Building Cryptographic Systems from Distributed Trust

Weikeng Chen

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-57
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-57.html

May 10, 2022

Building Cryptographic Systems from Distributed Trust

by

Weikeng Chen

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:

Professor Raluca Ada Popa, Chair
Professor Kenneth Bamberger
Professor Alessandro Chiesa

Spring 2022

Abstract

Building Cryptographic Systems from Distributed Trust

by

Weikeng Chen

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Raluca Ada Popa, Chair

Centralized systems are prone to data breaches, which may come from hackers and malicious or compromised employees inside the company. The scale and prevalence of such data breaches raise concerns from users, companies, and government agencies. In this dissertation, we study how to build systems that distribute centralized trust among many parties, such that the system remains secure as long as at least one of the parties is not compromised. We show how distributed trust can be used to secure systems for storage, machine learning, consensus, and authentication.

Contents

Acknowledgments

I would like to thank many people for their help and support in my PhD journey.

Thanks to my advisor, Raluca Ada Popa, for teaching me virtually everything about computer security and applied cryptography. She taught me to stay persistent and provided a lot of encouragement and support to my research and graduate study in general.

Thanks to my other co-authors, Ian Chang, Alessandro Chiesa, Emma Dauterman, Ryan Deng, Jorge Guajardo, Thang Hoang, Murat Kantarcioglu, Aurojit Panda, Ion Stoica, Katerina Sotiraki, Nicholas P. Ward, Attila A. Yavuz, and Wenting Zheng (alphabetically ordered).

Thanks to my friends and colleagues in RISELab, the Berkeley Security Group, and the Blockchain at Berkeley club.

Thanks to my friends and collaborators in the open-source community of zero-knowledge proofs and secure computation, including arkworks-rs, ZK Garage, and emp-toolkit.

Thanks to my family for their continuous support during my graduate study.

The research in this dissertation is supported by the NSF CISE Expeditions Award CCF-1730628, Jim Gray Fellowship, J. K. Zee Fellowship, as well as gifts from the Sloan Foundation, Bakar, Okawa, and Hellman Fellows Fund, Alibaba, Amazon Web Services, Ant Financial, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

Chapter 1

Introduction

Over the recent decades we see the rise of internet systems. Today, we share files over public cloud storage; we communicate with our friends and colleagues through online messaging apps; we share our email address, home address, and phone number to many websites that we visit. These systems are often ***centralized***, meaning that for each system, there is a centralized party who can see all our data and can use the data in any way that the party wants.

A problem with such centralization is that it is prone to data breaches, which may come from either hackers or malicious/compromised employees inside the company [1–13]. In the last decade, there are about 9500 data breaches in the US, leaking more than one billion records in total [14]. The scale and prevalence of such data breaches raise the concerns from users, companies, and government agencies.

- **Users** worry that the companies who have their data are incapable of protecting such data, as there have been many data breaches.
- **Companies** worry about their reputation and liability when data breaches happen, as shown by several companies that go bankruptcy after just one major data breach [15, 16]. Recently, companies find it difficult to prevent phishing attacks against their employees [17–20].
- **Government agencies** are concerned about the privacy of the data of the citizens and enterprises. They are concerned about the cyber espionage that targets intellectual property from companies and cyberattacks that target federal agencies and other government institutions [21, 22].

There has been many efforts, in different aspects, trying to minimize the risks of these centralized systems, including legislation including privacy laws [23], data privacy standards [24–27], and ongoing efforts in standardizing cryptographic standards [28, 29]. The congress has been discussing billion-dollar cybersecurity investment bill that tries to protect sensitive data and public critical infrastructure [30, 31].

1.1 Summary of the approach

In my research, I build cryptographically secure systems following these four steps.

Identify a security or efficiency problem in important applications. Deployment of secure systems often involves trade-offs between security and efficiency; however, both are necessary because a secure but inefficient system cannot be used in production, and companies may have to continue using existing efficient, yet insecure, systems. The first step in building cryptographic systems is to identify such problem in important applications.

For example, in Cerebro [32] and Holmes [33], we identify the demands of secure collaborative learning among two use cases: (1) banks want to train machine learning models for anti-money laundering and (2) fintech companies want to collaborate with existing credit bureaus to know user credit history. The difficult part is that these companies do not want to share data with each other, and therefore the training or inference must be done using secure cryptographic protocols such as MPC (secure multiparty computation), which leads to a severe efficiency overhead. The battle between security and efficiency here is the motivation of Cerebro [32] and Holmes [33]. Cerebro [32] tackles the lack of security by enforcing release policies and auditing the training process, as well as tackles the lack of efficiency through logical and physical planning in the cryptographic protocols. Holmes [33] tackles the lack of security by performing statistical tests over the input data, as well as tackles the the lack of efficiency through a hybrid use of zero-knowledge proofs and secure multiparty computation.

Another example of my work is incrementally verifiable ledger systems [34], which aims to settle the battle between security and efficiency in the blockchain system. It is a known performance issue that blockchain systems do not scale well because every full node must verify the entire blockchain from the very beginning, which leads to a very high participation cost. We identify this performance problem and provide implementations of recursive proofs, which are now being used in practice. A leading blockchain company, Aleo [35], deployed the verifier circuit in our work for running private computation on the blockchain and solving the re-execution problem.

Design cryptographic protocols that enable better security-efficiency trade-offs. To achieve a better trade-off between security and efficiency, we often need to design new algorithms or protocols, and this is a consistent theme of the work in this dissertation.

In Metal [36], we build a cloud storage system that hides metadata. Prior work only offers part of the desired security guarantees, and these constructions are already inefficient. To reconcile the strong security guarantees and the high performance demands, we design new protocols that allow file accesses without metadata leakage and with concrete efficiency.

Another example is in N-for-1 Auth [37], an authentication protocol for decentralized systems. The core of this system is to run a TLS client inside secure multiparty computation. Although one can follow the TLS protocol and perform each operation step-by-step in the secure computation, we find the performance impractical. To address this issue, we examine the TLS protocol and try to remove bottlenecks. Eventually, we identify several steps of the TLS protocol that, with some care, can be run outside the secure computation, and this new protocol enables a better trade-off between security and efficiency.

Design and build the system around it. Designing a cryptographically secure system not only requires cryptography, but also requires system designs because the system needs to work in the real world. This means that the system needs to be compatible with existing ecosystem, to be user-friendly, and to satisfy many other requirements in practice. We now give two examples.

In Metal [36], we design a secure and efficient protocol for users to access a file in remote storage, but there is another challenge: how do users *share* files with others without leaking metadata? Prior work often assumes that users can *"magically"* figure out a way to share confidential information about how to access a file, thereby enabling file sharing. But we know that it is difficult for users to figure out such a way without sacrificing the anonymity. Eventually, Metal [36] includes a dedicated module that provides an efficient and secure way to share files.

In N-for-1 Auth [37], we design an efficient protocol for running a TLS client inside secure computation, and we start to use this protocol to perform decentralized authentication for a number of authentication factors (e.g., email, SMS), and we find that *usability* is a challenge. For example, for security, users need to provide a different passcode to each authentication server. The process of having users to go through different websites is cumbersome. To address this challenge, N-for-1 Auth [37] provides a client software to allow users to use the system without worrying about the cryptographic detail.

Release open source artifacts. To bring cryptography to the real world, we need to first implement it. We take efforts to provide open-source implementation for the various cryptographic tools used in this dissertation. Here, we focus on two examples, and other work also has shown adoption.

For example, in Cerebro [32], we build an artifact of our system, which is submitted to USENIX Security 2021 for artifact evaluation, and the paper received the evaluation badge. Our artifact includes not only the implementations of new protocols that support physical planning and examples for implementing machine learning in secure computation, but also documentation and tools that enable one to evaluate the results provided in the paper. One of the reviewers of the artifact evaluation said that "the documentation was fantastically well written and easy to follow."

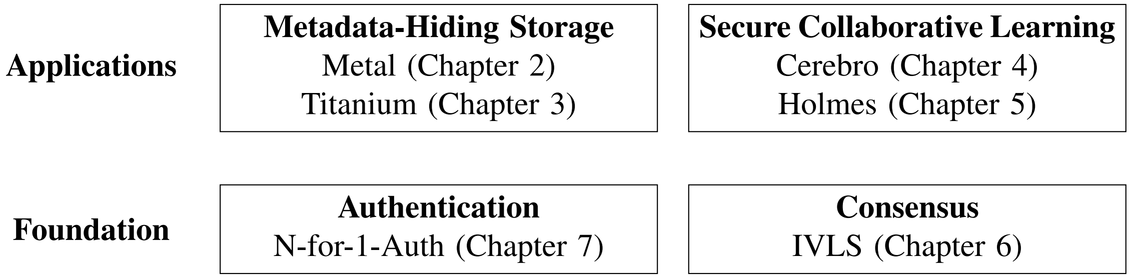
Another example is in incrementally verifiable ledger systems [34], which improves the state-of-the-art succinct zero-knowledge proofs. Many of the tools are implemented for the very first time. Today, they have been used in a number of blockchain companies, such as Aleo [35] as in their implementation of private smart contracts.

1.2 Overview of the works

As shown in Figure 1.1, the systems presented in this dissertation can be organized into two categories: applications and foundation. We choose to work on these areas because they are important today and have wide adoption. We now briefly describe each of these systems.

Metadata-hiding storage: Metal and Titanium. Metal [36], presented in Chapter 2, is a storage system that hides the metadata. Essentially, the servers that provide the storage service for the users know almost nothing about files that the users store—the servers do not see the file contents, do not know which user is using the system, and do not know which file is currently being accessed. This is a joint work with Raluca Ada Popa.

Titanium [38], presented in Chapter 3, is an enhanced version of Metal with stronger security guarantees. Compared with Metal which is secure against two semi-honest servers, Titanium,



Applications

Metadata-Hiding Storage
Metal (Chapter 2)
Titanium (Chapter 3)

Secure Collaborative Learning
Cerebro (Chapter 4)
Holmes (Chapter 5)

Foundation

Authentication
N-for-1-Auth (Chapter 7)

Consensus
IVLS (Chapter 6)

Figure 1.1: Summary of the works.

however, is secure against $n - 1$ out of n servers being maliciously compromised. This is a joint work with Thang Hoang, Jorge Guajardo, and Attila A. Yavuz.

Secure collaborative learning: Cerebro and Holmes. Cerebro [32], presented in Chapter 4, is a platform for collaborative learning. The platform aims to provide an end-to-end solution for collaborative learning. Beyond the basic support of machine learning, the platform can enforce release policies and auditing for detecting malicious inputs. This is a joint work with Wenting Zheng, Ryan Deng, Raluca Ada Popa, Ion Stoica, and Aurojit Panda.

Holmes [33], presented in Chapter 5, is a protocol for performing distribution tests in secure collaborative learning. Distribution testing enables organizations participating in collaborative learning to check if data contributed by each other is biased (or unbalanced). Although existing MPC protocols are already able to do so, the costs are extremely high. Holmes uses a hybrid construction that integrates MPC and zero-knowledge proofs, and Holmes is able to achieve significant speedups. This is a joint work with Ian Chang, Katerina Sotiraki, Raluca Ada Popa, and Murat Kantarcioglu.

Distributed authentication: N-for-1 Auth. N-for-1 Auth [37], presented in Chapter 7, provides decentralized authentication for a number of commonly used authentication factors and makes them friendly to the general users. This is a joint work with Ryan Deng and Raluca Popa.

Scalable blockchain systems: IVLS. Incrementally verifiable ledger systems (IVLS) [34], presented in Chapter 6, studies how to reduce the participation costs for blockchain systems and provides a solution through recursive composition of zero-knowledge proofs. This is joint work with Alessandro Chiesa, Emma Dauterman, and Nicholas Ward.

Chapter 2

Metal: A Metadata-Hiding File-Sharing System

File-sharing systems like Dropbox offer insufficient privacy because a compromised server can see the file contents in the clear. Although encryption can hide such contents from the servers, metadata leakage remains significant. The goal of our work is to develop a file-sharing system that hides metadata—including user identities and file access patterns.

Metal is the first file-sharing system that hides such metadata from malicious users and that has a latency of only a few seconds. The core of Metal consists of *a new two-server multi-user oblivious RAM (ORAM)* scheme, which is secure against malicious users, a *metadata-hiding access control* protocol, and a *capability sharing* protocol.

Compared with the state-of-the-art malicious-user file-sharing scheme PIR-MCORAM (Maffei et al.'17), which does not hide user identities, Metal hides the user identities and is $500\times$ faster (in terms of amortized latency) or $10^5\times$ faster (in terms of worst-case latency).

2.1 Introduction

Storing files on a cloud server and sharing these files with other users (e.g., as in Dropbox) are common activities today. To hide the confidential contents of files from a compromised server, academia and industry developed end-to-end encryption (E2EE) systems [39–45]; using these, the user encrypts the file contents, so a compromised server only sees the encryption of a file, and only permitted users can decrypt the file. Unfortunately, this approach leaves unprotected a lot of user and file metadata. Figure 2.1 summarizes metadata leakage in E2EE systems: notably, the user identities and file access patterns.

Such metadata is sensitive, which has become notorious in a related area—communication surveillance. Former NSA General Counsel, Stewart Baker, said *"Metadata absolutely tells you everything about somebody's life. If you have enough metadata, you don't really need content."* [46] Former NSA Director, Michael Hayden, stated: *"We kill people based on metadata."* [47] Since knowing whom a user calls is similar in spirit to knowing with whom a user shares a file,

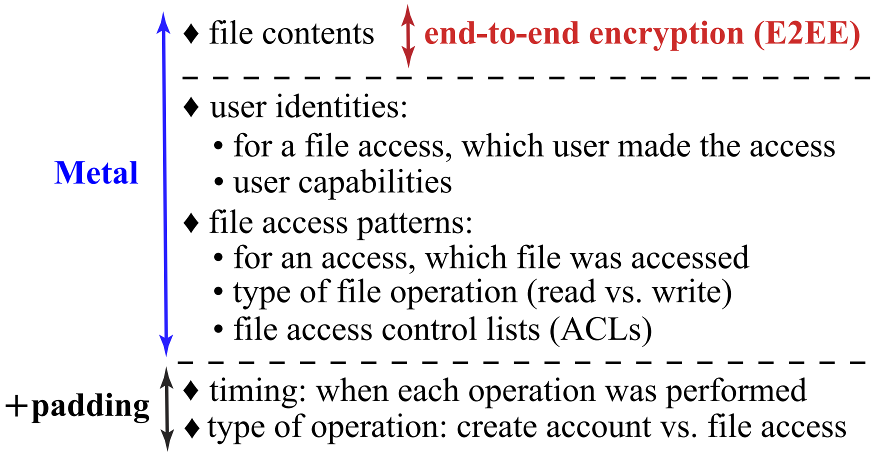

♦ file contents
end-to-end encryption (E2EE)
♦ user identities:
• for a file access, which user made the access
• user capabilities
Metal
♦ file access patterns:
• for an access, which file was accessed
• type of file operation (read vs. write)
• file access control lists (ACLs)
+padding
♦ timing: when each operation was performed
♦ type of operation: create account vs. file access

Figure 2.1: Scope of data/metadata protected by end-to-end encryption (E2EE) systems and Metal; padding in time and computation hides more metadata.

leaking metadata in file sharing is also worrisome. To illustrate this issue, consider some privacy concerns that arise when medical data is stored on the cloud:

The sensitive user identities. Consider that patient Alice and her oncologist Bob share Alice's medical profile in an E2EE system. Even with encryption, the server knows that Alice and Bob share some files with each other. With the side information that Bob is an oncologist, likely available from a Google search for Bob's name, the server knows that Alice is seeing an oncologist and thus infers that she suffers from cancer.

The sensitive file access patterns. Even without user identities, file access patterns are sensitive. Consider that some doctors share a folder with disease handouts that consists of many files, one for each disease. The server sees the access frequency of each file and can relate it to disease incidence rates, which can be found online [48]. Thus, the server can infer the disease in each file. If the server knows the time when Alice goes to the doctor, the server can infer Alice's disease by seeing which file is accessed by the doctor.

Further, there are many general attacks against anonymous systems leveraging social data [49–56] or access patterns [56–63], some of which might apply to file sharing.

The first attempt to hide such metadata is oblivious RAM (ORAM) [68–70]. However, ORAM relies on the trustworthiness of a *single* client or a proxy to maintain the confidentiality of the entire data storage. A recent line of *multi*-user ORAM schemes [64–67], shown in Table 2.1, is more relevant to our setting. These schemes attempt to retain some oblivious guarantees even when some users are compromised; for example, file accesses of an honest user remain hidden across all the files accessible only by honest users. Unfortunately, there are very few such works; the schemes that support file sharing, PIR-MCORAM [67] and GORAM [66], leak either user identities or file access patterns, as depicted in Table 2.1.

This paper presents Metal, the first cryptographic file-sharing system that hides both user iden-

Work	File sharing	Hide access patterns	Hide users	Server complexity	Setup
Secret-write PANDA [64]	No	No	No	Nearly polylog	1-server
AnonRAM-lin [65]	No	Yes	Yes	Linear	1-server
AnonRAM-poly [65]	No	Yes	Yes	Polylog (linear worst-case)	2-server
GORAM [66]	Yes	No	Partial	Polylog	1-server
PIR-MCORAM [67]	Yes	Partial	No	Linear	1-server
Metal (this paper)	Yes	Yes	Yes	Polylog	2-server

Table 2.1: Comparison of multi-user ORAM schemes when ***there are an unbounded number of malicious users***. All the listed schemes assume the server(s) to be semi-honest. The server computation complexity here is in respect to the number of files, assuming each file is of a constant size. The comparison will be discussed in more detail in Section 2.10.

tities and file access patterns both from the server and from malicious users. Figure 2.1 lists the various types of metadata that Metal protects. The scheme with the closest security guarantees, PIR-MCORAM [67], has a very high overhead. Although Metal is not a lightweight system either, it makes a big leap toward reaching practicality—Metal's access latency is $\geq 500\times$ (for amortized latency) or $\geq 10^5\times$ (for worst-case latency) shorter than that of PIR-MCORAM, and in absolute value, only a few seconds.

PIR-MCORAM's very high overhead is largely due to an unfortunate lower bound that challenges this research area: Maffei et al. [67] showed that, to hide access patterns, a single-server file-sharing system must basically scan all the files; hence, PIR-MCORAM scans every file in the system for each file access. To avoid this impossibility result, AnonRAM-poly [65] adopts a *two-server model*, where at least one server is honest. This model is also adopted by much prior work in related settings for similar reasons [71–74]. Metal adopts this two-server model as well.

Unfortunately, even in the two-server model, efficiency remains a troubling challenge. Putting aside the fact that worst-case accesses in AnonRAM-poly are still linear, a significant inefficiency in AnonRAM-poly is that each user's access requires the user to generate a heavy zero-knowledge proof (to prove to the servers that this user did not maliciously deviate from the protocol). Generating such a proof is already $20\times$ times slower than the overall access time of Metal (as described in Section 2.7.6). Further, AnonRAM-poly does not support file sharing; extending AnonRAM-poly to file sharing is challenging because its design makes it difficult to hide the access patterns across files with different sharing permissions. Finally, given the complexity, the authors of AnonRAM-poly have not implemented AnonRAM-poly.

With Metal, we propose a radically different design than AnonRAM-poly, which centers around file sharing and obviates the need for zero-knowledge proofs while resisting malicious users. In Section 2.7, we evaluate Metal extensively and show that its access time is within a few seconds for a file store of 2^{20} 64 KB files. We now overview Metal's techniques.

2.1.1 Overview of Metal's techniques

As a file-sharing system, Metal provides users with the ability to access files and to share permissions to files. When a user makes a request to Metal's servers, Metal checks if the user has the required permission, then the user can fetch or share a file. To understand how Metal performs these operations securely, we now overview Metal's techniques, organized by the challenges they address.

Challenge: Single-user nature of ORAM. ORAM [68–70, 75] can hide access patterns, but it supports only a single client. To share an ORAM with many users, prior work proposes trusting a proxy or trusting all users [76–80], which does not guarantee security in the presence of malicious users.

Primitive Metal: Inspired by ORAM, we start with a primitive construction of Metal (Section 2.5.2), which we describe as follows. In Primitive Metal, the two servers interact and run secure two-party computation (*S2PC*) [81–83], as we illustrate in Figure 2.2. The reader should intuitively think that what happens inside S2PC is "**safe**" (albeit expensive as we will see), and what happens outside is "unsafe". Now, the servers can run a *global single-user ORAM client* inside their S2PC, which ensures that neither server sees the state of this global ORAM client, as well as other components for access control and capability sharing. To store and share files, the users communicate with the global ORAM client in the S2PC, which accesses files stored in the servers' ORAM storage on a user's behalf.

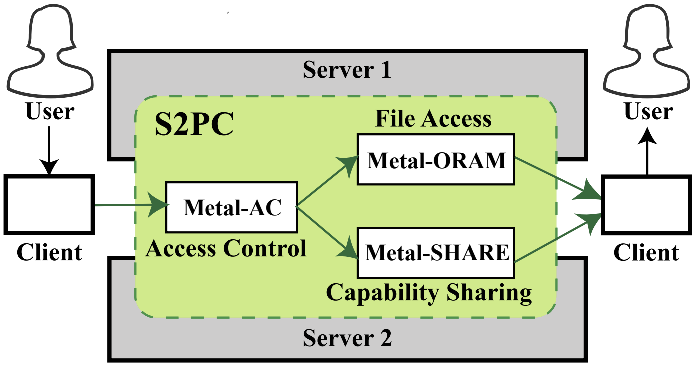


Figure 2.2: Metal's system architecture (described in Section 2.2.1).

This primitive scheme enables users to share files. For the servers to implement a service in this way, the S2PC protocol needs to be *reactive* [84–86]: the servers *repeatedly* take input into the S2PC, update some internal state, and provide output.

Challenge: Inefficiency of Primitive Metal. Though Primitive Metal has the desired security guarantees, it is highly inefficient. Our evaluation in Section 2.7.7 shows that the client ORAM access in S2PC requires a huge amount of server-server communication: $\geq$ 1 GB for each file access. It also requires $\geq$ 75 s to access one file in a store of 2^{20} 64 KB files.

Metal-ORAM: In Primitive Metal, the communication is high because the trusted global ORAM client, which runs inside S2PC, takes the file contents as input. We address this problem using our technique called ***synchronized inside-outside ORAM trees*** (Section 2.5.3), as follows. Metal maintains two ORAM trees on the servers: one containing the file contents called DataORAM and another one containing files' indices and locations called IndexORAM. DataORAM stays *outside* S2PC because it is large, and IndexORAM stays *inside* S2PC because it is small. Metal maintains the two ORAMs *synchronized*: after locating a file's identifier in the IndexORAM, one can find the file contents at the *same* location in DataORAM.

To keep the two ORAMs synchronized, the S2PC must apply the maintenance operations of an ORAM to IndexORAM and DataORAM *in the same way*; these operations include path selection and stash eviction. However, it is unclear how to capture these operations inside S2PC and how to apply them securely to DataORAM, which is outside S2PC.

For this problem, we develop our ***tracking and permutation generation technique*** (Section 2.5.6), which works as follows. During the ORAM access in IndexORAM, our circuit inside S2PC *tracks* the transformations applied to IndexORAM and converts them into a permutation. It turns out that the transformations are not naturally a permutation, but by "resurrecting" missing blocks in a certain way, Metal succeeds to create a permutation. Then, we use a custom S2PC protocol to apply the permutation securely and efficiently to DataORAM.

Altogether, in Metal, the general S2PC no longer touches the file data but works with the position maps and block locations, which reduces the overhead by $\approx 20\times$. We call this scheme Metal-ORAM, and we expect these techniques to be useful for other secure multi-party computation (SMPC) protocols.

Challenge: Performing oblivious access control in S2PC. A natural solution for file access control is to obliviously verify, inside S2PC, that the user's name appears in the file's access control list. However, since a file could involve thousands of users, checking the access control list in S2PC is expensive.

Metal-AC: Metal designs ***capability-based anonymous access control***, which we call Metal-AC (AC refers to access control); the unit of our access control, the *capability*, is inspired by the classical systems concept of a capability [87]. The key differences in Metal are that, given a capability, the servers cannot tell which file (or user) the capability is for, and that the capability is implemented cryptographically, checked inside S2PC by the two servers. By doing so, Metal-AC avoids the heavy handling of access control lists.

Challenge: Establishing anonymous identities. To preserve user anonymity, users must hide their real-world identities (e.g., email addresses) when sharing files. Simply choosing a pseudonym is insufficient because the servers or the malicious users can link the activities that involve this pseudonym together. Even if a user creates multiple accounts, the sharing of files between these accounts can link them together.

