.
- [ZGKPP17b] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. “vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases”. In: *Proceedings of the 38th IEEE Symposium on Security and Privacy*. S&P ’17. 2017, pp. 863–880.
- [ZGKPP18] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. “vRAM: Faster Verifiable RAM with Program-Independent Preprocessing”. In: *Proceedings of the 39th IEEE Symposium on Security and Privacy*. S&P ’18. 2018, pp. 908–925.
- [Zha+21] J. Zhang et al. “Doubly Efficient Interactive Proofs for General Arithmetic Circuits with Linear Prover Time”. In: *Proceedings of the 28th ACM Conference on Computer and Communications Security*. CCS ’21. 2021, pp. 159–177.
- [Zig] *Zigbee Gateway*. <https://www.zigbee.org/zigbee-for-developers/zigbee-gateway/>. Feb. 13, 2019.
- [ZKCAJD15] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, and P. Dutta. “The Internet of Things Has a Gateway Problem”. In: *HotMobile*. 2015.
- [ZYG11] S. Zarandioon, D. D. Yao, and V. Ganapathy. “K2C: Cryptographic cloud storage with lazy revocation and anonymous access”. In: *International Conference on Security and Privacy in Communication Systems*. 2011.
- [ZZZR05] L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron. “Cashmere: Resilient Anonymous Routing”. In: *NSDI*. 2005.