Metal-SHARE: In Metal, users share files via ***anonyms***, each of which is a secret identity exclusively shared between a pair of users. Different from traditional pseudonyms, a user's many anonyms are ***unlinkable*** to one another, so they will not reveal a user's identity even when put together. Metal's anonyms also permit one-sided anonymity; e.g., an anonymous whistleblower can send a file to a specific journalist.

We call this scheme Metal-SHARE. This scheme is efficient: even if a user creates millions of anonyms, the user's effort to receive a new file does not increase with the number of anonyms—Metal's client accumulates all file capabilities shared with a user even when they are under different anonyms.

When designing Metal with these strong privacy guarantees in mind, a number of other challenges popped up. For example, some naive solutions enable the servers to see how many files a user has received. Padding to the maximum number of files for each user is very expensive. Instead, Metal instantiates ***capability broadcast*** (Section 2.6.2) on the servers, which hides the per-user numbers of received files.

We describe Metal's security guarantees (Section 2.2.3) and provide security proof sketches in the paper.

2.2 Overview

We now describe Metal's system architecture, threat model, and security guarantees.

2.2.1 System architecture

Figure 2.2 shows Metal's system architecture, which consists of two servers and many users:

- The two servers run a secure two-party computation (S2PC) procedure (green part in Figure 2.2). This procedure is a *reactive* S2PC protocol: it continuously receives input, updates its internal state, and produces output.
- Each user runs a Metal client on the user's device. The user invokes the client's user-facing API functions (shown in Table 2.2) and receives results from the Metal client.
- The Metal client sends requests to the servers. The servers convert the requests to inputs to the S2PC procedure and run the S2PC. The servers then send the output from the S2PC procedure to the client (on the right of Figure 2.2).

Components. Metal consists of three components: Metal-AC for access control, Metal-ORAM for file access, and Metal-SHARE for capability sharing.

As Figure 2.2 shows, the client's request arrives at the first component, Metal-AC, which checks whether the user has the required permission. If so, the request is dispatched to Metal-ORAM for accessing a file or to Metal-SHARE for sharing.

API functions. The Metal client provides the user with some API functions (shown in Table 2.2). The client translates user API calls to requests to the servers, processes the servers' responses, and returns the results to the user. In addition, the client stores and manages the user's secret keys and capabilities in Metal.

We now provide an example about how two users Alice and Bob use Metal's API to store and share files. First, Alice and Bob each create an account using the CreateAccount function. Alice can then invoke ReadFile or WriteFile to read or write her files. Now, suppose that she wants to share a file with Bob.

To receive the file from Alice, Bob uses NewAnonym to generate a new anonym A_{Bob} and sends it to Alice via some out-of-band communication (as discussed in Section 2.2.2). After Alice receives this anonym, she grants Bob read access to one of her files by calling the SendCapability function, which produces such a capability and sends it to Bob.

Syntax of user-facing API functions	Description
CreateAccount() $\to U, \{F_{U,1}, F_{U,2}, ..., F_{U,\ell_{\mathsf{file}}}\}$	A user creates a new account U and creates ℓ_{file} empty files on the servers (Section 2.4).
ReadFile$(U, F) \to$ `fileContent`	A user with account U reads the file identified by F from the servers (Section 2.5.3).
WriteFile$(U, F,$ `newFileContent`$)$	A user with account U writes to the file identified by F on the servers (Section 2.5.3).
NewAnonym$(U) \to A_{U,i}$	A user with account U generates a new anonym $A_{U,i}$ with anonym index i (Section 2.6.1).
SendCapability$(V, F_V, A_{U,i},$ `permission`$)$	Another user with account V and file F_V sends a capability to access F_V (permission is *read*, *write*, or *read+write*) to the user who owns anonym $A_{U,i}$ (Section 2.6).
ReceiveCapability$(U) \to$ $(F_V,\ A_{U,i},$ `permission`$)$	A user with account U receives a capability to file F_V from another user V, sent through U's anonym $A_{U,i}$ (Section 2.6).

Table 2.2: Metal client's user-facing API functions.

Bob then uses the ReceiveCapability function to receive the capability for this file from the servers, in which Bob knows the file is sent through A_{Bob}. Since Bob can have many anonyms and Bob only gives A_{Bob} to Alice, Bob knows that the file is from Alice, assuming that her client is not compromised.

2.2.2 Threat model

Metal uses the following threat model: the attacker can compromise any set of users in a malicious way and one of the two servers in a semi-honest way, while the other server is not compromised. We assume that each user establishes secure connections with each server (such as TLS).

Metal makes two assumptions on communication:

- ***Anonymity network.*** Achieving anonymity requires users to hide their IP addresses. Metal assumes that each user uses an anonymity tool to contact the servers. Many such tools exist, providing varying degrees of anonymity, such as Tor [88], secure messaging [89–96], and a trusted VPN proxy.
- ***Out-of-band communication.*** Before sharing a file, a user must first know the recipient's identity (an anonym in Metal); otherwise, the user does not even know who should receive the file. Exchanging the anonym requires some out-of-band communication between the two users, which is similar to a Bitcoin user's telling another user its wallet address. The users can meet in person or use secure messaging. Metal strives to minimize the use of such out-of-band communication: every two users only need to use this channel to exchange their Metal anonyms *once*, and the subsequent file-sharing activities will be performed within Metal.

2.2.3 Security guarantees

We now describe Metal's security guarantees informally. We consider a set of malicious users $\mathsf{MalUsers}$ who collude with one another and with one of the servers, and consider an honest user U who can access file F. The malicious users can interact with the honest users, including sharing files with them. For file access, Metal provides the following guarantees:

(a) **Anonymity:** Neither the servers nor anyone in $\mathsf{MalUsers}$ can distinguish the honest user U from other honest users.

(b) **File secrecy and integrity:** If user U has never granted anyone in $\mathsf{MalUsers}$ read or write access to F, $\mathsf{MalUsers}$ learn nothing about F or cannot modify F, respectively.

(c) **Read obliviousness:** Neither $\mathsf{MalUsers}$ nor the servers know which file was read by user U, even if $\mathsf{MalUsers}$ have read/write capability to all files. That is, $\mathsf{MalUsers}$ cannot distinguish a read operation from a completely different read, by another honest user, to another file.

(d) **Write obliviousness:** If U never gave anyone in $\mathsf{MalUsers}$ the read capability to F, neither the servers nor anyone in $\mathsf{MalUsers}$ realizes that file F has changed. If someone in $\mathsf{MalUsers}$ has read capability, they legitimately learn that the file is changed, but they do not learn who changed it if more than one honest user has write permission to F.

(e) **Read/write indistinguishability:** Neither the servers nor anyone in $\mathsf{MalUsers}$ knows whether a honest user's file access request is read or write, if none of $\mathsf{MalUsers}$ has read capability to that accessed file.

For file sharing, consider another user V who wants to share file F with U, and U owns two anonyms $A_{U,i}$ and $A_{U,j}$ where $i \neq j$.

(f) **Capability sharing secrecy:** If user V sends a file F's capability to U via $A_{U,i}$, Metal does not reveal to the servers or other users (besides U and V) the following: U, V, F, $A_{U,i}$, or that U and V have access to F.

(g) **Anonym unlinkability:** Neither servers nor anyone in $\mathsf{MalUsers}$ can link $A_{U,i}$ and $A_{U,j}$ unless U reveals this linkage to compromised users.

(h) **Anonym authenticity:** If user U gives anonym $A_{U,i}$ to someone in $\mathsf{MalUsers}$, users in $\mathsf{MalUsers}$ cannot send files to the other anonym $A_{U,j}$ that $\mathsf{MalUsers}$ do not know.

Formalism and proofs roadmap. Metal achieves the guarantees above based on common cryptographic assumptions. In Section 2.8, we provide a simulation-based security definition and proof for Metal-ORAM. In Section 2.4 and Section 2.6.1, we provide proof sketches for Metal-AC and Metal-SHARE.

Non-guarantees. Metal does not hide when the user calls the API (timing) or which function the user is calling (e.g., sharing a permission vs. reading a file). These two leakages can be hidden by padding in time and in computation, which are easy to add to Metal (at an extra cost), as we discuss in Section 2.9. Metal does not protect against denial-of-service attacks by a server, and Metal does not protect against either server being malicious (beyond semi-honest), which we leave to future work.

2.3 The layout of S2PC in Metal

In this section we present the layout of the secure two-party computation (S2PC) that the two servers run in Metal. Metal's S2PC takes a specific form, within which we will plug in Metal's techniques. To instantiate the secure computation, Metal uses Yao's protocol [81–83, 97–99] in a reactive manner.

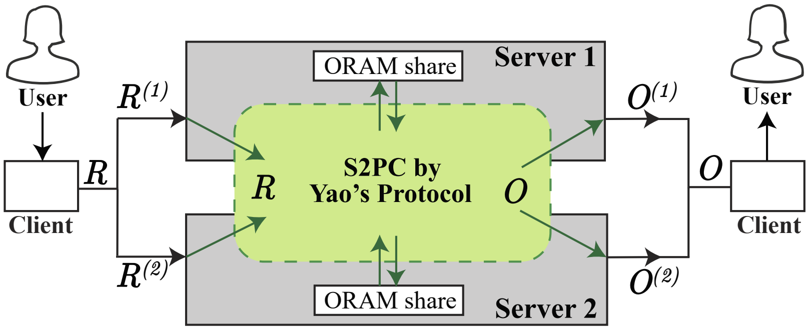


Figure 2.3: Metal's two servers run Yao's protocol to take user input and access ORAM storage.

Client sending a request. As Figure 2.3 illustrates, a client sends its request R (e.g., which file to access) to the two servers *secret-shared* (e.g., using XOR secret-sharing) into $R^{(1)}$ and $R^{(2)}$. In this way, no server sees the request in the clear. Server i will receive $R^{(i)}$. Inside S2PC, the servers combine the two shares $R^{(1)}$ and $R^{(2)}$ to create R.

Metal's servers also secret-share the ORAM store such that neither server knows the data stored in the ORAM. But, if they want to access some parts of the ORAM, they take their local shares of those parts as input and reconstruct those parts inside the S2PC. The two servers then update the ORAM store by outputting the updated shares from the S2PC.

Yao's protocol. Our S2PC is based on Yao's garbled circuits protocol. We present Yao's protocol as a black box here and in the form relevant to our S2PC. Yao's protocol enables two parties (here, the two Metal servers) to jointly compute a function over their own secret inputs without leaking the secret inputs to each other. Concretely, suppose that Server 1 has secret input $x^{(1)}$ and Server 2 has secret input $x^{(2)}$, they can compute a function $f(x^{(1)}, x^{(2)}) \rightarrow (y^{(1)}, y^{(2)})$ such that Server i learns only its own input $x^{(i)}$ and function output $y^{(i)}$, and nothing else about the other party's input or function output. To supply a random tape for the function, each server independently samples a share of the random tape, takes it as input to S2PC, and reconstructs the random tape by XORing the two shares inside the S2PC. By doing so, one of the two servers does not know the random tape.

In Metal, $x^{(i)}$ will consist of $R^{(i)}$, the ORAM share stored by Server i, and some other state. The function f processes the user's request by checking the capability and running ORAM client operations or file-sharing operations. The result of f is y, which consists of the response O to the client (as Figure 2.3 shows), an update to the ORAM, and other changes to the servers' state. The S2PC outputs O to the client in secret shares $O^{(1)}$ and $O^{(2)}$, where each server has one share.

Client receiving a response. The servers send the two shares to the client, who can put them together and obtain output O.

Metal's S2PC uses Yao's protocol in a *stateful and reactive* manner like some works in S2PC [84–86, 100–102]. That is, it does not compute just one function f within S2PC, but instead it runs a sequence of functions $\{f_1, f_2, ...\}$ continuously—this sequence of functions can keep state, take new inputs, reveal some outputs midway, and continue processing in this manner for many steps. This reactive property captures the fact that the servers offer a service, not only a one-time computation. The stateful nature is needed to maintain IndexORAM state.

2.4 Metal-AC: Anonymous access control

Metal's first component, Metal-AC, checks whether the user has permission to complete the request. Since it is the simplest of our three components, we present it first as a warm-up.

One natural design for Metal-AC is to store access control lists (ACLs) on the servers. However, materializing ACLs is expensive—to access ACLs obliviously, each file's ACL must first be padded to the size linear to the number of users, then be accessed by ORAM.

Instead, in Metal, each client on a user's machine stores the user's *capabilities*, which represent a user's permission to read or write a file and are reminiscent of operating systems' capabilities [87]. A user needs to present a capability (in secret shares) to the servers before accessing or sharing a file.

Metal uses authenticated encryption, which provides confidentiality and unforgeability, to implement capabilities. The two servers verify a capability by jointly decrypting the capability inside the S2PC. In Metal, a capability is a ciphertext of the access description under a key that is secret-shared between the two servers. For example, a capability to read and write file F has a description "File ID_F: R+W":

$$C_F^{\mathrm{R+W}} := \mathsf{AuthEnc}(\mathsf{capability\_key},\ \text{“File }\mathsf{ID}_F\text{: R+W”})\ .$$

Each server stores a share of the capability key, which is used for all users' capabilities. The servers grant and verify the capabilities inside the S2PC through secret shares, and thus one server cannot see the capabilities or the capability key.

Granting a capability. The servers, in S2PC, grant a capability to a user in the following two situations:

- When the user **creates an account** (by calling the CreateAccount function), the servers, in S2PC, reserve a continuous range of ℓ_{file} file identifiers for this user, who obtains a *multi-file capability* for reading and writing any of these ℓ_{file} files. Later, the user operates on these ℓ_{file} files. The user can use this capability to share the files.

– When the user **receives a capability of a file that another user shares with this user** (by calling the ReceiveCapability function), the user obtains a capability for this file, which is generated by the servers during the other user's invocation of SendCapability (described in Section 2.6). The user can use this capability to access the file but not to share the file.

To grant a capability, the S2PC between the servers decides an access description (e.g., file F with permission P_F) and proceeds as follows: the S2PC reconstructs the capability key, computes the capability, and returns the capability in the form of secret shares to the user's client, as described in Figure 2.4.

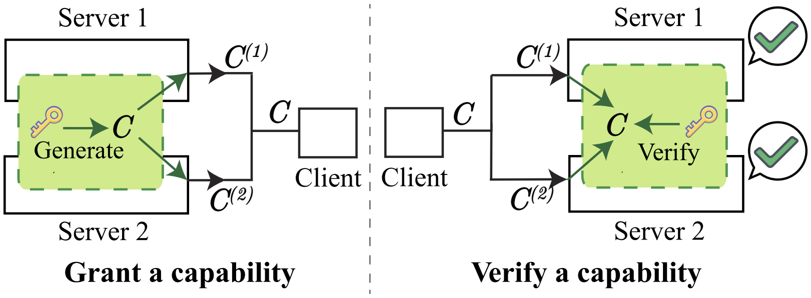


Figure 2.4: Metal-AC grants or verifies a capability C using the capability key, which is secret-shared between the two Metal servers.

Verifying a capability. Since users can be malicious, each user needs to present a capability to the servers before accessing some file. To start with, the user's client splits the capability into two secret shares and provides one to each server. Since each time the client uses fresh randomness for secret sharing, the servers do not know if the same capability is used again. Then, as Figure 2.4 shows, the S2PC uses the capability key to decrypt the capability.

If the access description is valid for the operation the user wants to perform, Metal-AC invokes Metal-ORAM or Metal-SHARE as in Figure 2.2 and provides the access description to the corresponding component inside S2PC.

Security proof sketch. Metal-AC uses authenticated encryption to hide the information inside the capability from the user, which avoids the leakage of the file owner due to file identifiers' being reserved in owner-specific continuous ranges during the account creation, and to prevent a malicious user from forging a capability. Metal-AC uses S2PC to distribute the access to the capability key, so that one server cannot grant a capability or see what is inside the capability. By using secret shares to exchange the capability between the client and the S2PC, anyone of the servers does not even see the capability.

In relation to the security guarantees we described in Section 2.2.3, Metal-AC ensures anonymity since none of the two servers can see what the capability is, and the user does not have the capability key; later in Metal-ORAM (Section 2.5.3), we can see that Metal-AC helps us achieve file secrecy and file integrity by allowing only those users with the valid capability to access that file. Metal-AC does not leak the capability as well as the access description inside the capability, which helps achieve obliviousness and read/write indistinguishability.

2.5 Metal-ORAM: Efficient two-server multi-user ORAM for file storage

In this section we describe how the two Metal servers store and obliviously access user files using Metal-ORAM. We first provide some background about ORAM as well as the construction of Primitive Metal and its limitation. Then, we describe Metal's synchronized ORAM trees as well as the tracking and permutation generation techniques, which overcome this limitation. We prove the security of Metal-ORAM in Section 2.8.

2.5.1 Background on ORAM

Metal-ORAM wants to use ORAM for this scenario: the two servers running a S2PC procedure store an array of files D in the S2PC state and they want to access the x-th file $D[x]$ inside the S2PC, without any server knowing the secret location x. ORAM for S2PC [84, 85, 100–102] is a cryptographic primitive that enables such oblivious data access in S2PC.

We identified Circuit ORAM [100] to be appropriate for our setting: the ORAM client has competitive performance, which is poly-logarithmic to the number of files even in the worst case, while other schemes such as SqrtORAM [101] and Floram [102] have a linear worst-case complexity. Circuit ORAM has the benefit that the user waiting time remains acceptable even in the worst case.

We now provide necessary background about Circuit ORAM for the reader to understand how Metal uses it. Circuit ORAM stores such a file array D in a binary tree. To store N files, Circuit ORAM uses a tree with height $h = \lceil \log_2 N \rceil$, as Figure 2.5 shows. Each tree node can store three fixed-size *blocks*. In addition to tree nodes, a stash temporarily stores some blocks that have not been added to the tree, up to the stash size bound.

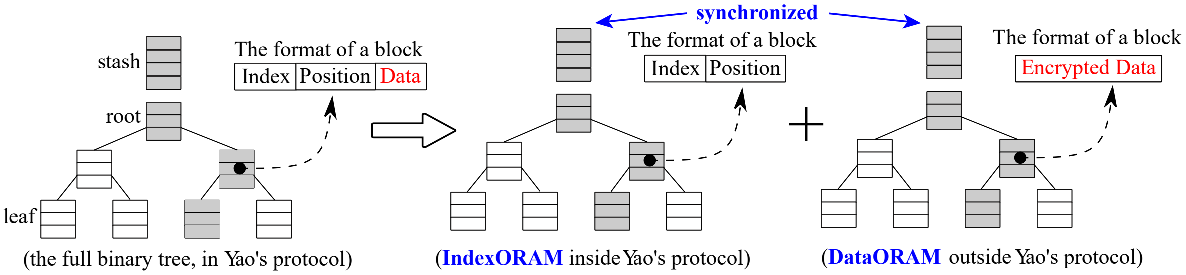


Figure 2.5: Metal-ORAM moves data out of Yao's protocol (Section 2.5.3). The data is too large to be processed efficiently in Yao's protocol.

Each block either is empty or stores the data of a file $D[x]$. Such a block consists of the index x, the data $D[x]$, and its position—the root-to-leaf tree path where this block resides. If a block is currently buffered in the stash, the block stores which tree path the block will be evicted onto.

To locate a block in this tree, Circuit ORAM keeps a position map, which maps each index $x \in \{1, 2, ..., N\}$ to the path on which the block resides if the block is not buffered in the stash. The position of $D[x]$ in the position map should match the position field in the block that contains $D[x]$.

Reading. To read a file, the two servers in the S2PC first look up the file's position in the position map, then look up the block by a linear search of both the stash and the path corresponding to this position. The two servers can then read the data in the block in S2PC. After reading the file, the two servers assign a new random position to this block, put the block into the stash, and update the position map accordingly.

Writing. To write a file, the two servers follow the similar steps, but when they put the block back into the stash, the two servers replace the data with the new data from the user. Note that accesses to the position map also need to be oblivious. Metal uses the standard recursive technique [70, 75, 103] to store the position map in ORAM.

Stash eviction. After each read and write, Circuit ORAM needs to perform a stash eviction, in which some blocks buffered in the stash are evicted into the tree to ensure that the stash does not overflow. For each eviction, Circuit ORAM chooses two paths of the tree [104] and rearranges the blocks in the stash and the blocks on these two paths. This rearrangement is the heaviest step and involves a lot of technical details less relevant to describe here but included in [100].

2.5.2 Primitive Metal

We now have enough background to describe Metal's primitive scheme, which serves as the foundation for our subsequent improvements. Recall that Metal's primitive scheme already provides the desired security guarantees, though it is slow. First, Circuit ORAM immediately gives us a way to achieve *read/write obliviousness* and *read/write indistinguishability*. Recall that the two servers now can run a function f that requests file data $D[x]$ as input from ORAM; none of the servers knows the index x or the data $D[x]$. The two servers also do not know whether the function f reads or writes the data because we pad the computation in function f; this padding overhead is small because reading and writing are similar in ORAM. Second, we obtain *anonymity* and *file secrecy/integrity* with the help of Metal-AC (Section 2.4).

We now outline this primitive scheme. In this scheme, all users' files are arranged in a file array D stored in the ORAM inside the S2PC and are padded to have the same size (e.g., 64 KB). A user who wants to read or write a file $D[x]$ first presents a capability to pass the check in Metal-AC. If the user passes the capability check, the two servers access the ORAM on the user's behalf. The two servers return the file data back to the user via secret shares, as described in Section 2.3. Note that if the user writes to a file rather than reads, the user receives dummy data in the response, as a result of padding.

Yet, Primitive Metal is slow: our experiments (Section 2.7.7) show that it takes ≥ 75 s to read a file in a store of 2^{20} 64 KB files. In addition, storing data in S2PC is quite expensive because every bit in S2PC is represented as a pair of garbled circuit labels, resulting in a storage overhead of at least $256\times$.

The bottleneck of Primitive Metal. The primitive scheme is slow due to the data-intensive operations inside Yao's protocol. Recall that Yao's protocol builds on garbled circuits; processing a large amount of data leads to many garbled gates being generated, transmitted, and evaluated, resulting in heavy computation and communication. In particular, the primitive scheme is (1) reading all the blocks in the stash and on an ORAM path and (2) rearranging all the blocks in the stash and on two ORAM paths during stash eviction.

Metal-ORAM avoids this bottleneck by moving all the file data out of Yao's protocol and processing such data with more efficient, customized protocols.

2.5.3 Moving data out of Yao's protocol: Metal's synchronized inside-outside ORAM trees

To avoid the primitive's limitation, Metal-ORAM splits the ORAM binary tree into two synchronized ORAM stores, IndexORAM and DataORAM, as illustrated in Figure 2.5. IndexORAM only contains small metadata with no file data, which is the only data structure that will be accessed ***inside*** Yao's protocol. DataORAM stores the file data ***outside*** Yao's protocol, not accessed by Yao's protocol.

IndexORAM. We use the recursive technique [70, 75, 103] to store the position map inside recursively larger ORAM trees; hence, IndexORAM is a *set of trees* of increasing sizes. This set of trees enables looking up the position in the last tree for a given file x. However, to simplify matters for clarity, we illustrate only the last tree in Figure 2.5 and we will refer to a single IndexORAM tree in the rest of the protocol description, with the understanding that Metal-ORAM is handling the logistics of the other smaller trees as well.

DataORAM. DataORAM—as Figure 2.5 shows—resembles IndexORAM's last tree but only stores file data. DataORAM stores the data in the form of ElGamal ciphertexts [105–108] under a *global* public key; each server has a share of the corresponding private key. Using the properties of ElGamal, the two servers can rerandomize the ciphertexts without knowing the private keys and can work together to decrypt ciphertexts as needed, which we will leverage in the construction of our protocols. In Metal, the DataORAM tree is stored on Server 1's disk.

Synchronization. Though we split the tree into two structures, we ensure that these two trees are ***synchronized*** in that the data of a file is at the *same* location in DataORAM as its index/position is in IndexORAM.

We now describe how to read and write a file with these synchronized inside-outside ORAM trees.

Reading. To read file $D[x]$, the two servers first find the file index in IndexORAM and retrieve the position p of the file using Primitive Metal's approach. After doing so, the S2PC procedure determines which block on the path stores the file, i.e., the i-th block on the path p is the block for $D[x]$. Due to the synchrony between IndexORAM and DataORAM, as Figure 2.5 shows, the encrypted data of $D[x]$ can be found also in the i-th block of the same path p in DataORAM.

However, we cannot simply have the two servers fetch the i-th block in DataORAM: while the servers can see the path p due to ORAM's guarantees, they should not see i. The location i is related to the block history [109], and revealing i to the servers breaks obliviousness. Therefore, Metal-ORAM combines threshold decryption and our *secret-shared doubly oblivious transfer* protocol (Section 2.5.4) in a way that the user receives the decryption of the i-th block on path p in DataORAM, i.e., the file data $D[x]$, but neither server learns i or $D[x]$.

After reading a file, the two servers need to perform the ORAM management routines: they put the index block into the stash of IndexORAM and put the data block into the stash of DataORAM. These blocks will later be evicted into the tree. We will describe the details of the reading protocol in Section 2.5.4.

Writing. To write a file $D[x]$, the two servers run the protocol in a similar manner, but we want to ensure that (1) the user with write permission does not see the file contents (since such a user might not have the read permission) and (2) the user-provided data is inserted into DataORAM.

Thus, the writing protocol makes the following changes: First, instead of reading the i-th block in the array, the protocol reads a dummy block, which contains empty file data; therefore, a user with only write capability does not see any file data in this operation. Second, when the two servers insert a data block back into the DataORAM's stash, the two servers instead write the user-provided block into the stash. The user-provided block is created in the following manner: the user secret-shares the file contents between the servers, each server encrypts one share, and the servers combine the two encrypted shares.

To make reading and writing indistinguishable, we merge their protocols as one protocol such that the servers are running the same protocol for reading or writing, with little overhead, as we will show in Section 2.5.4 and Section 2.5.5. This merged protocol does not reveal whether it is reading or writing to one of the servers, and the protocol still preserves file secrecy and integrity: a user with read capability cannot modify the file, and a user with write capability does not see the file data.

Stash eviction. The last aspect we need to take care of is stash eviction, which is needed after every read or write. The stash eviction rearranges some blocks in the tree. The challenge is that if Metal-ORAM only evicts the stash in IndexORAM, the synchrony between IndexORAM and DataORAM breaks.

Metal-ORAM remedies the synchrony by "somehow" capturing the rearrangement that happens to IndexORAM and also applying it to DataORAM. We cannot simply reveal the rearrangement to the servers since doing so breaks the ORAM obliviousness. Instead, Metal-ORAM provides a technique for *secure tracking and permutation generation*, described in Section 2.5.6, to convert Circuit ORAM's rearrangement into a permutation. Then, Metal-ORAM employs a distributed permutation protocol to apply the rearrangement to DataORAM, such that the two ORAM trees are re-synchronized.

2.5.4 Fetching blocks in DataORAM

We now describe how to fetch the data block in DataORAM without revealing the location i to the two servers. Circuit ORAM allows the two servers to learn which path the block is assigned to, so the two servers' task is to fetch the data block from among the $|\mathsf{stash}|$ blocks in the stash and the $(3 \times h)$ blocks on the path. Let $\vec{m}$ be the array of $N = (|\mathsf{stash}| + 3 \times h)$ blocks that these blocks form. The S2PC knows the location i; it secret-shares i between the two servers, such that Server 1 knows $i^{(1)}$ and Server 2 knows $i^{(2)}$. Below, we describe our *secret-shared doubly oblivious transfer* (SS-DOT) protocol, at the end of which Server 2 receives the i-th (encrypted) block in the array, without any server learning what i is. The fetched block is encrypted under ElGamal, and the decryption key is secret-shared between the two servers, so the two servers can run an existing threshold decryption protocol [65] and return the file contents to the user in a secret-shared form.

We then add read/write indistinguishability to this fetching operation. Recall that if a user only has write capability, the user should not see the file's data. To ensure such file secrecy as well as to make read/write indistinguishable, the two servers add a dummy block that does not contain any file data at the end of the array. The two servers now search from an array of $(|\mathsf{stash}| + 3 \times h + 1)$ blocks. If the user writes to a file $D[x]$, the S2PC secret-shares $i = (|\mathsf{stash}| + 3 \times h + 1)$ instead, such that the two servers fetch the dummy block, and the user sees only dummy data. This dummy block is unused when the user is reading a file; it merely stays in the array for padding.

Secret-shared doubly oblivious transfer. To fetch the i-th block, Metal uses the following customized protocol. Recall that each server has a share of i: $i^{(1)}$ and $i^{(2)}$, respectively. Server 1 has an array of file data blocks $\vec{m} = \{m_1, m_2, \ldots, m_N\}$. In our protocol, $N = |\mathsf{stash}| + 3 \times h + 1$, and Server 1 needs to rerandomize the blocks read from DataORAM, using the functionality of ElGamal encryption, before executing the SS-DOT protocol.[1] This protocol has Server 2 obtain the i-th block without either server learning i.

Oblivious transfer (OT) [110, 111] does not suffice for our task because in OT one server knows the index. Doubly oblivious transfer [112] does not suffice either because it does not support two-party secret-sharing and focuses on one-out-of-two transfer instead of one-out-of-N.

There are many ways to implement this simple functionality in S2PC, so we do not claim much novelty for this procedure. Yet what is important for us is to find a way that is efficient for our setting because this operation runs for every file access. We develop a simple and efficient procedure as follows:

1. The two servers $\mathcal{S}_1$ and $\mathcal{S}_2$, inside S2PC, reconstruct i from its shares $i^{(1)}$ and $i^{(2)}$, and generate N keys $\{k_1, \ldots, k_N\}$ such that $\mathcal{S}_1$ receives as output all these keys, and $\mathcal{S}_2$ receives only k_i.
2. For each $j \in \{1, ..., N\}$, $\mathcal{S}_1$ uses k_j to symmetrically encrypt 0 and m_j to obtain ciphertexts z_j and c_j, respectively, with authenticated encryption. $\mathcal{S}_1$ shuffles all the (z_j, c_j) pairs and sends them to $\mathcal{S}_2$.
3. $\mathcal{S}_2$ uses k_i to decrypt the first ciphertext of each pair: only one, say z_j, will decrypt to 0. It then decrypts the corresponding c_j, obtaining m_i. Note that j is independent of i because of Server 1's shuffle.

This procedure has the advantages that the computation in S2PC is independent of the length of m_i and that the messages m_i are symmetrically encrypted, which has small ciphertext expansion and efficient encryption/decryption.

[1] A trick to implement this rerandomization efficiently is to observe that Server 2 only sees one of these N blocks, and thus one can rerandomize these N blocks using the same randomness, which saves a lot of computation.

Security proof sketch. The security of SS-DOT, i.e., obliviousness for both parties, is a direct result of the security of S2PC and authenticated encryption.

2.5.5 Putting a block into DataORAM's stash

The next step is to put a block into the DataORAM's stash, which will be later evicted to the tree (Section 2.5.6). Recall that if the user is reading a file, Metal-ORAM should put back the file's current data block, and if the user is writing to a file, Metal-ORAM should insert the user-provided data block. This distinction is crucial for file integrity because we want to avoid a malicious user who only has the read capability to tamper with the file by changing the block.

Metal-ORAM implements this operation by a permutation. Suppose that we place in an array the following: the blocks in the stash, the block read during the fetching (Section 2.5.4), and the user-provided block in an array. The array therefore has $(|\mathsf{stash}| + 2)$ blocks. Suppose that the S2PC finds that the k-th block of the stash is vacant. If the user is reading the file, S2PC can generate a permutation σ_{read} that exchanges the k-th block with the $(|\mathsf{stash}| + 1)$-th block. If the user is writing the file, S2PC generates σ_{write} that exchanges the k-th block with the $(|\mathsf{stash}| + 2)$-th block instead. By doing so, the correct block is inserted into the stash (i.e., the first $|\mathsf{stash}|$ blocks of the permuted array). The servers then discard the last two blocks.

The challenge is to obliviously perform this permutation: neither server should learn k because leaking k breaks the ORAM obliviousness, and neither server should know which permutation, σ_{read} or σ_{write}, is performed because we want read/write indistinguishability.

Metal-ORAM *distributes this permutation* in a way that hides the permutation. Inside the S2PC, Metal-ORAM secret-shares the permutation into two permutations $\sigma^{(1)}$ and $\sigma^{(2)}$ between the two servers where the composition of $\sigma^{(1)}$ and $\sigma^{(2)}$ equals σ_{read} or σ_{write}. The two servers rerandomize the blocks and apply the permutations in turn; the result is the same as when applying σ_{read} or σ_{write} directly. Formally,

1. The two servers $\mathcal{S}_1$ and $\mathcal{S}_2$, inside S2PC, sample a random permutation $\sigma^{(1)}$ and compute $\sigma^{(2)} = \sigma \circ (\sigma^{(1)})^{-1}$ where $\circ$ denotes composition of permutations and $(\sigma)^{-1}$ denotes the inversion such that $(\sigma)^{-1} \circ \sigma$ is the identity permutation.
2. $\mathcal{S}_1$ rerandomizes the ciphertexts of the $(|\mathsf{stash}| + 2)$ blocks above, applies the permutation $\sigma^{(1)}$, and sends the permuted blocks to $\mathcal{S}_2$.
3. $\mathcal{S}_2$ receives the blocks from $\mathcal{S}_1$, rerandomizes the ciphertexts of the blocks, applies the permutation $\sigma^{(2)}$, and sends them back to $\mathcal{S}_1$.
4. $\mathcal{S}_1$ receives the blocks from $\mathcal{S}_2$ and stores the blocks in the corresponding locations in DataORAM.

This method has been used in a similar manner in SqrtORAM [101] to obliviously reorganize the data blocks.

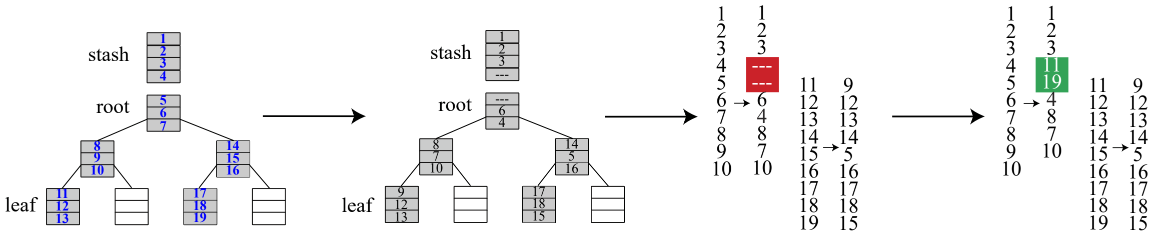
2.5.6 Re-synchronizing after eviction by tracking and permutation generation

After each access to the ORAM, Metal needs to evict the stash. We can run the Circuit ORAM's stash eviction algorithm inside S2PC to update IndexORAM, which rearranges the index blocks, but it breaks the synchrony with DataORAM: a file's index block in IndexORAM is now at a new location, but the data block in DataORAM is still at the previous location.

Metal-ORAM's solution is to extract how blocks in IndexORAM move during the stash eviction and apply the same movement to DataORAM. The challenge is to implement this efficiently. A prior scheme, Onion ORAM [113], uses private information retrieval for this purpose, but it requires a large number of data block operations that is quadratic to the number of blocks being moved, which is heavy.

Metal-ORAM instead develops an algorithm to *track* the changes to IndexORAM inside the S2PC and to *convert them* into a permutation. Then, Metal-ORAM uses the distributed permutation in Section 2.5.5 to rearrange the blocks in DataORAM.[2]

We demonstrate the tracking and permutation generation process in Figure 2.6 and provide the algorithm in Figure 2.7. We now explain the algorithm at a high level.



Add **trackers** to blocks in IndexORAM   Run Circuit ORAM's stash eviction   Extract the tracker changes   Massage to a permutation

Figure 2.6: Metal's tracking and permutation generation (Section 2.5.6).

Tracking. Before the stash eviction, as Figure 2.6 shows, Metal-ORAM attaches some "trackers" to the index blocks inside S2PC. Next, it performs the stash eviction per the ORAM algorithm and then observes how the trackers moved. In Circuit ORAM's stash eviction, some trackers disappear because the blocks that they were attached to were deleted (indicated by '---' in Figure 2.6). Thus, the list of trackers directly pulled from IndexORAM after the eviction is incomplete (i.e., with empty slots), as shown in Figure 2.6's red area. We give the details of tracking in Figure 2.7.

Permutation generation. These missing trackers prevent us from creating a permutation directly. Hence, Metal-ORAM brings back those numbers to the empty slots, as Figure 2.6's green area highlights. The resultant list of trackers becomes a permutation subsuming the changes in IndexORAM. Metal-ORAM feeds this permutation into the distributed permutation described in

[2] The permutation does not explicitly remove the previous version's data block from DataORAM: since the index of the previous version has been deleted in IndexORAM, that data block becomes inaccessible (treated as dummy) in our construction. The previous version's data block may be indirectly discarded later, as a result of a permutation process shown in Section 2.5.5.

Before Circuit ORAM's stash eviction:

- Recall that Circuit ORAM evicts blocks onto two paths. The algorithm appends a number from 1 to $(|\mathsf{stash}|+6\times h-3)$ (called *trackers*) to each block on the two paths in IndexORAM inside S2PC using the following order, as Figure 2.6 shows:
  1. the blocks on the first path, from stash to leaf
  2. the *unnumbered* blocks on the second path, from stash to leaf

After Circuit ORAM's stash eviction:

- The algorithm "peels off" the tracker numbers from the two paths in order and constructs an array. Some numbers will be missing (indicated by "---" in Figure 2.6).
- The algorithm uses linear scanning to find numbers in $\{1, 2, ..., |\mathsf{stash}| + 6 \times h - 3\}$ that have not appeared in the array (e.g., 11 and 19 as in Figure 2.6).
- The algorithm uses linear scanning to find the "---" slots in the array and fills into the slots the unused numbers. (The order is unimportant, but our algorithm starts with the smaller ones.)

Figure 2.7: Algorithms for tracking and permutation generation.

Section 2.5.5 to apply the permutation to DataORAM. Thus, the IndexORAM and DataORAM become re-synchronized, as desired, shown in Figure 2.7.

We have described how stash eviction works in our synchronized inside-outside ORAM trees. In our implementation, Metal-ORAM combines the permutation in Section 2.5.5 with the re-synchronizing permutation inside the S2PC, so every file access only uses one distributed permutation.

2.6 Metal-SHARE: Unlinkable capability sharing

We have achieved oblivious file storage, but we have not yet shown how a user shares files with other users. In this section we describe Metal-SHARE, which contributes the functionality of file sharing without introducing metadata leakage. We first describe two central notions of Metal-SHARE, anonyms and capability broadcast list, and then describe the sharing protocol.

Anonyms. Anonyms are anonymous identities that a user can leverage in file sharing. An anonym is similar to an email address for receiving emails or a Bitcoin wallet address for receiving Bitcoin; but, our usage of anonyms has the additional benefit that it hides the anonym owner's identity such that two anonyms of the same user cannot be linked to each other. Every user in Metal can locally create many anonyms without interactions with the servers. A user then gives his/her anonyms to others in order to receive file capabilities from them. For example, a user U who wants to receive (many) files from user V in the future can provide V with one of the anonyms, $A_{U,i}$ (where i is the anonym index), as Figure 2.8 shows.

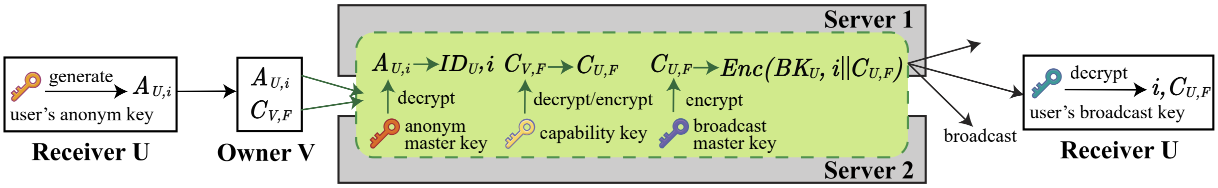


Figure 2.8: Sending and receiving a file's capability in Metal-SHARE.

Capability broadcast. When a user V sends a capability to U, V puts the capability on the servers so that the receiver U can later retrieve it from the servers. This allows U and V to share files even if they will never be online at the same time.

To avoid leaking metadata through network patterns of a user, Metal-SHARE has Server 1 publish a list of encrypted capabilities so that each user can download this list. A user can only decrypt those ciphertexts destined to the user. This is similar to a blockchain where all nodes download the blocks, but only some of the blocks contain transactions relevant to the node. In contrast to a blockchain, though, in Metal-SHARE, the blocks are much smaller than in a regular blockchain, and users only need to download and organize the capabilities periodically.

To implement this broadcast, Server 1 keeps a capability broadcast list that contains the encrypted capabilities for users to download. The two servers will encrypt, inside S2PC, each capability under the recipient's *broadcast key*, which the user receives during the account creation from the two servers' S2PC. When a user's client wants to download the list, Server 1 shuffles the capabilities in the list before sending them to the client.

After user V obtains the capability, V can—in the future—use this capability to access the file, without interacting with U. If U wants to revoke the permission, U must discard the old file, create a new file, and share the new file with other users who are supposed to retain the permission.

To prevent the broadcast list from growing monotonically, Metal-SHARE has each encrypted capability in the list to be deleted after a fixed interval (e.g., three days).

Example. We now illustrate how to share a capability. Suppose that user V owns file F and wants to send the read capability of file F to another user U, as Figure 2.8 shows. The procedure is:

1. **Get the receiver's anonym.** User V obtains one of U's anonyms from U, say $A_{U,i}$, as we discussed in Section 2.2.2. This anonym can be used for all future file-sharing activities from V to U.
2. **Send a capability.** User V who owns file F requests the servers to grant a capability for reading F to the anonym $A_{U,i}$. The servers check V's capability $C_{V,F}$, create a read capability $C_{U,F}$ for user U, and encrypt $C_{U,F}$ together with i under U's broadcast key, as Figure 2.8 shows. The ciphertext of $C_{U,F}$ is appended to the capability broadcast list.
3. **Receive a capability.** User U's client periodically downloads the new capability ciphertexts from Server 1's broadcast list and uses U's broadcast key to decrypt each ciphertext. In this manner, it finds capabilities that are destined to U, one of which will be capability $C_{U,F}$ with the anonym index i and the type of permission. U can learn which anonym was used by the

sender based on i. If $A_{U,i}$ was only provided to V, U knows that this file is from V.

2.6.1 Unlinkable anonyms

We now focus on the left part of Figure 2.8 and discuss how the user U generates a new anonym $A_{U,i}$, how the sender V uses this anonym, and how the S2PC processes the anonym.

Generating the anonym. User U has an *anonym key* AK_U that it received from the servers (via secret shares) during account creation. Using this key, U can generate anonym $A_{U,i}$ for any anonym index i. Informally, the anonym is an encryption of the user ID and the anonym index i.

Sending a capability to this anonym. Another user V who owns file F receives the anonym $A_{U,i}$, as Figure 2.8 shows. User V has the capability $C_{V,F}$ with full permission and wants to grant the read capability to U. To do that, V calls the server API with the anonym and V's capability, asking the servers to create a qualified capability (in this example, read-only) for U (request sent in secret shares as in Section 2.3).

Opening the anonym inside the two servers' S2PC. The two servers secret-share *the anonym master key* (AMK), which can decrypt everyone's anonyms. Thus, the two servers can open the anonym inside S2PC, as Figure 2.8 shows, and continue with the sharing protocol (Section 2.6.2).

Construction. Metal-SHARE implements anonyms using a special-purpose scheme that builds on Paillier encryption [114], additive secret sharing, and message authentication code (MAC). Our construction is in Figure 2.9, and we now describe the intuition behind the construction to show the insights.

First, anonyms need to achieve *anonym authenticity*, as defined in Section 2.2.3. If user U gave anonym $A_{U,i}$ to V, V should not be able to create another anonym $A_{U,i'}$ under a different anonym index $i' \neq i$, assuming that U has never leaked the anonym $A_{U,i'}$ to anyone in the set of colluding malicious users. Our solution is to give user U an anonym key that is derived from the servers' anonym master key, as the Setup and $\mathsf{UserKeyGen}$ algorithms in Figure 2.9 show. U then needs to append a message authentication code over the pair (ID_U, i), as the $\mathsf{AnonymGen}$ algorithm shows, so that another malicious user cannot forge an anonym for user U.

Second, anonyms must provide *anonym unlinkability*. Hence, we cannot expose ID_U, i, or the MAC to another user because such information may deanonymize U. The natural solution is to use public-key encryption to encrypt $(\mathsf{ID}_U, i, \mathsf{mac})$ in such a way that it can only be recovered in the servers' S2PC. For efficiency, we must move the public-key operations out of S2PC: the two servers will do threshold decryption outside S2PC, and inside S2PC they will merge the decryption results efficiently. There are a few public-key encryption schemes that can make this merging step efficient: Paillier encryption [114], Goldwasser-Micali encryption [115], and Brakerski-Gentry-Vaikuntanathan encryption with $\mathbb{Z}_2$ slots [116–119]. We choose Paillier because the size of a ciphertext is small. The $\mathsf{AnonymGen}$ and $\mathsf{AnonymDecrypt}$ algorithms in Figure 2.9 show how Metal-SHARE combines additively homomorphic secret sharing and Paillier encryption to instantiate anonyms.

Note that the sender V also needs to refresh the ciphertext such that the two servers do not realize that the refreshed ciphertext is from the same anonym during the threshold decryption.

Anonym.Setup(1^λ):
Run by the servers during the setup to generate the Paillier keys and the anonym master key.

- For $j \in \{1, 2\}$, $\mathcal{S}_j$ runs: $(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{Paillier.KeyGen}(1^\lambda)$, and publishes pk_j.
- For $j \in \{1, 2\}$, $\mathcal{S}_j$ samples a secret share of anonym master key $\mathsf{AMK}^{(j)} \leftarrow\$ \{0, 1\}^\lambda$ and stores $\mathsf{AMK}^{(j)}$.

Anonym.UserKeyGen$(\mathsf{ID}_U, \mathsf{AMK}^{(1)}, \mathsf{AMK}^{(2)})$
Run by the servers and user U (identified by ID_U) during the account creation to grant the anonym key AK_U to user U.

- $\mathcal{S}_1$ and $\mathcal{S}_2$ run a S2PC that takes $\mathsf{ID}_U, \mathsf{AMK}^{(j)}$ as input ($j \in \{1, 2\}$) and computes:
  - $\mathsf{AMK} := \mathsf{AMK}^{(1)} \oplus \mathsf{AMK}^{(2)}$.
  - $AK_U := \mathsf{PRF}_{\mathsf{AMK}}(\mathsf{ID}_U)$.
- $\mathcal{S}_1$ and $\mathcal{S}_2$ use the protocol in Section 2.3 to share AK_U with the user.
- The user stores the anonym key AK_U.

Anonym.AnonymGen$(\mathsf{ID}_U, i, AK_U, \mathsf{pk}_1, \mathsf{pk}_2)$
Run by the receiver user U to create an anonym with index i, using the anonym key AK_U.

- $s := \mathsf{ID}_U \parallel i \parallel \mathsf{MAC}_{AK_U}(\mathsf{ID}_U \parallel i)$.
- U additively secret-shares s:
  - $s^{(1)} \leftarrow\$ \{0, 1, ..., 2^{|s|+\rho}\}$.
  - $s^{(2)} := s^{(1)} + s$.
- $c^{(j)} \leftarrow \mathsf{Paillier.Enc}_{\mathsf{pk}_j}(s^{(j)})$ for $j \in \{1, 2\}$.
- Outputs the anonym $A_{U,i} := (c^{(1)}, c^{(2)})$.

Anonym.AnonymRerand$(A_{U,i}, \mathsf{pk}_1, \mathsf{pk}_2)$
Run by the file owner and sender V to rerandomize the receiver's anonym $A_{U,i}$ before sending the anonym (in secret shares) to the servers.

- Lets $A_{U,i} = (c^{(1)}, c^{(2)})$.
- V rerandomizes the anonym:
  - $r \leftarrow\$ \{0, 1, ..., 2^{|s|+2\rho}\}$.
  - $c^{(j)}_{\mathsf{new}} \leftarrow \mathsf{Paillier.AddPlain}_{\mathsf{pk}_j}(c^{(j)}, r)$ for $j \in \{1, 2\}$.
- Outputs the anonym $A^{\mathsf{rerand}}_{U,i} := (c^{(1)}_{\mathsf{new}}, c^{(2)}_{\mathsf{new}})$.

Anonym.AnonymDecrypt$(A^{\mathsf{rerand}}_{U,i}, \mathsf{sk}_1, \mathsf{sk}_2)$
Run by the servers upon receiving the (rerandomized) anonym from the file owner V to decrypt the anonym in preparation for the capability broadcast.

- V sends $A^{\mathsf{rerand}}_{U,i} = (c^{(1)}_{\mathsf{new}}, c^{(2)}_{\mathsf{new}})$ to the two servers.
- $\mathcal{S}_j$ runs $s^{(j)}_{\mathsf{new}} := \mathsf{Paillier.Dec}_{\mathsf{sk}_j}(c^{(j)}_{\mathsf{new}})$ ($j \in \{1, 2\}$).
- $\mathcal{S}_1$ and $\mathcal{S}_2$ run a S2PC that takes $(s^{(j)}_{\mathsf{new}}, \mathsf{AMK}^{(j)})$ as input ($j \in \{1, 2\}$), as follows:
  - $\mathsf{AMK} := \mathsf{AMK}^{(1)} \oplus \mathsf{AMK}^{(2)}$.
  - S2PC reconstructs s:
    - $s := s^{(2)}_{\mathsf{new}} - s^{(1)}_{\mathsf{new}}$.
    - Lets s be $\mathsf{ID}_U \parallel i \parallel \mathsf{mac}$.
  - $AK_U := \mathsf{PRF}_{\mathsf{AMK}}(\mathsf{ID}_U)$.
  - If $\mathsf{mac} = \mathsf{MAC}_{AK_U}(\mathsf{ID}_U \parallel i)$, $\mathsf{valid} = 1$. Otherwise, $\mathsf{valid} = 0$.
  - Stores $\mathsf{valid}, \mathsf{ID}_U, i$ in the S2PC's state for the use of capability broadcast (Section 2.6.2).

Figure 2.9: Algorithms of the customized encryption that instantiates anonyms; the use of Paillier encryption follows the common Paillier encryption syntax. Without loss of generality, we assume that the message size of the Paillier encryption set up by security parameter λ is larger than $|s| + 2\rho + 1$ bits where ρ is the statistical security parameter (80 bits in our implementation).

This step avoids the linkage among multiple uses of $A_{U,i}$. The sender V rerandomizes the encrypted secret shares using the additive homomorphism in Paillier encryption, as $\mathsf{AnonymRerand}$ in Figure 2.9 shows.

In this rerandomization step, we adapt a trick from [120] to rerandomize the anonym. As follows, ρ denotes the statistical security parameter. Before the rerandomization, the distribution of each of $s^{(1)}, s^{(2)}$ is statistically indistinguishable from a uniform distribution in $\{0, 1, ..., 2^{|s|+\rho}\}$ as a result of secret sharing. When we rerandomize the two shares by homomorphically adding $r \leftarrow\$ \{0, 1, ..., 2^{|s|+2\rho}\}$, the distribution of each of the new $s^{(1)}, s^{(2)}$ is now statistically indistinguishable from a uniform distribution in $\{0, 1, ..., 2^{|s|+2\rho}\}$ and is *statistically independent* of the original value of $s^{(1)}$ or $s^{(2)}$, which gives us the following guarantee: even if a user calls Send-Capability many times using the same anonym, one of the two servers, knowing the previous rerandomized anonyms, cannot link these anonyms together.

Security proof sketch. Anonym authenticity can be proved by reducing to the unforgeability of MAC. Anonym unlinkability can be reduced to the security properties of Paillier encryption and additive secret sharing. Since our secret sharing has a customized design, we detail its security as follows:

- **Sharing.** Recall from Figure 2.9 that the secret s is shared into $s^{(1)} \leftarrow\$ \{0, 1, ..., 2^{|s|+\rho}\}$ and $s^{(2)} = s^{(1)} + s$ where ρ is the statistical security parameter. The distribution of each share is statistically indistinguishable from a uniform distribution in $\{0, 1, ..., 2^{|s|+\rho}\}$. Since the two shares are then encrypted under separate Paillier keys, if an attacker only has one of the private keys, the attacker sees only one share, not s.
- **Rerandomizing.** As discussed above, the rerandomization algorithm homomorphically adds r to both shares, and the distribution of each new share (inside the Paillier encryption) is statistically indistinguishable from a uniform distribution in $\{0, 1, ..., 2^{|s|+2\rho}\}$ even with the knowledge of the original share. One of the servers does not learn s or the shares in the original anonym.

2.6.2 Capability derivation and broadcast

We now focus on the right part of Figure 2.8. Recall that the two servers secret-share the capability key, as described in Section 2.4. The two servers can decrypt the capability and recreate a capability with qualified permission, such as read-only.

Then, the S2PC encrypts the new capability $C_{U,F}$, along with the anonym index i, using the broadcast master key in such a way that only user U's broadcast key can decrypt it (the user receives this key during the account creation). The ciphertext is revealed to Server 1, who then appends this ciphertext to the broadcast list. User U downloads the new ciphertexts during the past intervals since U was last online. U uses U's broadcast key to try decrypting each ciphertext, among which U can find $C_{U,F}$, the anonym index i, and the type of permission. With this new capability, U can read the file F using Metal-ORAM.

Discussion on hiding the number of incoming files. The broadcast in Metal-SHARE has an overhead linear to the number of file-sharing operations in the whole system, which is not ideal. The benefit of this broadcast is that it hides the number of incoming files and avoids leaking users' use patterns.

One alternative is to use private information retrieval (PIR) like Pung [91, 95]. However, each invocation to Pung can only retrieve a fixed number of data entries. If a user has comparably much more files than other users, this user has to run Pung's protocol multiple times, from which the attacker can still learn this user's use patterns. Another solution is to have users send capabilities to one another via encrypted emails (e.g., PGP [121], Autocrypt [122], and ClaimChain [123]), but it does not hide the sharing patterns (the sending of emails).

One seemingly working solution to avoid the linear broadcast is to set a fixed bound N for the number of capabilities that a user can download during an interval T, and a user downloads exactly

N capabilities every interval T. In case of insufficient capabilities, the user pads the number to the bound N. If a user cannot retrieve all the capabilities (more than N), the user retrieves the rest of them the next time. Unfortunately, this solution leaks use patterns, as we now discuss.

Consider the following scenario: When users receive file capabilities, they may subsequently perform a few noticeable operations, e.g., adding a new line to the file. If a user has too many incoming capabilities and many of them are deferred to be downloaded the next time in this approach, other users may notice this user's delayed responses to some files and learn that more than N files are sent during an interval. Inevitably, avoiding this leakage requires each user to retrieve all his/her capability ciphertexts as in Metal-SHARE.

Making broadcast efficient. Though capability receiving has to be linear, Metal-SHARE improves the efficiency and makes it practical for the client. In Metal-SHARE, the capabilities are encrypted symmetrically under the user's broadcast key (derived from the broadcast master key, which is secret-shared between the servers). As a result, the encryption, transmission, and decryption costs become small. Concretely, if the broadcast list has 10^4 capabilities, a user only needs to download 1 MB and can decrypt all of them in 10 ms.

2.7 Performance

In this section we discuss Metal's asymptotic efficiency and concrete efficiency, compare Metal with PIR-MCORAM and AnonRAM-poly, and compare Metal with Primitive Metal to show how Metal's techniques improve the performance.

2.7.1 Asymptotic efficiency

Since each function (described in Table 2.2) of Metal may be called independently by the users' clients, we describe each function separately. We first describe the notation we will use.

Notation. We consider that the system supports N_{user} users, $N_{\mathsf{file}} \geq N_{\mathsf{user}}$ files in total, file size D, and N_{anonym} anonyms per user, as well as a broadcast list with N_{list} entries. As follows, we use $O_\lambda(\cdot)$ to express the complexity that hides a polynomial of the security parameter λ, while $N_{\mathsf{user}}, N_{\mathsf{file}}, N_{\mathsf{anonym}}, N_{\mathsf{list}}$ are polynomially bounded by λ. Like [100], we parameterize Circuit ORAM with $\frac{1}{N^{\omega(1)}}$ failure probability (in Metal, 2^{-80}).

Cost for CreateAccount, NewAnonym, and SendCapability. Given that $N_{\mathsf{user}}, N_{\mathsf{file}}, N_{\mathsf{anonym}}$ are polynomially bounded by the security parameter λ, we write their asymptotic efficiency in a way that only depends on λ for simplicity:

– The client's and each server's compute time is $O_\lambda(1)$.

– The server-server communication is $O_\lambda(1)$.

– The client-server communication is $O_\lambda(1)$.

Cost for ReadFile and WriteFile. ReadFile and WriteFile have the computational complexity and communication complexity that are linear to the file size D and poly-logarithmic to the number of files N_{file}. Concretely,

- Each server's compute time is $O_\lambda((D + \log^2 N_{\mathsf{file}}) \log N_{\mathsf{file}}) \cdot \omega(1)$.
- The client's compute time is $O_\lambda(D)$.
- The server-server communication (including the communication for SS-DOT and distributed permutation) is $O_\lambda((D + \log^2 N_{\mathsf{file}}) \log N_{\mathsf{file}}) \cdot \omega(1)$.
- The client-server communication is $O_\lambda(D)$.

Cost for ReceiveCapability. ReceiveCapability mainly returns the user the requested part of the broadcast list. At the end of each epoch, Server 1 shuffles the encrypted capabilities, incurring an additional cost. Given that $N_{\mathsf{user}}, N_{\mathsf{file}}, N_{\mathsf{anonym}}$ are polynomially bounded by the security parameter λ, we can write the asymptotic efficiency as follows:

- Server 1's compute time is $O_\lambda(N_{\mathsf{list}} \log N_{\mathsf{list}})$ when shuffling.
- The client's compute time is $O_\lambda(N_{\mathsf{list}})$.
- The client-server communication is $O_\lambda(N_{\mathsf{list}})$.

2.7.2 Implementation

We implemented Metal in C/C++. We use Obliv-C [124] for Yao's protocol with the improved half-gate garbling scheme for multi-instance security, as described in [125, 126], which provides 125-bit security. We use Absentminded Crypto Kit [127] for ORAM and OpenSSL for TLS.

For Metal-AC's authenticated encryption, we use the EAX mode [128], which we deemed to be the most efficient mode for our setting after an extensive search.

The rerandomizable encryption in Metal-ORAM is implemented using ElGamal encryption over Curve25519-Ristretto group [106–108] with a constant-time encoding and a few optimizations. This scheme is about $80\times$ faster than standard ElGamal encryption over a Schnorr group [105] and $200\times$ faster than standard Paillier encryption [114] for our setting.

2.7.3 Evaluation Setup

Machine configuration. We used two `r4.2xlarge` machines on Amazon EC2 as the servers, one in Northern California, one in Oregon, each with eight CPUs and 61 GB memory. We situated them in different regions to simulate the real-world scenario that the servers are in different trust domains. The user ran in a `t2.xlarge` machine in Canada with four CPUs and 16 GB memory. We allocated the user in Canada to simulate that the user is from a remote location. In our experiment, we measured the latency from this machine. Metal-ORAM's DataORAM is stored on Server 1's Amazon gp2 volume.

Network latency. We measured the round-trip time (RTT) and bandwidth. The inter-server RTT was 19 ms, and the client-server RTT was 70 ms. Measured under AWS's guidelines [129], the inter-server bandwidth per connection was $\approx$ 290 MB/s, and the client-server bandwidth was $\approx$ 17 MB/s.

API functions	**Time (s)**	**Time (s)**
	without Tor	**with Tor**
CreateAccount	0.417 ± 0.001	4.1 ± 0.2
ReadFile / WriteFile	3.62 ± 0.01	8.0 ± 0.2
• client preprocessing	0.056 ± 0.006	
• server accessing IndexORAM	0.134 ± 0.001	
• server encryption	0.007 ± 0.001	
• server fetching	0.126 ± 0.001	
• server eviction	1.526 ± 0.002	
• server threshold decryption	0.038 ± 0.001	
• client postprocessing	0.009 ± 0.001	
NewAnonym	0.03 ± 0.01	
SendCapability	1.13 ± 0.01	4.56 ± 0.19
ReceiveCapability	1.759 ± 0.002	6.2 ± 0.4

Table 2.3: The latency of user-facing API functions, measured with and without Tor, for a store of 2^{20} 64 KB files. The result is the average of one hundred measurements, with the confidence interval under two-sided Student's t distribution with 90% confidence.

2.7.4 Metal's performance

To measure the latency of each operation in Metal, we use a setup with 2^{20} 64 KB files (in total 64 GB of data). To measure the latency of receiving a capability from the servers, we have a user download 10^4 capabilities from Server 1's broadcast list.

We measured the latency of these operations with and without Tor in Table 2.3. As we remark in Section 2.2.2, Metal can use other anonymity networks beside Tor. We evaluated on Tor [88] because it is a popular tool. The results without Tor more cleanly show the overhead specific to Metal.

We show the end-to-end benchmark result in Table 2.3, together with a breakdown of the cryptographic operations in our file access API. From the table, we can see that the latency for each file access is a few seconds. The latencies for creating an account and sending/receiving capabilities are also small.

We show the measurements of how the latency of a file access depends on the file size and the number of files. Figure 2.10 has an exponential x-axis for number of files and a linear y-axis for time. We can see the latency increases linearly to the file size and grows logarithmically to the number of files.

For large-scale measurement, we measured the setup with 2^{20} 1 MB files (where each file is padded to 1 MB). It takes 28.8 ± 0.1 s to access a file. Though Metal works with fixed-sized files, larger files can be segmented into smaller files of fixed size, and clients fetch file segments instead of files as the user needs them. As a result, the cost to access the file then depends on the number of segments.

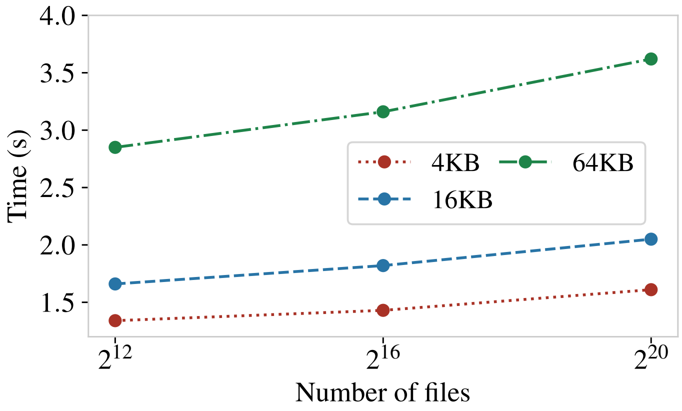


Figure 2.10: File access latency vs. the number of files / file size. The result is the average of one hundred measurements.

Metal-ORAM does not support parallel accesses since Circuit ORAM is not parallel. Thus, a user who wants to access a file has to wait for previous accesses to be complete. This restriction has the benefit of strong consistency. But, one who wants to make Metal-ORAM more parallelizable can distribute the computation (described in Section 2.9) or extend our techniques to parallel ORAM (e.g., Circuit OPRAM [130]).

Network I/O and the size of garbled circuits. We measured the inter-server network I/O and the size of garbled circuits (the number of AND gates) in Table 2.5 for different ORAM implementations. We can see that the network I/O grows almost logarithmically to the number of files. The circuit size, which represents the amount of computation in Yao's protocol, does not grow with the file size.

2.7.5 Comparison with PIR-MCORAM

As we discussed in Section 2.10, only PIR-MCORAM [67] simultaneously provides file sharing and some access patterns protection in the presence of malicious users. Unlike Metal, PIR-MCORAM leaks user identities, but it has the advantage that it only uses a single server. However, this single-server setting results in a latency that is at least linear to the number of files, which becomes slow. Metal's latency, on the other hand, is sublinear to the number of files.

Since PIR-MCORAM [67] is not open-source, we could not perform an end-to-end evaluation of it. Nevertheless, the evaluation in PIR-MCORAM's paper [67] provides measurements and discusses the linear behavior of the results (the main cost is in the zero-knowledge proofs). Hence, we can extrapolate the results for amortized and worst-case time from PIR-MCORAM. We did not compare with TAO-MCORAM [67, 79], another system in the same paper, since TAO-MCORAM uses a trusted proxy, which is not a fair comparison with Metal.

Table 2.4 shows the results. We can see that when the file size and the number of files are

File size	**Amortized time (s)**		**Worst-case time (s)**	
	$\mathbf{2^{16}}$ files	$\mathbf{2^{20}}$ files	$\mathbf{2^{16}}$ files	$\mathbf{2^{20}}$ files
4 KB	≈ 15	≈ 135	≈ 3000	≈ 47600
16 KB	≈ 39	≈ 519	≈ 10900	≈ 190200
64 KB	≈ 135	≈ 2055	≈ 47600	≈ 760700

Table 2.4: Latencies extrapolated from PIR-MCORAM [67].

large, PIR-MCORAM has a high latency, especially the worst-case time. For 2^{20} 64 KB files, its amortized time for file access is $\geq 500\times$ Metal's latency (of 3.62 s). Also, PIR-MCORAM leaks user identities, which Metal hides.

2.7.6 Comparison with AnonRAM-poly

AnonRAM-poly [65] is another anonymous storage system that also uses the two-server model but does not support file sharing among users. AnonRAM-poly [65] is not implemented. To estimate a lower bound of its latency we implemented the zero-knowledge proofs for file uploading used in AnonRAM-poly—which the users generate and the servers verify for every access. We implemented them using the disjunctive Schnorr's protocol [131–133], and we evaluated its performance under the same evaluation setup as in Section 2.7.3. Even with multi-threading, such zero-knowledge proofs take ≥ 80 s per file access for a store of 2^{20} 64 KB files. In addition, AnonRAM-poly has other expensive components, such as the zero-knowledge proofs in oblivious PRF and oblivious sorting between the two servers. AnonRAM-poly is therefore at least $\geq 20\times$ slower than Metal.

2.7.7 Comparison with Primitive Metal

We described Metal's primitive construction in Section 2.5.2, which directly comes from Yao's protocol and does not use Metal-ORAM techniques to move file data out of Yao's protocol. It provides the desired functionality and security but not efficiency. To demonstrate the poor performance of Primitive Metal, we measured the latency of a single ORAM access of the strawman using three state-of-the-art ORAM schemes [100–102] with the implementation in Absentminded Crypto Kit [127]. Note that these implementations only support in-memory storage, which makes them prone to running out of memory but enjoy faster I/O since Metal stores data on the gp2 storage. Table 2.5 shows the measurements, which we now discuss alongside with how Metal improves it, on three dimensions:

- **Storage overhead reduction.** Table 2.5 shows that Primitive Metal soon runs out of memory because the implementation has a storage size blowup of $128\times$. Instead, Metal stores the data outside S2PC and therefore has a smaller blowup.

# Files / File size	Amortized time (s)			Network I/O (MB)			# $\times 10^6$ AND gates			Worst-case time (s)		
	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}
(This paper) Metal, as discussed in Section 2.7.4. This is the end-to-end benchmark including time for Metal-AC.												
4 KB	1.34	1.43	1.61	25.7	39.0	55.2	1.43	2.19	3.16	$\star$	$\star$	$\star$
16 KB	1.66	1.82	2.05	32.0	46.7	64.4	1.43	2.19	3.16	$\star$	$\star$	$\star$
64 KB	2.85	3.16	3.62	60.7	81.4	105	1.43	2.19	3.16	$\star$	$\star$	$\star$
Primitive Metal using Circuit ORAM [100], as discussed in Section 2.7.7, considering only ORAM access time.												
4 KB	3.56	4.46	†	430	508	†	13.5	16.0	†	$\star$	$\star$	$\star$
16 KB	13.6	16.2	†	1690	1976	†	53.1	62.2	†	$\star$	$\star$	$\star$
64 KB	55.5	66.8	†	6717	7844	†	212	247	†	$\star$	$\star$	$\star$
Primitive Metal using SqORAM [101], as discussed in Section 2.7.7, considering only ORAM access time.												
4 KB	3.10	14.7	†	319	1477	†	6.57	30.2	†	437	9810	†
16 KB	†	†	†	†	†	†	†	†	†	†	†	†
64 KB	†	†	†	†	†	†	†	†	†	†	†	†
Primitive Metal using Floram [102], as discussed in Section 2.7.7, considering only ORAM access time.												
4 KB	3.62	4.55	9.76	100	129	290	2.77	2.77	2.77	4.04	9.33	88.0
16 KB	6.67	9.80	31.2	399	514	1152	11.0	11.0	11.0	7.54	30.1	342
64 KB	19.5	32.3	†	1592	2048	†	44.1	44.1	†	24.0	110	†

Table 2.5: Metal-ORAM's file access latencies compared with Primitive Metal ($\star$ = same as amortized, † = out-of-memory). Network I/O is measured using `iftop`. All results are the average of (at least) one hundred ORAM write operations.

- **Latency reduction.** By extrapolating Table 2.5's result, we estimate the file access latency for Primitive Metal is ≥ 75 s for 2^{20} 64 KB storage (for Circuit ORAM [100]). Metal uses tree-based ORAM, which has a poly-logarithmic worst-case complexity and significantly less computation. In particular, Metal avoids the linear worst-case time as in SqrtORAM [101] and Floram [102].
- **Network I/O reduction.** Metal reduces the amortized network I/O because it no longer does data-intensive computation in Yao's protocol (as shown in the circuit size in Table 2.5) and only transfers a few file data blocks per file access. If we extrapolate Table 2.5's result, the amortized network I/O of Primitive Metal is ≥ 1 GB accessing a 64 KB file in 2^{20} files. In comparison, the network I/O for Metal is about 105 MB, as Table 2.5 shows.

2.8 Security proof of Metal-ORAM

In Section 2.4 and Section 2.6.1 we provided the security proof sketches for Metal-AC and Metal-SHARE, which we believe are sufficient to reconstruct a formal proof. In this section we prove the security of Metal-ORAM in the following real-ideal paradigm:

– In the **real world** two servers run the Metal-ORAM protocol. An adversary A sees the view of one of the servers and can control a set of users in a malicious way.

– In the **ideal world** an ideal functionality $\mathcal{F}_{\text{Metal-ORAM}}$ realizes Metal-ORAM with the desired security guarantees. The simulator Sim forges A's view like in the real world.

Metal-ORAM is secure if A's output in the real world is computationally indistinguishable from Sim's in the ideal world.

2.8.1 Ideal functionality

The ideal functionality $\mathcal{F}_{\text{Metal-ORAM}}$ stores the file data in array $\mathsf{FileData}$, where $\mathsf{FileData}[\mathsf{ID}_F]$ stores the file identified by ID_F. $\mathcal{F}_{\text{Metal-ORAM}}$ has the following interface:

Configure. Ask Sim which server to compromise, denoted by $\mathsf{CompromisedSrvID} \in \{1, 2\}$.

Read. On receiving $(C_F^{P_F}, \mathsf{NewFileData}, \mathtt{READ})$ from a user in two shares, check if $C_F^{P_F}$ is valid (using Metal-AC), check $\mathsf{NewFileData}$'s format, and if both checks pass,

- obtain ID_F from Metal-AC, find $\mathsf{FileData}[\mathsf{ID}_F]$, and secret-share $\mathsf{FileData}[\mathsf{ID}_F]$ into $\mathsf{TargetData}^{(1)}$ and $\mathsf{TargetData}^{(2)}$; send $\mathsf{TargetData}^{(1)}$ and $\mathsf{TargetData}^{(2)}$ to the user.
- send Sim $(\mathtt{NEW\_REQUEST}, \mathsf{TargetData}^{(j)}, \mathsf{NewFileData}^{(j)}, C_F^{P_F,(j)})$ where $j = \mathsf{CompromisedSrvID}$. (This message reflects the communication between the user and the compromised server.)

If $C_F^{P_F}$ is invalid or $\mathsf{NewFileData}$ is malformed, send the user and Sim $\mathtt{INVALID\_CAPABILITY}$ or $\mathtt{INVALID\_FORMAT}$, respectively. For invalid formats, send which shares are malformed.

Write. On receiving $(C_F^{P_F}, \mathsf{NewFileData}, \mathtt{WRITE})$, check $C_F^{P_F}$ and $\mathsf{NewFileData}$. If both checks pass,

- obtain ID_F and change $\mathsf{FileData}[\mathsf{ID}_F]$ to $\mathsf{NewFileData}$.
- use the dummy file data as $\mathsf{TargetData}$, secret-share it into $\mathsf{TargetData}^{(1)}$ and $\mathsf{TargetData}^{(2)}$, and send them to the user.
- send Sim $(\mathtt{NEW\_REQUEST}, \mathsf{TargetData}^{(j)}, \mathsf{NewFileData}^{(j)}, C_F^{P_F,(j)})$ where $j = \mathsf{CompromisedSrvID}$.

Otherwise, send to the user and Sim $\mathtt{INVALID\_CAPABILITY}$ or $\mathtt{INVALID\_FORMAT}$, as described above.

2.8.2 Simulator

The simulator Sim learns from $\mathcal{F}_{\text{Metal-ORAM}}$ certain information about a request and knows the system setup, such as the number of files supported by the servers. Sim works as follows:

Initialize. Run A and control A's execution and network. Let A choose $\mathsf{CompromisedSrvID}$ and forward it to $\mathcal{F}_{\text{Metal-ORAM}}$. Sample two ElGamal key pairs, one for each server. Send both public keys and the compromised server's private key to A. Merge the public keys into one global public key. If $\mathsf{CompromisedSrvID} = 1$, instantiate DataORAM.

Initiate a file access request. When A wants to send a request to the servers, forward the request to $\mathcal{F}_{\text{Metal-ORAM}}$.

Forge the compromised server's view. On receiving a message from $\mathcal{F}_{\text{Metal-ORAM}}$, simulate the compromised server's view and provide this view to A. In what follows, we describe the case when A compromises Server 1, and we omit the case for Server 2, which is similar.

If the request is valid, parse the message from $\mathcal{F}_{\text{Metal-ORAM}}$ as $(\mathtt{NEW\_REQUEST}, \mathsf{TargetData}^{(1)}, \mathsf{NewFileData}^{(1)}, C_F^{P_F,(1)})$ and run as follows:

- invoke the simulator for Yao's protocol for the capability check.
- simulate the view in which Server 1 checked the format of $\mathsf{NewFileData}^{(1)}$ and exchanged the results of the format check with Server 2.

- invoke the simulator for Yao's protocol for ORAM access to a random path in IndexORAM and for sharing a fake block location (denoted by i); this step corresponds to reading the file index in Section 2.5.3.
- simulate the SS-DOT to act as if Server 2 would obtain the i-th block on that path (or a dummy block), as follows:
  - invoke the simulator of Yao's protocol for the first step of SS-DOT, which, within S2PC, reconstructs i, samples $N = |\mathsf{stash}| + 3 \times h + 1$ keys, and outputs these N keys to Server 1 and the i-th key to Server 2.
  - simulate the view where Server 1 read blocks from the (fake) storage, encrypted and rerandomized the blocks as the protocol specifies, and sent the encrypted blocks to Server 2.
- simulate the threshold decryption as follows:
  - encrypt $\mathsf{TargetData}^{(1)}$ with the global public key (the ciphertext is denoted by $\mathsf{FakeReadData}$).
  - simulate the view where Server 1 engaged in the threshold decryption of $\mathsf{FakeReadData}$ with Server 2, obtained $\mathsf{TargetData}^{(1)}$, and sent $\mathsf{TargetData}^{(1)}$ to the user.
- simulate the joint encryption of the user-provided new file data (in secret shares), as follows:
  - encrypt $\mathsf{NewFileData}^{(1)}$ provided by the user.
  - simulate the view in which Server 1 received a random encrypted data block from Server 2 and homomorphically added the ciphertext together; the resultant ciphertext is denoted by $\mathsf{FakeNewData}$.
- simulate the distributed permutation as follows:
  - sample a permutation $\sigma^{(1)}$ of the numbers $\{1, 2, ..., |\mathsf{stash}| + 6 \times h - 1\}$.
  - invoke the simulator for Yao's protocol for the ORAM eviction in IndexORAM and the permutation generation inside S2PC, where Server 1 received $\sigma^{(1)}$.
  - simulate the view where Server 1 constructed an array of the size $|\mathsf{stash}| + 6 \times h - 1$, which began with data blocks from the two paths selected by the reverse lexicographic order and followed by $\mathsf{FakeReadData}$ and $\mathsf{FakeNewData}$.
  - simulate the view where Server 1 rerandomized and permuted the array according to $\sigma^{(1)}$ and sent the array to Server 2.
  - sample an array of $|\mathsf{stash}|+6\times h-1$ encrypted dummy data blocks, denoted by $\mathsf{FakePermutedArray}$.
  - simulate the view where Server 1 received from Server 2 $\mathsf{FakePermutedArray}$ and stored it in the storage.
- provide Server 1's view to A.

If the request is invalid because $C_F^{P_F,(1)}$ is invalid, proceed as follows:
- invoke the simulator for Yao's protocol for the capability check, which fails.
- simulate the view where Server 1 responded to the user that the request was invalid.
- provide Server 1's view to A.

If the request is invalid because the data format is incorrect (though $C_F^{P_F,(1)}$ is valid), proceed as follows:
- invoke the simulator for Yao's protocol for the capability check, which passes.
- simulate the view where Server 1 performed the format check of $\mathsf{NewFileData}^{(1)}$.
- simulate the view where Server 1 exchanged the format check results with Server 2 and re-

sponded to the user that the request was invalid.

– provide Server 1's view to A.

Continue running A and output whatever A outputs.

2.8.3 Proof of indistinguishability

We use the following hybrids (denoted by $\mathsf{H}.$) to show that the view that Sim forges is computationally indistinguishable (denoted by $\approx$) from the compromised server's view in the real world. As follows, we focus on the case where Server 1 is compromised, and particularly, the situation when Sim receives a `NEW_REQUEST` message from $\mathcal{F}_{\text{Metal-ORAM}}$.

Consider q requests, where q is polynomially bounded by the security parameter. Our proof will replace the simulated view of each of the q requests one by one, starting from the first request, with the view in the real execution for the same q requests. We use $\mathsf{H}_{t,i}$ to denote the i-th sub-hybrid of the t-th hybrid, in which t requests have been handled. We start with $\mathsf{H}_{0,0}$, which is the same as the simulated view in the ideal world. For each $t \in \{0, 1, ..., q-1\}$, we define:

- $\mathbf{H}_{t,0}$ is $\mathsf{H}_{0,0}$ (if $t = 0$) or $\mathsf{H}_{t-1,7}$ (if $t \neq 0$). In the following hybrids, we focus on the handling of the $(t+1)$-th request.
- $\mathbf{H}_{t,1}$ replaces the simulated view of capability check with the real execution's view. Security of S2PC implies $\mathsf{H}_{t,1} \approx \mathsf{H}_{t,0}$.
- $\mathbf{H}_{t,2}$ replaces the simulated view of the ORAM access to IndexORAM with the real execution's view. Both views have the same distribution of ORAM access patterns. Security of S2PC implies $\mathsf{H}_{t,2} \approx \mathsf{H}_{t,1}$.
- $\mathbf{H}_{t,3}$ replaces the simulated view of the first step of SS-DOT with the real execution's view where both views generate the same N random keys. Security of S2PC implies $\mathsf{H}_{t,3} \approx \mathsf{H}_{t,2}$.
- $\mathbf{H}_{t,4}$ replaces the simulated view for threshold decryption with the real execution's view. In the simulation, Sim encrypts $\mathsf{TargetData}^{(1)}$, pretending to be from Server 2, and has Server 1 decrypt this ciphertext, in which the view of Server 1 (receiving and decrypting) has the same distribution as in the real execution. Thus, we have $\mathsf{H}_{t,4} \approx \mathsf{H}_{t,3}$.
- $\mathbf{H}_{t,5}$ replaces the simulated view for joint encryption with the real execution's view. The difference between the two views is that, in $\mathsf{H}_{t,4}$, Server 1 receives a random data block, while in $\mathsf{H}_{t,5}$, Server 1 receives the ciphertext of $\mathsf{NewFileData}^{(2)}$, encrypted by Server 2. Using semantic security of ElGamal encryption, we have $\mathsf{H}_{t,5} \approx \mathsf{H}_{t,4}$.
- $\mathbf{H}_{t,6}$ replaces the simulated view for ORAM eviction in IndexORAM and for permutation generation with the real execution's view that generates the same share of permutation $\sigma^{(1)}$ for Server 1. Security of S2PC implies $\mathsf{H}_{t,6} \approx \mathsf{H}_{t,5}$.
- $\mathbf{H}_{t,7}$ replaces the simulated view for the rest of the distributed permutation (the parts after S2PC) with the real execution's view, which writes the data blocks into the storage (and updates the rest of the transcript accordingly). The main difference is that Server 1 receives a random data block array instead of the one permuted by $\sigma^{(2)}$. Because these data blocks are encrypted under randomness unknown to Server 1, Server 1 cannot distinguish these two arrays in different views. The rest is the same, and thus $\mathsf{H}_{t,7} \approx \mathsf{H}_{t,6}$.

The last hybrid, $\mathsf{H}_{q-1,7}$, has the same distribution as the real world's view. The hybrid arguments show that the simulated view is computationally indistinguishable from the real world's view. Therefore, we have the following theorem:

Theorem 1. *(informal) Assuming standard cryptographic assumptions and in the random oracle model, Metal-ORAM's protocol provides the claimed security guarantees.*

2.9 Extensions

In this section we discuss certain extensions to Metal.

Parallel accesses. We can leverage a system technique called quorum consensus [134]. For example, we can improve the read performance by having K pairs of Metal servers with the same file data but independent ORAM store. They can load-balance the user's read requests and are very likely to improve the throughput by $K\times$. The write performance will decrease because a user needs to submit the write request to all K pairs of the servers. In systems where the write requests happen very infrequently, such a design can be helpful in reducing the average latency. Note that this direction to improve the performance often has to sacrifice the read/write indistinguishability.

Padding to hide timing and type of operation. The leakage of timing and type of operation can be hidden by padding in time and computation. To do so, we first modify Metal server API functions to support a *dummy mode* that does not make any actual change but exhibits the same execution patterns. We will not discuss how to implement this dummy mode, but it will rely mostly on general techniques. Then, we ask each user's client to routinely call each server API function; when a client is expected to call a server API function but has nothing to do, the client simply invokes the function in the dummy mode. Nevertheless, such padding is very expensive (e.g., the broadcast list will be lengthy).

2.10 Related work

We organize the related work in the following categories:

(1) E2EE storage systems. A line of storage systems uses end-to-end encryption. Academic works include DepSky [41], M-Aegis [44], Mylar [42], Plutus [40], ShadowCrypt [43], Sieve [45], and SiRiUS [39]. In industry, there are Keybase [135], PreVeil [136], and Tresorit [137]. These systems are practical, but they leak user identities and file access patterns.

(2) Anonymous storage systems. There has been a line of works on anonymous storage systems. Earlier academic works include Eternity [138], Publius [139], Freenet [140], and Free Haven [141]. Some peer-to-peer file-sharing systems have been deployed in the real world, including Napster, Gnutella, and Mojo Nation [142].

(3) Single-user oblivious storage systems. Oblivious storage systems are designed to conceal file access patterns and provide strong privacy. Single-user oblivious storage systems focus on the setting where there is only one user [68, 69, 103, 143–147] or a group of *trusted* users that can be treated as one user's multiple clients [76–80, 148]. A number of works add the support of asynchronous access [76–80] and improve the security against malicious servers [148]. Multi-cloud ORAM [85, 149] uses multiple non-colluding servers to achieve a high throughput, but it does not support malicious users using the same ORAM store.

(4) Multi-user oblivious storage systems. Multi-user oblivious storage systems are more challenging since every single user is not fully trusted. There are only a few works in this direction [64–67, 150, 151]. We discuss them as follows.

Secret-write PANDA [64] is a multi-user oblivious storage that does not support data sharing, and thus differs from Metal in terms of functionality. Another disadvantage is that it needs to bound the number of malicious users—which is difficult for systems with open membership—and the performance degrades linearly to this bound. If the number of malicious users is not bounded sufficiently at the setup of the scheme, the scheme cannot provide the desired privacy guarantees. In addition, secret-write PANDA runs expensive cryptography, and the performance is likely very impractical.

AnonRAM-lin and AnonRAM-poly [65] enable many mutually distrusting users to use the same ORAM storage anonymously, but these users cannot share files. AnonRAM-lin has a linear overhead, the performance of which can be prohibitive. And though AnonRAM-poly has a polylogarithmic overhead, extending it with file sharing is hard: it reveals at which level the data block is in the Goldreich-Ostrovsky ORAM (GO-ORAM) [68, 69, 152], which involves file access history. This is not a problem in AnonRAM-poly because users do not share files. But, if we add file sharing, a group of users sharing the same file will now learn information about one another's access patterns. Fixing this problem requires substantial changes. Moreover, AnonRAM-poly has a linear worst-case overhead, which is undesirable for practical systems [75].

GORAM [66] is a multi-user oblivious storage system with anonymity and obliviousness against servers. Its limitation is that GORAM does not provide obliviousness against malicious users, which makes GORAM harder to be used for open systems like Dropbox [153] where any user can sign up. In addition, GORAM does not hide the owner of a file.

PIR-MCORAM [67] is a multi-user oblivious file-sharing system that uses a single server and hides a very large class of metadata, including file access patterns (except whether the operation is reading or writing), but it reveals the user identities to the server when the user writes to a file. Metal improves over PIR-MCORAM by avoiding the linear complexity and hides the user identities in both reading and writing. Compared with Metal, PIR-MCORAM has the benefit of using only one single server.

There are also some multi-user ORAM schemes that focus on multiple semi-honest users' sharing files [150, 151].

(5) RAM-model secure computation. Primitive Metal builds on top of RAM-model secure computation (RAM-SC) [84, 85, 100–102, 154]. With Primitive Metal being limited in functionality and performance, Metal represents a comprehensive solution for file storage.

(6) Miscellaneous. Secure messaging [89–94, 96] also strives to hide metadata in user communication. Nevertheless, it does not store data persistently and usually requires users to stay online, which is difficult in practice. Metadata-hiding storage can also be constructed using hardware enclaves [155–157], but it requires additional hardware assumptions.

Chapter 3

Titanium: A Metadata-Hiding File-Sharing System with Malicious Security

End-to-end encrypted file-sharing systems enable users to share files without revealing the file contents to the storage servers. However, the servers still learn metadata, including user identities and access patterns. Prior work tried to remove such leakage but relied on strong assumptions. Metal (NDSS '20) is not secure against malicious servers. MCORAM (ASIACRYPT '20) provides confidentiality against malicious servers, but not integrity.

Titanium is a metadata-hiding file-sharing system that offers confidentiality and integrity against malicious users and servers. Compared with MCORAM, which offers confidentiality against malicious servers, Titanium also offers integrity. Experiments show that Titanium is $5\times$ to $200\times$ faster or more than MCORAM.

3.1 Introduction

Many companies offer cloud storage with end-to-end encryption, such as BoxCryptor [158], Icedrive [159], Keybase Filesystem [135], MEGA [160], pCloud [161], PreVeil [136], Sync [162], and Tresorit [137]. The enthusiasm in end-to-end encryption stems from the public's concerns about how personal data is misused [163] and how hackers have stolen databases of large enterprises [164].

However, end-to-end encryption is not the end, because cloud servers still see metadata. Metadata such as whom the user shares files with is similar to communication privacy—the Stanford MetaPhone study [165] found that phone call metadata is *"densely interconnected, susceptible to re-identification, and enables highly sensitive inferences"*. A former NSA General Counsel said, *"Metadata absolutely tells you everything about somebody's life"* [46].

Extracting secrets from access patterns has been an important area of security research, with much success. There are works that deanonymize users using social connections [49, 50, 52–55, 166–168], compromise encrypted databases with access patterns [61, 169–174], and break secure

End-to-end encrypted storage	**Metadata-hiding file-sharing systems**
Alice and Journalist have accounts	Users remain anonymous
Alice and Journalist share file F • Alice has write permission • Journalist has read permission	F's access control list is unknown
Alice wrote to F on May 26 Journalist read F on May 27	Someone accessed some file on May 26 Someone accessed some file on May 27

⚠ On May 28, the scandal was reported

Figure 3.1: Comparison of security guarantees between end-to-end encrypted storage and metadata-hiding file-sharing systems.

	Security				**Function**	**Server overhead**	
Scheme	**Anonymity**	**Malicious users**	**Malicious servers**	**Corruption threshold**	**File sharing**	**Comp.**	**Inter-server comm.**
GORAM [66]	No	No	No	$(1, 1)$	Yes	polylog	N/A
PIR-MCORAM [67]	No	Yes	No		Yes	linear	
AnonRAM-lin [65]	Yes	Yes	No		No	linear	
FHE-MCORAM [177]	Yes	Yes	Partial†		Yes	linear	
AnonRAM-polylog [65]	Yes	Yes	No	$(1, 2)$	No	polylog	polylog
Metal [36]	Yes	Yes	No	$(1, 2)$	Yes	polylog	polylog
DPF-MCORAM [177]	Yes	Partial†	Partial†	$(\sqrt{N} - 1, N)$ ‡	Yes	linear	N/A
Titanium (this paper)	Yes	Yes	Yes	$(N - 1, N)$	Yes	polylog	polylog

Table 3.1: Comparison of cryptographic metadata-hiding file storage systems. † FHE-MCORAM does not offer integrity against malicious servers, and DPF-MCORAM does not offer integrity against malicious servers or users. ‡ DPF-MCORAM supports only $N = 4, 9, 16$, i.e., security against 1-out-of-4, 2-out-of-9, or 3-out-of-16 corrupted servers, which is indeed weaker than those under $(N - 1, N)$, such as 1-out-of-2, 2-out-of-3, and 3-out-of-4.

hardware with memory access patterns [175, 176]. These attacks might also be applied to end-to-end encrypted file-sharing systems.

To understand the impact of metadata leakage, consider an application (Figure 3.1) and how it would benefit from metadata protection. A whistleblower, Alice, wants to report a company's scandal to a journalist. If they communicate via the end-to-end encrypted storage, the servers know that Alice shares files with a journalist and when the files are accessed. If the servers collude with the company, the company may find out the whistleblower. Moreover, a hacker or a malicious employee of the cloud may already know the whistleblower's identity.

Alice and the journalist need a storage system that hides metadata from the servers. This system must have anonymity, so the server does not learn whom it is talking to. It must hide access patterns, so the server cannot infer the user behaviors.

Does such a metadata-hiding file-sharing system exist? In the last decade, researchers have been trying to design practical metadata-hiding storage [36, 65–67, 177]. This is challenging because there is almost nothing to trust: both users and servers can be malicious. We need malicious security.[1]

3.1.1 The need for malicious security

The first step toward malicious security is to handle malicious users. For anonymity, there must be many users, and we cannot assume that none of them are malicious. Over the years, security against malicious users has been achieved, as shown in Table 3.1.

In contrast, there is no solution to guarantee security against malicious servers. Several constructions [36, 65–67] all assume *semi-honest* servers. A recent work [177] is the closest to this goal, but it does not offer integrity against malicious servers.

Malicious security should be the standard for distributed applications, rather than semi-honest security. This is because malicious attacks can even look *indistinguishable* from honest executions, so the attackers will *never* be caught for behaving maliciously. The possibility of undetectable malicious attacks is concerning to users who need metadata privacy to protect themselves, such as Alice and the journalist.

In Section 3.2, we present several malicious attacks on semi-honest constructions. Though these attacks fall beyond the scope of semi-honest assumptions, our goal is to show why malicious security is necessary. Particularly, one of the attacks, ciphertext-substitution attack (in Section 3.2.1), can decrypt the entire storage in Metal [36], and the attack is indistinguishable from normal execution, so the attacker will never be caught.

Moreover, integrity, which is not ensured in [177], is critical. Malicious users should not be able to write to other users' files. Malicious servers should not be able to serve incorrect or outdated files without being caught. Achieving integrity in the presence of a malicious adversary is challenging, and sometimes impossible. For example, in any single-server construction, it is impossible to have integrity [178–183] against the malicious server because the server can always present different versions of the files to different users, which we discuss in Section 3.2.2.

3.1.2 Toward efficient file access

As one may expect, hiding access patterns will incur significant overhead, and we cannot expect it to catch up with existing cloud storage like Dropbox [153]. However, we want it to be at least practical enough so that users who need metadata privacy can use it at a reasonable cost.

The dominating overhead is server computation. As Table 3.1 shows, some prior constructions have a linear overhead, where they perform *linear passes on the entire storage* to hide access patterns, while others have a smaller overhead.

Linear passing is expensive. For example, in a file-sharing system with ten million files, when the user writes to a file of 16KB, it needs to write at least 150 GB to the disk. Even if the disk

[1] *Malicious security* ensures security against adversaries who can behave arbitrarily to compromise privacy and integrity of the system. This is in contrast to *semi-honest security*, where adversaries will follow the protocol faithfully.

is a solid-state drive, it would take twelve minutes.[2] Moreover, solid-state drives naturally cannot sustain such massive write accesses [184, 185]. A common solid-state drive with a lifetime write limit of 150 TB can only sustain 1000 such writes, and then be disposed as e-waste.

For efficiency, it is necessary to avoid linear passes. This requirement rules out the single-server construction due to a lower bound by Maffei, Malavolta, Reinert, and Schröder [67]: in a single-server file-sharing system, hiding file access patterns must incur a linear overhead. This lower bound holds even for the semi-honest server, meaning that the server can simply look at the trace of disk accesses and infer file access patterns unless linear passes over the entire storage are used.

For this reason, our construction, named Titanium, distributes trust across multiple servers like [36, 65, 177]. Titanium can tolerate up to $N-1$ out of the N servers being maliciously corrupted. This is the best we can achieve without being subject to the lower bound. Titanium avoids linear passes by using a sublinear oblivious access algorithm on the multi-server model, Circuit ORAM [100], and making improvements that reduce its eviction overhead by up to a half.

3.1.3 Titanium's goals and techniques

Titanium offers confidentiality and integrity guarantees against malicious servers and users, as well as a sufficient level of efficiency for sensitive file sharing scenarios. To understand how Titanium achieves these properties, we give a high-level overview of Titanium's techniques, organized by the goals it achieves and our approaches.

Goal 1: security against malicious users. The standard solution to hide access patterns, oblivious RAM (ORAM) [**ORAM:goldreich1987towards**, **ORAM:gentry2013optimizing**, 75, 103, 143, 152, 186, 187], is inherently *single-user*, meaning that if the ORAM storage is shared with malicious users, the privacy vanishes. To share files securely with many users, new approaches are needed.

• *Approach:* We make Titanium secure against malicious users by *minimizing the users' participation* in the protocol. The only operations that the users perform are sending requests and receiving the responses, through an API we define in Section 3.3. The users never touch the data stored on the servers. This approach provides a clean interface and makes it easy to reason about security against malicious users.

As Figure 3.2 shows, when a Titanium user wants to read or write a file, the user sends a request to the N servers. This request is sent in secret shares so that any $N-1$ servers do not know what the request is. The servers together run a **proxy** in an N-party secure computation, where the proxy's state is hidden from the servers. The proxy checks whether the user's request is legitimate, and if so, on behalf of the user, accesses the storage and responds to the user (through the servers). Now, malicious users can at most craft an unauthorized request, but the proxy will not process it. By doing so, Titanium achieves malicious security against users.

Goal 2: integrity for data storage. Prior work cannot guarantee integrity for data in the storage against malicious servers. A malicious server can, for example, perform an rollback attack, where

[2] Measured on an AWS gp2 solid-state drive block storage.

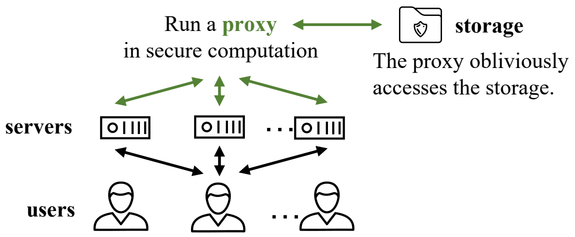


Figure 3.2: Titanium's system model.

a user receives an outdated file from malicious servers. In Section 3.2.2, we show that malicious servers (or users) can break integrity in some prior semi-honest constructions.

• *Approach:* We want files to be written only by users who are authorized to write, and wants all the authorized readers to see the latest version of the file. This requires some sort of message authentication code (MAC), over the entire file storage. What is challenging is that such MACs must not reveal user or file identities. Titanium uses authenticated secret sharing [188] to store the data, which hides the MACs from all the parties and thus retains confidentiality. Moreover, at least one of the servers is assumed to be honest. The MACs enable this server to detect if an incorrect version of the file is sent to the user or another file that the user did not request, thereby ensuring integrity, as shown in Figure 3.3.

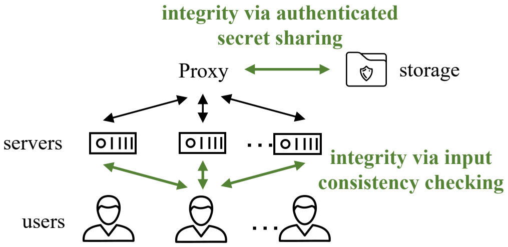


Figure 3.3: Providing integrity in Titanium for (1) data in the storage and (2) user inputs and outputs.

Goal 3: integrity for user inputs and outputs. Though authenticated secret sharing provides integrity for the data storage, it does not provide integrity for data sent to and received from the user. If a malicious server can modify such data, a user may read another file, write data into another file, or share a file with a stranger. Attaching a signature does not work because it is not anonymous. An existing primitive designed for the client-server model, secret-sharing non-interactive proofs (SNIP) proposed in Prio [73, 189, 190], does not work either because it does not offer integrity against malicious servers.

• *Approach:* We design a maliciously secure input/output protocol between the servers and users, so users can confirm that the **proxy** receives the correct inputs and the user receives the correct outputs from the **proxy**, as shown in Figure 3.3. This protocol also provides privacy, as the servers do not see the inputs and outputs. We adopt a tool commonly used in cryptographic proof systems, Schwartz-Zippel lemma [191–193], for this consistency checking.

Goal 4: efficient file access. As Table 3.1 shows, several prior works [65, 67, 177] have a linear server overhead. Though linear-time protocols could sometimes be faster than sublinear protocols, as shown by Floram [102], it is not the case when there are a lot of files. Linear-time protocols may cause a long waiting time for the users and incur an unreasonable amount of cost for the servers.
• *Approach:* Titanium distributes the trust among N servers in a way that it tolerates up to $N-1$ servers being corrupted. This model allows Titanium to avoid the linear lower bound in [67]. Then, Titanium uses an existing sublinear oblivious access protocol, Circuit ORAM [100], and makes improvements to reduce its overhead, as shown in Table 3.2. Our improvement removes a significant amount of unnecessary computation in Circuit ORAM, and is equivalent to the original algorithm.

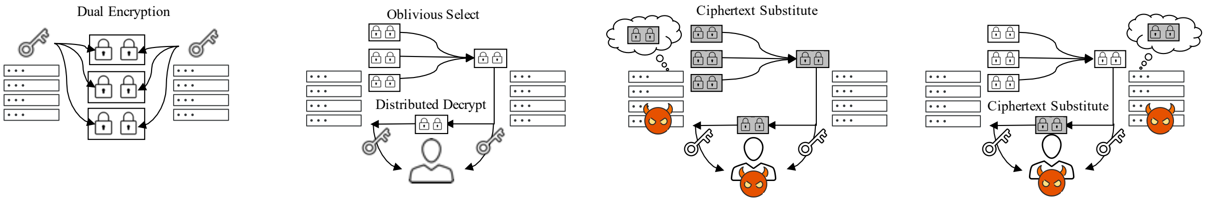
Table 3.2: Comparison with the eviction in Circuit ORAM [100] and our improved eviction. The number of AND gates represents the cost of secure computation in boolean circuits.

	Circuit ORAM [100]	Improved (Section 3.5)	Improvement
# AND gates (4 KB blocks)	14.2 million	7.7 million	1.8×
# AND gates (16 KB blocks)	55 million	29 million	1.9×

3.1.4 Summary of contributions

Our contributions are as follows.

- We study the vulnerabilities of semi-honest file-sharing systems against malicious attackers with concrete attacks (Section 3.2).
- We present Titanium, a metadata-hiding file-sharing system with confidentiality and integrity against both malicious servers and users (Section 3.8).
- We design new protocols for users to communicate with N servers with confidentiality and integrity against malicious servers (Table 3.4).
- We propose an optimized algorithm to perform the Circuit-ORAM eviction [100] in secure computation more efficiently (Section 3.5). Our method reduces the overhead of Circuit-ORAM eviction in secure computation by up to a half.



(a) Metal encrypts the file's data under two keys, where each server has one of the keys.

(b) A file access operation in Metal consists of oblivious selection and distributed decryption.

(c) Server 1 can substitute the ciphertext in oblivious selection and learns its plaintext.

(d) Server 2 can substitute the ciphertext in distributed decryption and learns its plaintext.

Figure 3.4: A data confidentiality attack to Metal [36], in which a malicious server substitutes ciphertexts in the protocol.

3.2 Why malicious security?

In this section, we discuss why semi-honest security is insufficient in practice, by presenting several classes of attacks. We stress that all these attacks fall beyond the threat models of these semi-honest systems, and are not one of their design goals. We show these attacks to support our statement that malicious security is necessary in practice.

3.2.1 Data confidentiality attacks

Semi-honest constructions may be vulnerable against a malicious attacker, which might allow a malicious attacker to break basic security guarantees, such as data confidentiality.

Ciphertext-substitution attack in Metal [36]. Metal assumes two semi-honest servers that do not collude. We show that in Metal, if one of the servers is malicious and colludes with a malicious user, it can learn the content of *any* file of its choice, by substituting ciphertexts in the protocol with the ciphertext that the server wants to decrypt.

As Figure 3.4 shows, (a) in Metal, each file is encrypted under two keys, and each belongs to one server; (b) when the user accesses a file, the two servers run an oblivious selection to locate the ciphertext for the file that the user requests, and then perform a distributed decryption over the ciphertext, using their keys; here, Server 1 provides the candidates for the selection and Server 2 receives the selection result and initiates the distributed decryption over it. Ciphertexts are properly rerandomized so the data access is oblivious.

(c) Server 1 sees all the ciphertexts of the files in the system. If Server 1 wants to decrypt a specific ciphertext C^* (the gray box in Figure 3.4), it colludes with a user (which can be a secret account owned by Server 1) and lets the user initiate an arbitrary file access request. During the protocol, Server 1 replaces all the ciphertexts for oblivious selection with C^*. So regardless of the oblivious selection, Server 2 receives C^*, and will run a distributed decryption protocol for C^*

with Server 1. The user who colludes with Server 1 receives the decryption of C^* and forwards it to Server 1, which concludes this attack.

(d) Server 2 can perform a similar attack as follows. Server 2 also sees a lot of ciphertexts in Metal. If Server 2 wants to decrypt a specific ciphertext C^*, it colludes with a user and lets the user initiate an access request. After the oblivious selection protocol, Server 2 simply ignores the result of the selection and initiates a distributed decryption of C^*, as in Figure 3.4. The user receives the decryption of C^* and forwards it to Server 2.

We stress that this attack is concerning because the malicious attacker will never be caught. Since all the ciphertexts in Metal are re-randomized, they are indistinguishable from one another, and the honest server will never know if the other server is malicious. This limitation may frustrate users, as there is no deterrence for a server to become malicious.

3.2.2 Data integrity attacks

Semi-honest constructions may be vulnerable against a malicious attacker who wants to tamper with the honest user's data. Here we present a few examples.

Ciphertext-substitution attack in Metal [36]. The attack in Figure 3.4 can be used to make an honest user receive a manipulated copy M^* of the file. When an honest user requests a file, the malicious server simply encrypts M^* to obtain ciphertext C^* and performs the same attack with C^*, so that the honest user receives M^* instead of the actual file data.

Overwriting attack in DPF-MCORAM [177]. In [177], a user writes to a file by sending a distributed point function (DPF) to the servers. A "good" DPF only updates the user's file, but a malicious user can craft a "bad" DPF that modifies someone else's file or overwrites the entire storage with random data. An existing solution, verifiable DPF [194], can only address the latter but not the former and is extremely slow in this setting. Solving this problem may require zero-knowledge proofs on AES operations, which is costly. In addition, since each writing operation in [177] changes the entire storage on each server, there are unlikely any frequent backups of the storage, and such attacks may make data unavailable to honest users, which affects data availability.

Rollback attacks in single-server constructions. There is an impossibility result [178–183] saying that single-server storage systems cannot offer integrity guarantees against the malicious server, as the server can always present an old version of the storage to certain users. A separate system, either another server [181] or blockchain [182, 183], has to be used to recover such integrity guarantees. In Titanium, we prevent rollback attacks by authenticated secret sharing that tolerates up to $N-1$ out of N servers being malicious. That is, as long as one honest server has the authentication tag of the storage if other (malicious) servers provide outdated versions, the integrity check will fail with overwhelming probability.

3.2.3 Metadata confidentiality attacks

Selective-failure attacks [113, 195–199] enable an attacker to learn a small amount of metadata based on whether a failure happens. When a malicious server deviates from the protocol, it can observe whether the user receives the correct data or not (through side information). If the user receives the incorrect data and behaves differently, it is considered a *failure*. Whether or not a failure happens may depend on the metadata, so a malicious server can learn some metadata using this attack. We now give two examples of selective-failure attacks in existing systems.

Selective-failure attack in Metal [36]. When Server 1 participates in oblivious selection in Figure 3.4, Server 1 can replace the first K blocks with dummy data. If a failure happens, it means that the user is reading a file that has been accessed *recently*, which is a small metadata leakage. This is because Metal uses a tree-based ORAM construction, where the top of the tree often stores files that are accessed recently.

Selective-failure attack in PIR-MCORAM [67] and FHE-MCORAM [177]. In a single-server construction that uses private information retrieval (PIR), a malicious server can perform the PIR over a database in which a subset of the data is replaced with dummy. If a failure happens, it means that the user is reading a file that belongs to the modified subset, which is a small metadata leakage. This issue is common in single-server constructions based on PIR.

3.3 Overview

In this section, we define Titanium's system model, threat model, and out-of-scope leakages and assumptions.

3.3.1 System model

We consider a file-sharing system with many users and N servers. The servers collaboratively provide storage services to the users. Each user can store a number of files on the servers and share these files with other users, under some access control policies. Each user can read or write a file that the user has permission to.

As Figure 3.5 shows, for a user to perform an operation in Titanium, the user first makes an API request to the servers. The request is in secret shares, so any $(N-1)$ servers cannot recover what is in the request. The N servers, upon receiving the user's request in secret shares, forward the requests to the proxy. The proxy is not a separate entity in the system but is a program executed in secure computation by the N servers. Since the proxy's state is hidden from the servers, we present it as a separate part for ease of presentation. The proxy checks (in secure computation) if the user has permission to perform the action, and if so, interacts with the storage on behalf of the user, such as reading a file or writing to a file.

Next, the proxy sends the API response to the user and asks the servers to forward the response. The response is forwarded also in secret shares, so any $(N-1)$ servers do not see what is in the response. The user reconstructs the proxy's response from secret shares, which concludes an API call in Titanium.

Titanium's API. Users in Titanium interact with the servers through the API defined as follows.

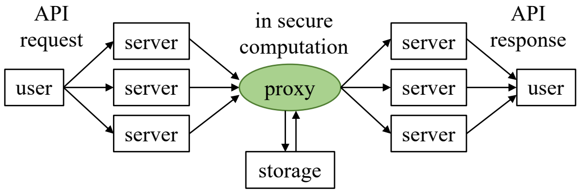


Figure 3.5: The workflow of Titanium.

- $\texttt{CreateAccount}() \to (\texttt{uid}, \texttt{credential})$. A new user joins the file-sharing system by calling this API. If the system has capacity for the new user, the user will be assigned a user ID and the corresponding credential, which is used later by the proxy to authenticate the user.
- $\texttt{CreateFile}(\texttt{credential}) \to \texttt{fid}$. Each user can create a new file by calling this API. If the system has capacity, the proxy assigns a new file, and the user obtains the file ID and becomes the owner of this file. The user can grant and revoke permission of the files to other users.
- $\texttt{Read}(\texttt{credential}, \texttt{fid}) \to \texttt{data}$. A user can request the latest version of a file that the user has read permission to by calling this API. The proxy checks if the user has permission using the credential. If the user has permission, the proxy responds to the user with the file data.
- $\texttt{Write}(\texttt{credential}, \texttt{fid}, \texttt{data}^*) \to \bot$. A user can update a file with new data that the user has write permission to by calling this API. The proxy updates the file accordingly.
- $\texttt{Grant}(\texttt{credential}, \texttt{fid}, \texttt{uid}, \texttt{perm}) \to \bot$. The owner of a file can grant permission (defined as “read” and “write”) to another user, given that the owner knows the other user’s ID, by calling this API. The proxy checks that the caller of the API is indeed the owner of the file and modifies the access control policies of the file accordingly.
- $\texttt{Revoke}(\texttt{credential}, \texttt{fid}, \texttt{uid}, \texttt{perm}) \to \bot$. Similarly, the owner of a file can revoke permission previously granted to another user. The proxy checks the caller’s ownership of the file and updates the file’s access control policies.

Toward more privacy. In Titanium, the user and file IDs are hidden to the servers and to any users that the honest user does not interact with. However, the API above requires the owner to know the recipient’s user ID before sharing, and a recipient of a file knows the file ID after being granted the permission. For more privacy, the user and file IDs can be *hidden* from these users by replacing these IDs with “randomized user ID” and “randomized file ID”. A user can have many randomized user IDs that can be provided to a file owner to gain permission, while the owner cannot link this randomized ID to the other ID that a user has. Similarly, the owner does not need to provide the unique file ID to each other user who has access to the file, but a randomized file ID suffices.

Such randomization has been done and formalized in [36], where randomized user and file IDs are called *anonyms* and *capabilities*, respectively. For ease of presentation, Titanium’s API does not specify these IDs to be randomized, but Titanium can support it by adopting these primitives (which are pretty lightweight) directly from [36].

3.3.2 Threat model

Titanium's threat model is as follows. Up to $N-1$ out of the N servers can be malicious and collude with one another. We assume at least one server is honest, which does not collude with any corrupted servers. All the corrupted servers can arbitrarily deviate from the protocols.

Titanium can tolerate an arbitrary number of users to be malicious and collude with one another and with the corrupted servers. Malicious users may, for example, attempt to access the data of honest users even though they do not have permission. For honest users to remain anonymous from the servers in the system, they are expected to use some sort of anonymity network where the IP addresses and communication patterns, such as latency, do not reveal the user identities. Tor [88] can be one of such solutions, but Titanium can also work with other anonymous communication solutions.

Metadata-hiding properties. Titanium offers all the metadata-hiding guarantees covered in existing works, modeled as follows. Let $\mathcal{A}$ denote the adversary that controls all the corrupted parties. Let U be the honest user who has access to file F.

(a) *Data secrecy and integrity.* If U has never granted read or write permission of F to a malicious user corrupted by $\mathcal{A}$, then $\mathcal{A}$ neither learns anything about F nor modifies F.

(b) *Read obliviousness.* $\mathcal{A}$ cannot distinguish which data block was read by U, even if $\mathcal{A}$ has full permission on the entire storage. That is, $\mathcal{A}$ cannot distinguish a read operation from another read operation, by another honest user, to another file.

(c) *Write obliviousness.* If U has never granted the read permission of F to a malicious user corrupted by $\mathcal{A}$, then $\mathcal{A}$ will not realize if F is updated. But if someone among the corrupted users has read permission to F, they legitimately learn that F has been changed, but if at least two honest users have write permission to F, $\mathcal{A}$ does not know who has changed it.

(d) *Read/write indistinguishability.* If no one in $\mathcal{A}$ has read permission on F, then $\mathcal{A}$ does not know whether an honest user reads or writes to F.

(e) *Anonymity.* $\mathcal{A}$ cannot distinguish which honest user made the access requests, if the honest users communicated with the servers in a way that hid network information.

Formalization. We define the security of Titanium in the real-ideal paradigm [200], and show that Titanium securely realizes an ideal functionality, shown in Figure 3.6, in Section 3.11.

Definition 2. A protocol Π is said to securely realize $\mathcal{F}_{\mathsf{FileSharing}}$ in the presence of static malicious adversaries that compromise up to $N-1$ out of N servers, if, for every non-uniform probabilistic polynomial time adversary $\mathcal{A}$ in the real world, there exists a non-uniform probabilistic polynomial time simulator $\mathcal{S}$ in the ideal world, such that the outputs of the two worlds are computationally indistinguishable.

Out-of-scope leakages and assumptions. In this paper, we make the following assumptions: (1) DoS attacks are out-of-scope; (2) the size of a file is fixed, so there is no size leakage; (3) all the requests are processed in sequential order. These are also standard assumptions in prior work [36, 100, 145, 148, 177, 201].

There are some solutions to partially remove these assumptions. To mitigate DoS attacks from users, anonymous payment or client puzzles can be used (see Section 3.11.4). For size leakage, to our knowledge, there is no efficient way to prevent it without padding to the largest file size (which is extremely costly), but there is mitigation including partial padding [202], differential privacy, delayed accessing by downloading different chunks of the file at different times, or only accessing part of the file that is needed. Finally, to our knowledge, it is unclear how to enable parallel oblivious access without requiring a strong assumption [79, 203–205]. Thus, we leave it as an open-research problem.

3.4 Making the proxy's Access to the Storage maliciously secure

In this section, we describe the instantiation of the Titanium storage and proxy and how they are made maliciously secure.

3.4.1 File storage with authenticated secret sharing

Data in the file storage is secret-shared among the N servers using authenticated secret sharing [206]. In Titanium, we represent the file storage as elements in a field $\mathbb{F}$. Each of the N servers has a share of every field element x, and the sum of these shares equals x, as illustrated below.

$$\sum_i x^{(i)} = x \in \mathbb{F}$$

where $x^{(i)}$ is the share of x held by the i-th server. The secret-sharing of a field element x effectively hides the value of x from the servers, thereby providing confidentiality.

Each field element is authenticated by a MAC $m = \alpha \cdot x$, where α is the MAC key. The MAC is secret-shared among the N servers in a similar way. Each server has a share of m, and the sum equals m, as illustrated below.

$$\sum_i m^{(i)} = m \in \mathbb{F},\ m = \alpha \cdot x \in \mathbb{F}$$

The MAC key, α, is also secret-shared among the N servers. This allows them to perform a few operations over these secret shares. For example, the N servers can work together to add, subtract, or multiply two secret field elements that are represented in secret shares, and the computation results are also in secret shares. The servers can also perform integrity checks on the computation results. Therefore, when a malicious server manipulates the results of the computation, an honest server among the N servers can detect such manipulation and refuse to release the incorrect results to the user. As a result, when a user in Titanium receives the response from the N servers, the user is assured that an honest server has checked the response.

In Titanium, the proxy performs two basic types of computation over the file storage. The first is oblivious data selection, in which the proxy is given two field elements and wants to select one

Ideal functionality $\mathcal{F}_{\mathsf{FileSharing}}$

(For ease of presentation, we omit the session sid "sid" below.)

Initialize. Upon invocation, it sends $(\texttt{init})$ to Sim. $\mathcal{F}_{\mathsf{FileSharing}}$ maintains the following maps:

- a credential-user map $\mathcal{C}$ from credential to user ID uid
- a file-storage map $\mathcal{D}$ from fid to file data
- a file-permission map $\mathcal{P}$ from $(\texttt{fid}, \texttt{uid})$ to a permission attribute set $p \subseteq \{\text{‘}\texttt{read}\text{’}, \text{‘}\texttt{write}\text{’}, \text{‘}\texttt{own}\text{’}\}$.

CreateAccount. If the system has capacity for the new user, it creates a random credential, sets $\mathcal{C}[\texttt{credential}] := \texttt{uid}$, sends $(\texttt{uid}, \texttt{credential})$ to the user, and sends a signal $(\texttt{create-account})$ to Sim.

CreateFile. If the system has capacity for the new file, it creates a new file with ID fid, assigns the user with owner permission $\mathcal{P}[\texttt{fid}, \texttt{uid}] := \{\text{‘}\texttt{read}\text{’}, \text{‘}\texttt{write}\text{’}, \text{‘}\texttt{own}\text{’}\}$, and sends a signal $(\texttt{create-file})$ to Sim.

Read. If the user with ID uid and correct credential sends a request to read a file with ID fid, it checks if $\text{‘}\texttt{read}\text{’} \in \mathcal{P}(\texttt{fid}, \texttt{uid})$. It returns the data $\mathcal{D}[\texttt{fid}]$ (or dummy if the user is not authorized) to the user and a signal $(\texttt{access})$ to Sim.

Write. If the user with ID uid and correct credential wants to write new data to a file with ID fid, it checks if $\text{‘}\texttt{write}\text{’} \in \mathcal{P}(\texttt{fid}, \texttt{uid})$, and if so, it updates $\mathcal{D}[\texttt{fid}]$ and sends a signal $(\texttt{access})$ to Sim.

Grant. When the user with ID uid and correct credential wants to grant read or write permission of file with ID fid to another user $\texttt{uid}^*$, it first checks if $\text{‘}\texttt{own}\text{’} \in \mathcal{P}(\texttt{fid}, \texttt{uid})$, and if so, adds the corresponding permission to $\mathcal{P}(\texttt{fid}, \texttt{uid}^*)$ and sends a signal $(\texttt{grant})$ to Sim.

Revoke. When the user with ID uid and correct credential wants to revoke read or write permission of file with ID fid to another user $\texttt{uid}^*$, similarly, it first checks if $\text{‘}\texttt{own}\text{’} \in \mathcal{P}(\texttt{fid}, \texttt{uid})$, and if so, drops the corresponding permission from $\mathcal{P}(\texttt{fid}, \texttt{uid}^*)$ and sends a signal $(\texttt{revoke})$ to Sim.

Figure 3.6: The file-sharing ideal functionality $\mathcal{F}_{\mathsf{FileSharing}}$.

of them in secure computation. The proxy performs such selection in an oblivious manner so that the servers running the secure computation do not know which one is being chosen. Each selection operation incurs an overhead, so for efficiency, we want to do as few selections as possible. This is reflected in Titanium's effort to reduce the overhead of Circuit ORAM in Section 3.8, which minimizes the number of data selections.

The second is to compute a random linear combination of a (large) number of field elements, which is used in our input/output checking protocol described in Section 3.6. This can be done efficiently, which the input/output checking protocol leverages.

3.4.2 Running the proxy in secure multiparty computation

In Titanium, the proxy receives the API request from users, accesses the file storage, and sends the API response to the users. This implies that the proxy knows all the secret information in Titanium, and therefore its state and execution must be hidden from the servers and users. To do so, Titanium runs the proxy in an N-party secure computation, which ensures that up to $N-1$ out of the N servers cannot see what is being executed in the secure computation, and if malicious servers want to manipulate the results, the honest servers can detect such discrepancy and refuse to provide the incorrect results to the users.

Secure multiparty computation [81, 82, 86, 200, 207–209] enables N parties to evaluate a function $f(x_1, \ldots, x_N)$ where the i-th party provides input x_i. The results, denoted by $(y_1, \ldots, y_N)$, are released to the parties, such that the i-th party receives y_i. This allows the proxy to have its private state.

Secure computation comes with an overhead, as it runs much slower than plaintext computation. Therefore, Titanium must minimize the amount of computation. The dominating part of the computation is the *data selection* during the oblivious accesses. We discuss how we reduce this overhead in Section 3.8.

In practice, we can alleviate the user from waiting for the secure computation to finish and receiving the API response, by having the N servers precompute some of the proxy's secure computation before a user sends an API request and use the precomputation to run the secure computation faster. We analyze how this can be useful for users in Section 3.9.

3.5 Making the proxy's access to the storage oblivious and efficient

In this section, we describe how we make the proxy in Titanium access the file storage obliviously and efficiently.

- The first requirement, obliviousness, means that the servers should not know which file is being accessed or which users have access to this file. This is necessary to hide the metadata, as we describe in Section 3.3.2.
- The second requirement is efficiency, which is to make the Titanium protocol practical enough for certain use cases where such a high level of privacy is needed.

Titanium makes the following efforts to fulfill these two requirements. Titanium leverages a state-of-the-art oblivious RAM protocol, Circuit ORAM [100] and then improves its eviction procedure (the main overhead) for better efficiency, which achieves an improvement of up to $2\times$.

3.5.1 Background on Circuit ORAM

We now present some necessary background of Circuit ORAM for readers to understand how our improvement works.

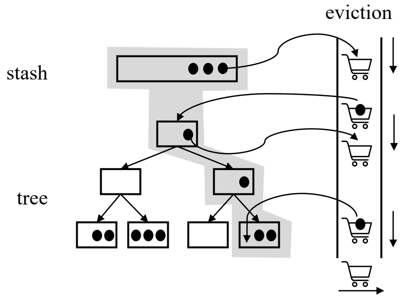


Figure 3.7: A "small shopping cart" illustration of how the Circuit ORAM eviction works.

Circuit ORAM enables the proxy in Titanium to access the file storage, without revealing which file is being accessed. To do so, Circuit ORAM assigns a random location for each file, and when a file is being accessed, it is assigned to a new random location. As a result, for any sequence of file accesses, the locations being accessed on the physical storage are random and do not depend on which files are being accessed.

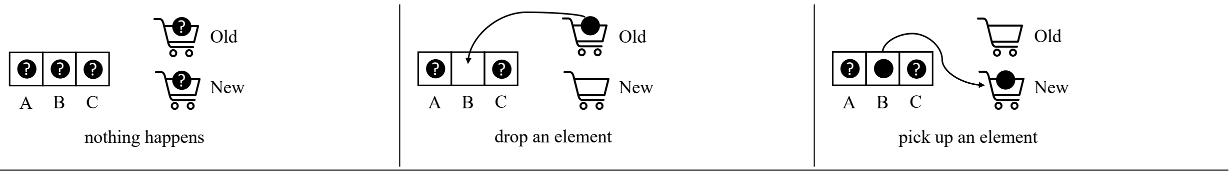
The challenge that Circuit ORAM faces is to *securely* and *efficiently* move the file to a new location, after each access. First, the moving of the file must be done in a way that hides the new location, otherwise, the servers can associate the old and new locations. Second, the moving must be done efficiently, meaning that it should ideally access only a few locations on the physical storage. A linear pass of the entire storage, in which security can be achieved trivially, is too expensive.

The solution is to use layers of "write caches" for the storage, and instead of moving the file to the new location, the file is first moved to the write cache. These caches are, during the subsequent accesses to the storage, gradually being evicted to lower levels of caches, and eventually to the actual locations where the file should reside. For this reason, when the proxy wants to read a file, the proxy also needs to look at the write caches, as the file may have not yet been evicted out from layers and layers of write caches.

We illustrate the write cache that Circuit ORAM uses in Figure 3.7, which follows a binary tree structure, of k levels. The leaf layer of the tree contains 2^{k-1} buckets, where each bucket can store Z files (often $Z = 3$). All the other layers, including the root, are write caches for their descendants. An extra array, called stash, is the top level of the write caches. Readers who are familiar with CPU architecture can consider the stash as the L1 cache, the root of the tree as the L2 cache, and the layer immediately before the leaf layer as the $\mathrm{L}(k+1)$ cache.

A problem that write caches must handle is space, since the stash and each bucket in the tree has a limited size, files in the write cache must be relocated to lower layers of the cache, or the leaf layer of the tree, to release some space for higher layers of the cache. Circuit ORAM provides an eviction procedure for this purpose. Such eviction is in essence similar to cache replacement for CPU.

This eviction procedure can be illustrated with "small shopping cart", as in Figure 3.7. Circuit



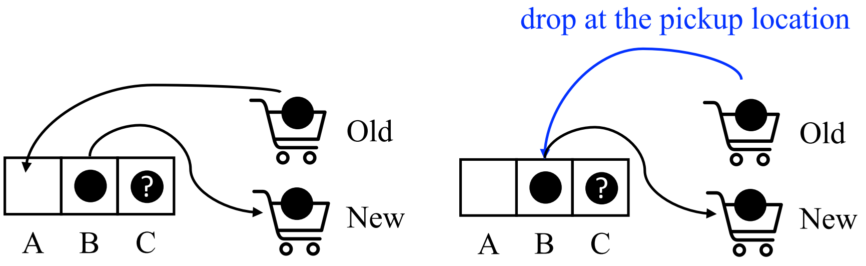
nothing happens

drop an element

pick up an element

Circuit ORAM (6 conditional data operations)		
`cond_set(new <= A, false)`	`cond_set(new <= A, false)`	`cond_set(new <= A, false)`
`cond_set(new <= B, false)`	`cond_set(new <= B, false)`	`cond_set(new <= B, true)`
`cond_set(new <= C, false)`	`cond_set(new <= C, false)`	`cond_set(new <= C, false)`
`cond_set(A <= old, false)`	`cond_set(A <= old, false)`	`cond_set(A <= old, false)`
`cond_set(B <= old, false)`	`cond_set(B <= old, true)`	`cond_set(B <= old, false)`
`cond_set(C <= old, false)`	`cond_set(C <= old, false)`	`cond_set(C <= old, false)`
Improved version (3 conditional data operations)		
`cond_swap(A <=> old, false)`	`cond_swap(A <=> old, false)`	`cond_swap(A <=> old, false)`
`cond_swap(B <=> old, false)`	`cond_swap(B <=> old, true)`	`cond_swap(B <=> old, true)`
`cond_swap(C <=> old, false)`	`cond_swap(C <=> old, false)`	`cond_swap(C <=> old, false)`
`new <= old`	`new <= old`	`new <= old`

Figure 3.8: Comparison of the eviction procedure in Circuit ORAM and the improved version. Part 1: the simple cases.



pick and drop an element in Circuit ORAM

pick and drop an element in the improved version

Circuit ORAM (6 operations)	Improved version (3 operations)
`cond_set(new <= A, false)`	`cond_swap(A <=> old, false)`
`cond_set(new <= B, true)`	`cond_swap(B <=> old, true)`
`cond_set(new <= C, false)`	`cond_swap(C <=> old, false)`
`cond_set(A <= old, true)`	`new <= old`
`cond_set(B <= old, false)`	
`cond_set(C <= old, false)`	

Table 3.3: Part 2: the case when pick and drop happen in the same level.

For secure computation over boolean data:

```
cond_swap(L <=> R, cond)
– diff = 000...0
– cond_set(diff <= L ⊕ R, cond)
– left <= left ⊕ diff
– right <= right ⊕ diff
```

For secure computation over arithmetic data:

```
cond_swap(L <=> R, cond)
– diff = 000...0
– cond_set(diff <= L - R, cond)
– left <= left - diff
– right <= right + diff
```

Figure 3.9: Implementation of `cond_swap`.

ORAM chooses a random path of the tree, illustrated in a gray background in the figure, and does a pass over all the caches on this path. The algorithm picks a file from the top level of the cache, puts it into a "shopping cart" that is so small that it can only carry at most one file, and can choose to pick or drop files along the way. A file can move to any write cache that is its ancestor. The shopping cart is empty in the end, and write caches along the path are updated.

So far, Circuit ORAM sounds like an efficient algorithm, but in order to achieve obliviousness, the shopping cart must update the buckets along the path, in a way that hides what kind of movement is done. This requires the algorithm, running inside the proxy in secure computation, to perform the same data operations regardless of the eviction plan, and therefore it needs to pad the actual eviction plan with a lot of dummy data operations. As a result, it becomes the main overhead of the entire oblivious file access operation.

Circuit ORAM, in order to reduce the amount of data operations, intentionally makes the eviction *a single pass along the path* and makes the shopping cart *small*, to reduce the number of fake operations used for padding. Circuit ORAM has been implemented in many libraries [101, 102, 210–217]. We now introduce an improved eviction subroutine that cuts the eviction overhead by up to a half, and this approach has not been explored before in the literature or implemented in any existing libraries.

3.5.2 Our improved eviction procedure

One challenge in developing Titanium proxy is to reduce the file access latency. When the file size increases, the eviction overhead in secure computation becomes the dominating cost. An eviction in a store of 2^{20} 4KB files requires conditional data operations on about 1MB of data. For efficiency, it is important to reduce the number of such conditional data operations.

We contribute an improved eviction procedure for Circuit ORAM, which reduces the overhead by up to a half, as already shown in Table 3.2. The new procedure is general-purpose in that it

```
Get the target array from prior steps.
hold $:=\perp$, dest $:=\perp$.
for $i = 0$ to $L$ do
    towrite $:=\perp$.
    if (hold $\neq\perp$) and ($i ==$ dest) then
        towrite := hold.
        hold $:=\perp$, dest $:=\perp$.
    if target[$i$] $\neq\perp$ then
        hold := read and remove deepest block in path[$i$].
        dest := target[$i$].
    Place towrite into bucket path[$i$] if towrite $\neq\perp$.
```

Figure 3.10: The original Circuit ORAM eviction algorithm.

```
Get target, isdeepest, and isfirstempty arrays from prior steps.
hold $:=\perp$, dest $:=\perp$.
for $i = 0$ to $L$ do
    for $b = 0$ to $Z$ do                          ▷ $Z = 3$
        swap := false.
        if target[$i$] $\neq\perp$ then    ▷ pick or pick and drop
            swap := isdeepest[$i$][$b$].
            dest := target[$i$].
        else if $i =$ dest then                    ▷ drop only
            swap := isfirstempty[$i$][$b$].
        swap path[$i$][$b$] and hold if swap is true.
```

Figure 3.11: The improved variant of eviction algorithm.

improves Circuit ORAM in any setting, but the improvement is larger when file sizes are large. The new procedure is equivalent to the original Circuit ORAM algorithm, thereby reusing its security analysis. We believe that existing implementations of Circuit ORAM should use this new procedure.

Our observation is that Circuit ORAM (shown in Figure 3.10) is paying unnecessary overhead due to its *modular design*. At each level, eviction consists of "pick" and "drop". When dropping, the original procedure invokes a general-purpose subroutine to conditionally insert the file. However, such a modular design obscures an opportunity for optimization. There is a specialized subroutine, in which "pick" and "drop" do not need to be separate, and their conditional data operations can be combined.

We formally present our eviction algorithm in Figure 3.11 and compare it with the original Circuit ORAM algorithm in Figure 3.10. But we feel it is easier to understand by illustrating the differences with the "small shopping cart" example again.

As shown in Figure 3.8 and Table 3.3, when the shopping cart arrives at a specific layer, it interacts with the bucket that represents the write cache. We present the cart before the interaction as "old" and the cart after as "new". There are four cases: (1) nothing happens, (2) drop an element, (3) pick up an element, and (4) pick and drop an element. In all these cases, Circuit ORAM performs six conditional-set operations, which amount to six data selections. Especially, in the most complicated case in Table 3.3, six seems necessary because "pick" and "drop" may interact with different slots in the bucket, since Circuit ORAM always drops the element in the first availability in each bucket.

Indeed, the first availability is not a requirement. As the slots in the bucket are equivalent, the file in the old cart can be dropped to any empty slot in the bucket. It happens that after we pick up the file, the place where we pick becomes empty, and is okay to drop the file simply at the location

The proxy outputs data to the user, as follows:

```
1: To send $s_1, s_2, ..., s_n$, the proxy samples a random number $r$ and asks the servers to send shares of $r$ and $\{s_i\}_n$ to the user.
2: The user reconstructs $r$ and $\{s_i\}_n$ from the shares, samples a random $\beta \in \mathbb{F}$, and broadcasts $\beta$ to all the servers.
3: The proxy receives $\beta$ from the servers, computes $f(\beta) \leftarrow r + \sum_{i=1}^{n} \beta^i \cdot s_i$, releases $f(\beta)$ to the servers, and asks each server to forward $f(\beta)$ to the user.
4: The user receives $f(\beta)$, computes $f'(\beta)$ locally, and checks if $f(\beta) = f'(\beta)$.
```

The proxy receives data from the user, as follows:

```
1: The proxy samples some random elements $r_1, r_2, ..., r_n$ and uses the output protocol to deliver them to the user securely.
2: The user broadcasts $s'_i \leftarrow s_i - r_i$ to all servers.
3: The servers provide $\{s'_i\}_n$ to the proxy, which reconstructs $s_i \leftarrow s'_i + r_i$.
```

Figure 3.12: The maliciously secure input/output protocols.

where we pick. With this observation, we can replace the six conditional-set operations with three conditional-swap operations, while each conditional-swap can be efficiently implemented with one conditional-set, for both secure computations based on boolean and arithmetic circuits, as we show in Figure 3.9.

The new procedure also appears simpler than the original algorithm. As shown in Figure 3.11, the swap conditions can be computed simply from two arrays isdeepest and isfirstempty, indicating whether a slot is deepest and whether it is the first empty in the bucket, which are free byproducts of prior steps in the computation of the eviction plan.

3.6 Securing the proxy's communication with users

We now describe the protocol that enables the proxy to communicate with the user securely. The protocols must ensure that malicious servers cannot see the content of such communication or manipulate such communication, as long as at least one of the servers is honest. At the core of this protocol is an efficient batch check protocol that ensures the user receives the same data as what the proxy wants to send. This batch check uses an algebraic tool, Schwartz-Zippel lemma, which is a common tool in cryptographic proof systems.

Batch checking using Schwartz-Zippel lemma [191–193]. Let us consider that the proxy wants to send n field elements to the user, denoted by $s_1, s_2, ..., s_n$. The proxy can sample a random number r and ask the servers to send their secret shares of these field elements to the user, so the user receives $r', s'_1, s'_2, ..., s'_n$ where $r = r'$, $s_i = s'_i$ unless malicious servers have manipulated the data.

Now the user and the proxy want to check if they have the same data. There are many possible ways to do so, such as evaluating a collision-resistant hash function over these field elements. However, since these methods must be evaluated inside the secure computation, they would be much slower than the one based on polynomial identity testing we now present. Titanium lets the proxy and the user each construct a univariate degree-n polynomial using the data.

$$\begin{aligned}\text{Proxy}: f(x) &= r + s_1 \cdot x + s_2 \cdot x^2 + s_3 \cdot x^3 + ... + s_n \cdot x^n\\ \text{User}: f'(x) &= r' + s'_1 \cdot x + s'_2 \cdot x^2 + s'_3 \cdot x^3 + ... + s'_n \cdot x^n\end{aligned}$$

The user then chooses a random point $x = \beta \in \mathbb{F}$ and tells all the servers this point. If at least one server is honest, the proxy can either receive the correct β or detect a malicious attack (and terminate the protocol). Now, both the proxy and the user knows β, they evaluate this polynomial over point β, and the proxy releases $f(\beta)$ to all the servers.

Each server sends $f(\beta)$ to the user. If at least one server is honest, the user can either receive the correct $f(\beta)$ or detect a malicious attack and terminate the protocol. The user checks if $f(\beta) = f'(\beta)$. If so, the Schwartz-Zippel lemma shows that, with a probability of $1 - n/|\mathbb{F}|$, the two polynomials are the same: that is, $s_i = s'_i$. We present the protocols in Figure 3.12.

Reducing the number of client-server rounds with Fiat-Shamir transform [218]. The protocols in Figure 3.12 have two client-server rounds for input and output. It can be reduced to one round, with some small amount of additional computation, if one uses Fiat-Shamir transform. In the output protocol, the servers can first commit to the shares of r and $\{s_i\}_{1,..,n}$ they are going to send to the user and broadcast these commitments to each other. The broadcast needs to be made in a way that servers cannot change their messages after seeing someone else's. This is done by requiring the servers to commit to the message first and then open it. Then, they use a cryptographically secure hash function (modeled as a random oracle) to derive β from these commitments and let the proxy compute $f(\beta)$. So, each server now, in addition to sending the shares, also sends the commitments that they receive from each other, the opening of their own commitments, as well as β and $f(\beta)$ to the user. If the user receives the same set of commitments from all the servers, the user verifies that β is derived correctly, that commitments are opened correctly, and that $f'(\beta) = f(\beta)$. The input protocol, which uses the output protocol as a subroutine, also reduces to one client-server round.

Comparison with prior works. Securing a client's communication to an entity in secure computation has been explored before, by Jakobsen, Nielsen, and Orlandi [219] and Damgård, Damgård, Nielsen, Nordholt, and Toft [220]. Compared with them, our protocol requires much fewer operations in secure computation, at the cost of one more round (a very minor cost in our use case), or no additional round if Fiat-Shamir transform is used, as shown in Table 3.4.

3.7 Performing file access control in the proxy

In Titanium, we want to enable the owner of a file to grant permission to another user and to revoke previously granted permission. A user has one of the five types of permission to a file:

	[219]	**[220]**	**Ours**
Input			
◇ # client-server comm. rounds	1	1	2 or 1
◇ # random shares	$n \cdot N + 3$	$3n$	$n + 1$
◇ # multiplications	1	$2n$	0
Output			
◇ # client-server comm. rounds	1	1	2 or 1
◇ # random shares	$n \cdot N$	$2n$	1
◇ # multiplications	0	$2n$	0

Table 3.4: Comparison with prior works on input/output check. N: number of servers, n: number of inputs/outputs.

Method	**Pros and cons**
Access control matrix (Section 3.7.1)	Pros: efficient revocation, many users can share the same file Cons: large server storage overhead
Access control list (Section 3.7.2)	Pros: efficient revocation; small server storage overhead Cons: a file can only be shared with a few users (or entities)
Capabilities (Section 3.7.3)	Pros: small server overhead; many users can share the same file Cons: users need to store capabilities locally; inconvenient revocation

Table 3.5: Summary of pros and cons of several access control methods that hide metadata.

no permission, read-only, write-only, read-and-write, and ownership. There are several ways to implement this access control without leaking metadata, with different trade-offs, and one may be more suitable than others sometimes. We summarize their pros and cons in Table 3.5.

3.7.1 Our main approach: access control matrix

The main approach for Titanium is to store the access control policies in a matrix of size that is the number of users times the number of files, where each entry of the matrix stores a three-bit value representing which permission the user has for this file. This appears to be practical in a few setups. For example, assuming that the number of users equals $\sqrt{\text{number of files}}$, for a system with 2^{24} files, each file stores about 1 KB of the data for a column in the access control matrix. To check and update permission, the proxy simply does a linear scan. This method allows the owner to grant and revoke permission with the same experience as in traditional file-sharing systems. A file can also be shared with a large number of users. For this reason, we use it as the main approach and evaluate it in Section 3.9.

3.7.2 Alternative: access control list

If the system has many users, the access control matrix becomes impractical. In practice, most files are shared with only a few users, and to handle a file shared with a lot of users, one may instead share a special "group" account between these users instead of adding each of them into the file's access control list. In this case, we can limit the number of users/groups who share a file to a number d that is much smaller than the total number of users in the system (e.g., $d = 10$), which suffices for many use cases, and the overhead is much smaller.

3.7.3 Alternative: capabilities

Another approach, proposed in Metal [36], is, instead of storing access control lists on the servers, the proxy gives users some cryptographic tokens (called "capabilities"), where each token represents permission to a file. To check permission, the proxy checks if the token is valid. This alleviates the servers from storing any access control data. However, in this approach, it is hard to revoke permission: the owner has to invalidate the old file (aka, revoke everyone), create a new file and reshare the permission with any user whose permission is not revoked. The cost increases when the file is shared with many users.

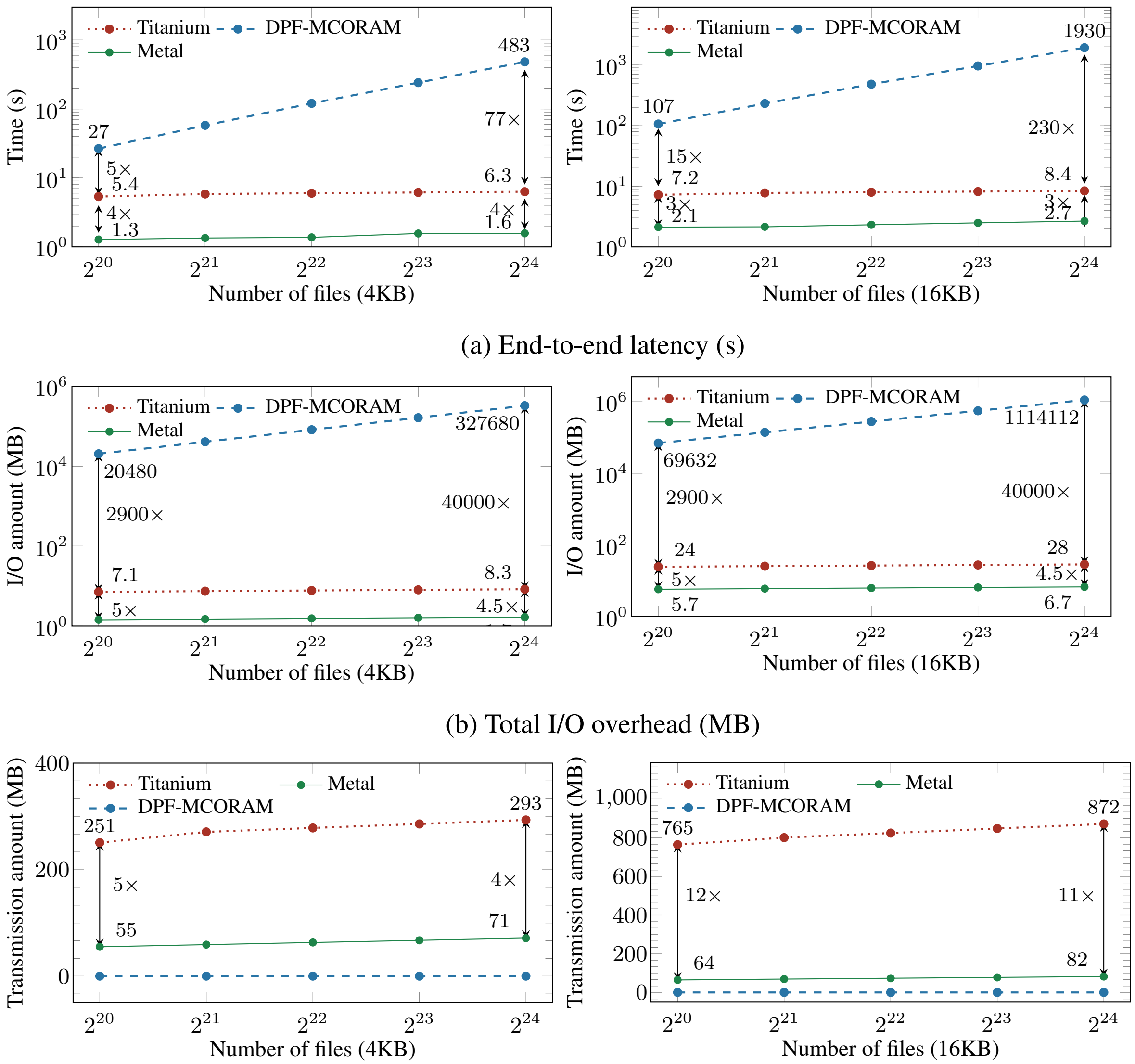
3.8 Putting it together

We have presented all the components of Titanium and showed how they provide malicious security. In this section, we describe how they are put together as a metadata-hiding file-sharing system. For ease of description, we use the example of Alice and the journalist from Section 3.1.

Onboarding. Alice joins the storage system by installing a client software from a trusted source, in which Alice selects a set of N servers who are providing this service and already have a lot of users (for anonymity). Throughout the description, we assume Alice has found a way to hide the network patterns. This includes using the free WiFi services in a café or store, using relays such as Mozilla VPN or Apple Private Relay, using Tor, or a combination of them. The anonymity network does not need to be perfect, as the servers can only know an API request comes from a specific IP address, but do not know what is inside the API request. We assume the client software can block other software on the device from connecting to the Internet during the use of the software, which reduces the risk of de-anonymization due to other software on the device.

Create an account and a file. Alice sends an API request for registration, and the proxy running inside secure computation (Section 3.4.2) on the server-side assigns a user ID and credential to Alice. The journalist joins the system as well. To prepare the materials to be shared with the journalist, Alice creates a file in the storage system via an API call. The proxy assigns a file ID to Alice, and Alice is the owner of the file (Section 3.7). Alice receives the user ID, the credential, and the file ID through the output protocol (Section 3.6) for integrity.

Uploading the file. Alice uploads the file using the client software. The client software makes the API call and transfers the data to the proxy using the input protocol (Section 3.6) for integrity.



(a) End-to-end latency (s)

(b) Total I/O overhead (MB)

(c) Total inter-server network overhead (MB)

Figure 3.13: Comparison between DPF-MCORAM (tolerating 1-out-of-4 corruption), Titanium, and Metal (both tolerating 1-out-of-2).

Then the proxy checks if Alice has permission to write to the file, by looking up the access control matrix (Section 3.7), from which the proxy knows Alice is the owner of the file. If so, the proxy accesses the file storage (Section 3.4.1). The file access is followed by an eviction procedure using our improved algorithm (Section 3.8). In the end, though this is a write request, the proxy still returns a dummy file to make read and write indistinguishable from the servers and network. Alice receives the dummy file using the output protocol (Section 3.6).

Sharing the file. Alice obtains the journalist's user ID through a reliable mechanism, such as from an in-person meeting with the journalist. Alice uses the client software to invoke the API to grant read permission to the journalist, through the input protocol (Section 3.6). The proxy, after checking the integrity of the request and Alice's ownership of the file, modifies the access control matrix to grant the journalist read permission to the file. The journalist, who receives the file ID from Alice in some way, such as through an anonymous broadcast service already shown in Metal [36], can now read the file. To make this read request indistinguishable from a write request, the journalist's client software uploads a dummy file in the input protocol. The remaining operations are similar to writing a file.

Revoking the permission. When the journalist receives the file and reports the scandal, Alice can revoke the journalist's access to the file, as a precaution in case the journalist's account is compromised. To do so, Alice uses the client software to communicate with the proxy and submit the revocation API request using the input protocol (Section 3.6).

3.9 Evaluation

In this section, we answer the following questions:

1. What is the overhead of Titanium?
2. How does Titanium compare with the state-of-the-art?
3. What is the breakdown of the overhead of Titanium?
4. How does the improved ORAM eviction protocol in Section 3.8 compare with the original Circuit ORAM algorithm?
5. If servers can do precomputation, what is the latency that a user in Titanium would experience?

3.9.1 Setup

We implemented Titanium and benchmarked its performance using standard libraries for secure computation, including EMP-toolkit [221], SCALE-MAMBA [217], and MP-SPDZ [215]. Specifically, since MP-SPDZ has an efficient offline phase via LowGear, we used it to generate Beaver triples for secure computation in arithmetic circuits. On the other hand, we used SCALE-MAMBA for the online phase due to its support of mixed circuit. The field size for authenticated secret sharing is 64 bits. Since the users simply interact with the API, our evaluation focuses on the server overhead.

We used `c4.8xlarge` AWS instances for the servers, each with 36 CPU cores and 60 GB RAM. We used Linux `tc` tool to limit the bandwidth of each server to 2 Gbps and added a network round-trip latency of 20 ms between them.

We evaluate on a file storage with 2^{20} to 2^{24} files of size ranging from 4 KB to 16 KB, which we consider to be the practical region of Titanium. As for the access control matrix, we consider $\sqrt{2^{24}} = 2^{12}$ users, the same as in [36].

We note that due to ORAM security, read and write requests in Titanium are provably indistinguishable and incur the same overhead (i.e., memory usage, latency, network communication).

In fact, the user who wants to read a file is also performing a dummy write to the storage and vice versa (see Section 3.9.5).

3.9.2 Performance of Titanium over prior multi-server schemes

To understand the price that Titanium pays for malicious security, we consider Metal, a two-server semi-honest scheme, as our baseline for comparison. We then compare Titanium with DPF-MCORAM scheme in [177] that offers partial malicious security to showcase our advantages. We compare the case of $(t, N) = (1, 4)$ in DPF-MCORAM with $(1, 2)$ in Titanium and Metal (note that $(1, 2)$ is more secure than $(1, 4)$). Given that the experimental evaluation of DPF-MCORAM is not available, we estimated its overhead using the state-of-the-art library in Express [222] for its distributed point functions (DPF).

Figure 3.13 presents the performance of Titanium and its counterparts in terms of end-to-end latency, I/O access, and inter-server communication overhead for each file access request.

As shown in Figure 3.13, in a setup with two servers, the overhead of Titanium grows almost polylogarithmically to the number of files. The overhead also grows linearly to the file size. This matches the expectation of the Titanium algorithm: (1) Titanium uses Circuit ORAM, which can access the storage in time sublinear to the number of files; (2) when the proxy performs data operations on the files, the amount of computation naturally grows linearly to the file size.

Compared with Metal, Titanium is approximately $3 - 4\times$ slower in terms of end-to-end delay. Specifically, Metal takes 1.3 s $-$ 2.7 s to access a file with sizes ranging from 4 KB to 16 KB in storage with 2^{20} to 2^{24} files, while Titanium takes 5.4 s $-$ 8.4 s per access. Titanium also incurs higher I/O access (i.e., around $5\times$) and inter-server network communication (e.g., $5 \times -11\times$) than Metal. This is because Titanium needs to authenticate the data in secure computation, while Metal does not as it only offers semi-honest security.

Compared with DPF-MCORAM, Titanium is several orders of magnitude faster and has lower I/O overhead as shown in Figure 3.13. In the log scale, we can see that the overhead of Titanium grows slowly because Titanium's overhead is polylogarithmic to the number of files, while DPF-MCORAM grows linearly to the number of files (so, it is a straight line in the figure). One advantage that DPF-MCORAM offers over Titanium and Metal is that it does not require servers to communicate with each other as shown in Figure 3.13c. Despite such advantage, DPF-MCORAM requires linear processing and, as a result, its I/O access, and computation cost is two to three orders of magnitude of Titanium's.

3.9.3 Experiments with varying numbers of servers

We evaluated the scalability of Titanium with a varying number of servers. Figure 3.14 presents the access latency of Titanium with two to five servers. We can see that the overhead of Titanium grows almost linearly to the number of servers. This matches the expectation as well because the cost of each operation that the proxy is performing grows linearly to the number of servers running the secure computation. The linear behaviors also show a trade-off between security and efficiency. To reduce metadata leakage for large files, it is preferred to increase the file size limit, but this would

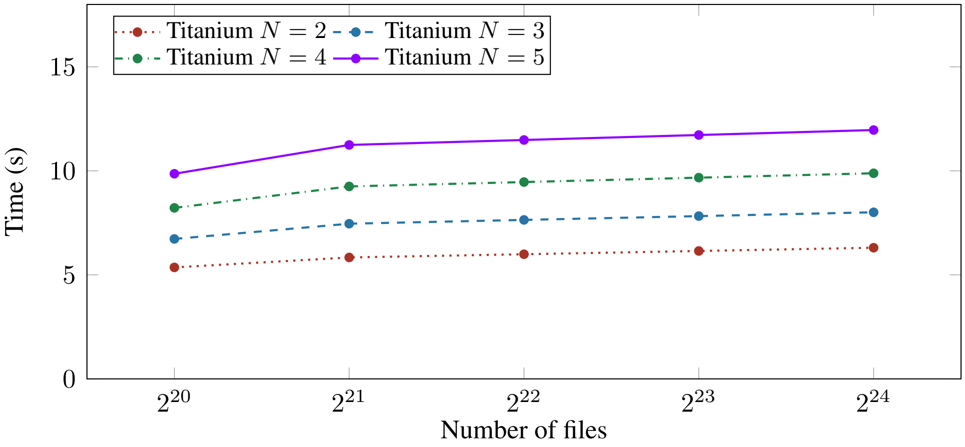


Figure 3.14: Overhead with 4KB files and varying number of servers.

increase the overhead linearly. To better distribute the trust among the servers, it is preferred to increase the number of servers, but this would also increase the overhead. In practice, a user chooses a specific Titanium system depending on the trade-off that the user would like to make.

Titanium is more scalable than DPF-MCORAM when increasing the number of servers for higher corruption tolerance. Remark that DPF-MCORAM needs N servers to tolerate $t = \sqrt{N}$ corruptions while Titanium requires N servers for $t = N - 1$ corruptions. As shown in Table 3.6, the cases of $(t, N) = (2, 9)$ and $(3, 16)$ in DPF-MCORAM, which are less secure than the cases of $(t, N) = (2, 3)$ and $(3, 4)$ respectively in Titanium, are very expensive. This is due to the high overhead of running DPF inside homomorphic secret sharing (which is commonly implemented with leveled fully homomorphic encryption) over a large amount of data. We use Gentry, Halevi, and Smart's implementation [223] on running ciphers in homomorphic encryption to approximate these numbers. Overall, compared with DPF-MCORAM, Titanium has the following advantages: (1) Titanium provides integrity against malicious servers, (2) Titanium does not do linear passes, (3) with N servers, Titanium tolerates $N - 1$ corrupted servers, while DPF-MCORAM tolerates $\sqrt{N}$, (4) Titanium can support many servers, while DPF-MCORAM is restricted to $N = 4, 9, 16$.

Table 3.6: More comparison with DPF-MCORAM.

	DPF-MCORAM	Titanium	Titanium
	(2,9) or (3,16)	(2,3)	(3,4)
2^{20} files	0.6 years	6.7 s	8.2 s
2^{21} files	1.3 years	7.5 s	9.3 s
2^{22} files	2.6 years	7.6 s	9.5 s
2^{23} files	5.1 years	7.8 s	9.7 s
2^{24} files	10.2 years	8.0 s	9.9 s

Note that we did not compare Titanium against Metal in this experiment because Metal is designed for the semi-honest setting and is restricted to the two-server model.

3.9.4 Comparison with single-server counterparts

We compare Titanium with two notable state-of-the-art single-server counterparts including PIR-MCORAM [67] and FHE-MCORAM [177]. As discussed previously, the single-server model means that the system must incur a linear overhead and cannot provide full integrity against malicious servers.

Since PIR-MCORAM does not have an open-source implementation, we extrapolate its results reported in [67] and estimate that it would take at least 100 s for the same file access in our setting.

Though there is neither evaluation nor implementation of FHE-MCORAM, we can estimate a lower bound of its overhead based on the FHE cost. To make FHE bootstrapping efficient, parameter choices are important. An efficient instantiation is shown by Halevi and Shoup [224], using packed FHE ciphertexts on a specific cyclotomic ring, so that the per-plaintext-bit cost of bootstrapping is small. We estimated that the overhead of single file access in FHE-MCORAM, for a store of 2^{20} files of size 4 KB, would take 55 days, and for 2^{24} files of size 16 KB, it would be about nine years.

3.9.5 Cost analysis

We perform a cost breakdown analysis to investigate how each processing phase impacts the performance of Titanium. Table 3.7 presents the detailed costs of Titanium in terms of end-to-end latency, memory usage, and communication when reading or writing a 4KB file in storage with 2^{20} files.

Due to Circuit ORAM, for each file request, Titanium incurs two major processing phases: retrieval and eviction. The retrieval is to read a file from the storage while the eviction is to write a file to the storage. Any file request from the user incurs both retrieval and eviction processing at the proxy. For read requests, the retrieval phase reads the requested file from the storage, while the eviction phase writes the retrieved file back to the storage. For write requests, the retrieval

Table 3.7: Cost breakdown of Titanium per file request in storage with 2^{20} files and file size of 4 KB. Read and write requests incur the same amount of processing overhead.

	Latency (s)	**Memory usage (MB)**	**Inter-server comm. (MB)**
ORAM retrieval • read request: read the requested file • write request: read but ignore	0.53	1.8	108
ORAM eviction • read request: evict the file back • write request: evict with new file	2.7	5.3	142

phase reads the file from the storage but ignores the results, while the eviction phase writes the file with new data to the storage. Therefore, any file request in Titanium incurs the same processing overhead regardless of whether the type of the request is read or write. The proxy decides whether the user sees the file (in the case of "read") and whether the user's input becomes the new data of the file (in the case of "write") using oblivious selection over the request type indicated by the user.

As shown in Table 3.7, we can see that eviction is the most dominating part, especially in end-to-end latency where it contributes more than 80% to the total cost. It is because the eviction needs to compute a complicated eviction plan in secure computation and perform more data operations. The inter-server communication and I/O costs of the eviction phase are also higher than that of the retrieval phase. This is because the eviction in Circuit ORAM performs about twice the number of operations on the files compared with the retrieval phase.

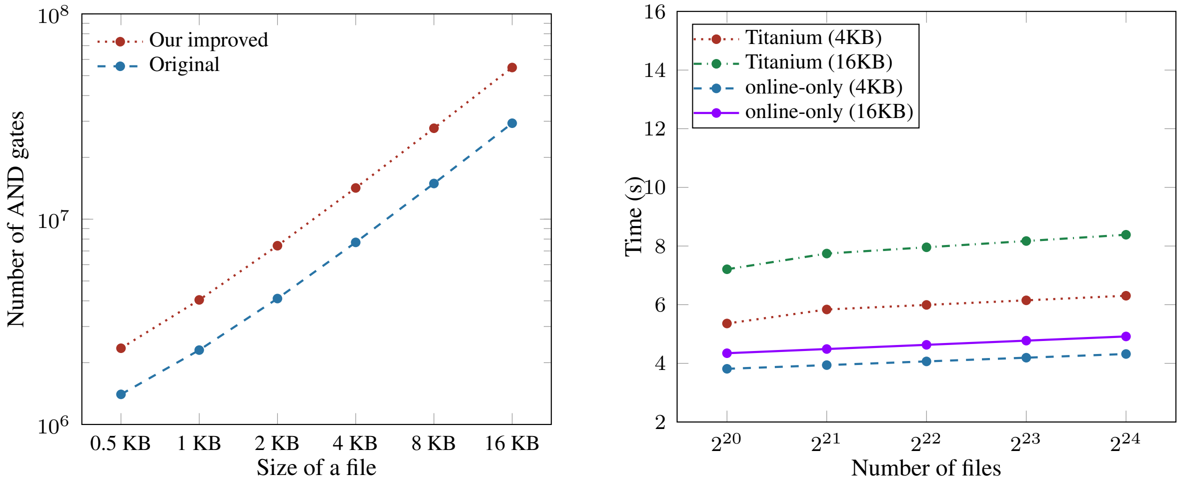
3.9.6 Improvement of ORAM eviction procedure

To understand how the new algorithm in Section 3.8 improves over Circuit ORAM, we evaluate their costs in the eviction procedure. As shown in Figure 3.15a, the computation cost, represented by the number of AND gates in secure computation based on boolean circuits, grows linearly to the file size. For all the file sizes that we consider, the improved protocol performs better than the original protocol. This improvement varies by file size. When the file size is 0.5 KB, the improvement brought by Section 3.8 is $1.7\times$, and when the file size is 16 KB, the improvement becomes $1.9\times$. This is because in Circuit ORAM, besides the data operations, there is a cost to compute the eviction plan, which is independent of the file size. When the file is small, there is still an improvement but is smaller because the cost of computing the eviction plan remains a significant part. When the file is large, the data operations dominate the overhead of eviction, and the improvement gets closer to $2\times$.

3.9.7 User waiting time given precomputation

The main overhead of Titanium is the proxy's operations, which are computed in the N-party secure computation, by the N servers. A common approach to reducing the running time of secure computation is to do precomputation—when there are no API requests, the servers themselves precompute part of the secure computation, so later when a user wants to access a file, the servers can run the proxy with such precomputation, so the running time is shorter. However, the precomputation is one-time. If there is insufficient precomputation, the user will need to wait for the original execution of the proxy. In practice, this is still useful in that it can reduce the user waiting time when the storage system is not overly loaded. Precomputation can also be done by additional machines.

Precomputation is useful for a few Titanium API functions, specifically reading a file. As shown in Figure 3.15b, with two servers, the online-only version, which uses precomputation, reduces the waiting time. The saving is larger with larger file size. For example, the online-only version can reduce the waiting time by 30% when the file size is 4 KB. It becomes 40% when



(a) Eviction

(b) User waiting time

Figure 3.15: (a) Our improved eviction vs. original one; (b) User waiting time in Titanium with vs. without precomputation (b).

the file size is 16 KB. This is because when the file size is small, the network latency, which precomputation does not help, contributes more to the user waiting time.

3.10 Related work

We now discuss the related work, organized into four categories.

Revealing secret information from patterns. Several areas of security have been exploring how to learn secret information from patterns. We know social connections can be used to reveal identities in the social network [49, 50, 52–55, 166–168], encrypted databases may leak access patterns that reveal data [61, 169–174, 225], and secure hardware suffers from side-channel attacks [175, 176].

Single-user oblivious storage. Single-user oblivious storage has been explored in various system settings including single-server model [68, 75, 103, 104, 147, 187, 226] and distributed (i.e., multi-server) model [85, 201, 227–229]. Some other works focus on specific demands for privacy, such as hiding only the write patterns [230–232], enabling parallel accesses [130, 148, 204, 205, 233], and building a usable system [76, 79, 149, 234].

Sharing oblivious storage among many users. Maffei et al. [67] showed that a single-server multi-user oblivious storage must have at least linear server computation for secure data sharing against client-server collusion. As a result, all single-server constructions are either linear or insecure upon collusion [64–66]. It is also impossible to achieve integrity against the malicious server in the single-server setting, as the server can always violate the integrity by serving outdated data to some users [178–181, 183]. A few recent works [**EUROCRYPT19:Hamlin-Ostrovsky-Weiss-Wichs:PANDA**, 36, 65, 177] explored how to share oblivious storage among many mutually distrusting users by distributing the trust to multiple servers.

Oblivious storage using trusted hardware. There is a line of works using secure hardware for oblivious storage systems, generally built on FPGAs [145, 234] and secure enclaves [155, 157, 202, 203, 235, 236]. Although secure hardware-based constructions tend to be very efficient, it requires a strong security assumption on the hardware (e.g., isolation, tamper-resistant, enclaves) unlike Titanium, which only requires the hardware to operate as normal. To hide the access patterns, these protocols often need to perform oblivious operations inside the secure hardware to defend against side-channel attacks [145, 237, 238].

3.11 Proof sketches

We now present our proof for the security of Titanium. The proof is straightforward, as we can simply invoke a simulator for secure computation that captures both the computation in boolean circuits (used to find the file and compute the eviction plan) and the authenticated secret sharing that we use.

3.11.1 Definitions

We first describe the ideal world, the real world, and the simulator. Let $\mathcal{F}_{\mathsf{FileSharing}}$ and Sim denote the ideal functionality and the simulator (i.e., the ideal-world adversary), respectively. Since users are considered anonymous, and the only thing that identifies them is the credential and the user ID. In the following discussion, we assume that adversary $\mathcal{A}$ can decide the requests that clients make through the N servers. The N servers in the ideal world become dummy and simply forward data between the client software and $\mathcal{F}_{\mathsf{FileSharing}}$.

We say that the protocol securely realizes the ideal functionality if the output of the ideal world is computationally indistinguishable from the output of the real world.

Ideal world:

- **Initialization.** $\mathcal{F}_{\mathsf{FileSharing}}$ creates a credential-user map, a file storage map, and a file permission map.
- **Create an account.** Sim can instruct the client software to send a request to $\mathcal{F}_{\mathsf{FileSharing}}$ for a new account, where the client software receives the user ID and a credential, which is then forwarded to Sim
- **Create a file.** Sim can instruct the client software to send a request to $\mathcal{F}_{\mathsf{FileSharing}}$ for a new file, using a user ID and credential. The client receives the file ID, which is then forwarded to Sim.
- **Access a file.** Sim can instruct the client software to send a file access request to $\mathcal{F}_{\mathsf{FileSharing}}$, with a credential, a file ID, and the operation to perform. The client either receives the file data or dummy data, which is then forwarded to Sim.
- **Grant and revoke permission.** Sim can instruct the client software to send a permission change request to $\mathcal{F}_{\mathsf{FileSharing}}$, with a credential, the file ID, the other user's ID, and the change to make.

- **Output.** In the end, Sim outputs the simulated views of the servers and simulated output of the real adversary $\mathcal{A}$.

Real world:

- **Initialization.** $\mathcal{A}$ chooses a number of servers to corrupt. We let $\mathcal{A}$ choose exactly $N - 1$ servers. The servers, following the protocol, initiate the secure computation of the proxy.
- **Create an account.** $\mathcal{A}$ can instruct the client software to send a request to create an account, where the client software receives the user ID and a credential and forwards them to $\mathcal{A}$.
- **Create a file.** $\mathcal{A}$ can instruct the client software to send a request to create a new file, using a credential, where the client receives the file ID and forwards it to $\mathcal{A}$.
- **Access a file.** $\mathcal{A}$ can instruct the client software to send a file access request, with a credential, a file ID, and the operation to perform. The client either receives the file data or dummy data as the response, which is then forwarded to $\mathcal{A}$.
- **Grant and revoke permission.** $\mathcal{A}$ can instruct the client software to send a permission change request, with a credential, the file ID, the other user's ID, and the change to make.
- **Output.** In the end, the servers and $\mathcal{A}$ output their views.

The simulator:

- **Initialization.** We assume Sim has a black-box access to $\mathcal{A}$. Sim asks $\mathcal{A}$ which servers to corrupt and then simulates the transcript of corrupted servers in the initialization protocol to $\mathcal{A}$, by invoking the simulator for secure computation.
- **Create an account.** If $\mathcal{A}$ wants to create an account, Sim forwards this request to the dummy servers and returns the simulated transcript of the corrupted servers to $\mathcal{A}$, by invoking the simulator for secure computation. Sim forwards the user ID and the credential to $\mathcal{A}$.
- **Create a file.** When $\mathcal{A}$ wants to create a file, Sim forwards this request to the dummy servers and returns the simulated transcript of the corrupted servers to $\mathcal{A}$, by invoking the simulator for secure computation and using random paths to simulate the Circuit ORAM traces.
- **Access a file.** When $\mathcal{A}$ wants to perform a file access operation, Sim forwards this request to the dummy servers and returns the simulated transcript of the corrupted servers to $\mathcal{A}$, by invoking the simulator for secure computation and using random paths to simulate the Circuit ORAM traces.
- **Grant and revoke permission.** When $\mathcal{A}$ wants to perform a permission change, Sim forwards this request to the dummy servers and returns the simulated transcript of the corrupted servers to $\mathcal{A}$, by invoking the simulator for secure computation and using random paths to simulate the Circuit ORAM traces, similar to the file access.
- **Output.** In the end, Sim outputs whatever $\mathcal{A}$ outputs.

3.11.2 Security with abort

In Titanium, $\mathcal{A}$ can abort at various stages. We now discuss why such aborting does not help $\mathcal{A}$ learn any secret.

- **Case 1: Aborting the secure computation.** If $\mathcal{A}$ corrupts the servers to perform invalid operations in secure computation, it will be discovered by the honest servers. The honest servers will abort the protocol, and no more operations can be done. The simulator for secure computation can still simulate the transcript of the corrupted parties up to the aborting, without leaking secret data.

- **Case 2: Manipulating a user's input or output.** The user can verify the output from the servers via the input/output protocol. Due to the Schwartz-Zippel lemma [191–193], the user can detect if $\mathcal{A}$ manipulates the response with an overwhelming probability. The same applies to the user's input.

3.11.3 Proof of indistinguishability

We now show that the outputs of the real world and the views of the ideal world are computationally indistinguishable.

Proof. We prove the security using hybrid arguments. That is, we use a sequence of hybrids (denoted by $\mathsf{H}_\bullet$) to show that the output of the ideal world is computationally indistinguishable (denoted by $\approx$) from the output of the real world. Consider that there have been q requests, where q is a number polynomially bounded by the security parameter. We use the same proof strategy as in Metal [36]: we replace the output of each of the q requests one by one with the output from real execution. We let H_t denote the t-th hybrid, in which the outputs of t out of q requests have been replaced. We start with H_0, which is the output of the ideal world.

Without loss of generality, we assume $\mathcal{A}$ does not abort. For each $t \in \{0, ..., q\}$, we define H_t as follows: H_t has the output of the first t requests in the real world, and the output for the remaining requests is in the ideal world (which would be a list of the requests and the output of the simulators). Our goal is to show that for $t \in \{0, ..., q-1\}$, $\mathsf{H}_t \approx \mathsf{H}_{t+1}$. We now consider several cases.

- **Create an account.** Note that H_{t+1} replaces the Sim's simulated output of the servers corresponding to creating an account with the real execution. The simulated output is a direct result of the use of the simulator for secure computation. By the security of the secure computation, we have that the simulated output is computationally indistinguishable from the real execution, and therefore $\mathsf{H}_t \approx \mathsf{H}_{t+1}$.
- **Create a file.** Similarly, H_{t+1} replaces the Sim's simulated output of the servers corresponding to creating a new file (reserving space for the user's new file and giving the user ownership permission to the space) with the real execution. Besides the use of the simulator for secure computation, another difference is that the ORAM traces (for updating the access control matrix) in the ideal world are sampled from uniform random. Due to the security of Circuit ORAM, the traces in the real execution are also uniformly random and statistically independent from the file being accessed. Adding that the security of secure computation shows that the simulated output is computationally indistinguishable from the real execution, we know that $\mathsf{H}_t \approx \mathsf{H}_{t+1}$.
- **Read or write a file.** We now discuss "read" and "write" together since these two operations are designed to be indistinguishable from each other (see Section 3.8 and Table 3.7) and have the same computation patterns. H_{t+1} replaces the simulated output with the real execution. To generate the simulated output, Sim invokes the simulator of secure computation and uses ran-

domly sampled paths in simulating the ORAM. Due to the security of Circuit ORAM and secure computation, the simulated output is computationally indistinguishable from the real execution, and therefore $\mathsf{H}_t \approx \mathsf{H}_{t+1}$.

- **Grant or revoke permission.** In Titanium, the access control matrix described in Section 3.7.1 is stored together with the file data. So, granting and revoking is similar to writing to a file, with the difference that the modification is done on the access control matrix. Therefore, when H_{t+1} replaces the simulated output with the real execution, for the same reasons, the simulated output is computationally indistinguishable from the real execution, and therefore $\mathsf{H}_t \approx \mathsf{H}_{t+1}$.

Now, by hybrid arguments, we know $\mathsf{H}_0 \approx \mathsf{H}_q$, which means that when processing q requests, the output of the ideal world and the output of the real world are computationally indistinguishable.

3.11.4 Prevent DoS attacks from users

In this paper, we do not consider denial-of-services (DoS) attacks by the users. Nevertheless, it remains an issue in practice. A user who exhausts the servers' resources by uploading or downloading a large amount of data would prevent other users from using the system. There have been many existing defenses in systems *without anonymity*. However, in Titanium, anonymity must be preserved.

Titanium can leverage the anonymous prevention mechanisms discussed in prior work, Ghostor [183] and Alpenhorn [239]: (1) anonymous payment and (2) Proof-of-Work (PoW).

- *Anonymous payment.* Titanium can require a user to pay for each data access. Specifically, each user deposits some money by doing a sender-anonymous payment to the servers, and the user then proves this payment in zero knowledge to the servers. The servers then issue a number of blind signatures [240] according to the paid amount, where one blind signature serves as a "one-time token" for one data access. The user needs to present an unused blind signature to the servers for each file access. Since the servers are "blinded", the user anonymity is preserved.

- *Proof-of-work.* Titanium can deter a malicious user by asking this user to solve a cryptographic puzzle [241–243]—also commonly known as proof of work—for each data access. This does not fully prevent DoS attacks and could be expensive for resource-constrained users, but it can limit the ability of the malicious attackers, and can be combined with other mechanisms.

Chapter 4

Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning

Many organizations need large amounts of high quality data for their applications, and one way to acquire such data is to combine datasets from multiple parties. Since these organizations often own sensitive data that cannot be shared in the clear with others due to policy regulation and business competition, there is increased interest in utilizing secure multi-party computation (MPC). MPC allows multiple parties to jointly compute a function without revealing their inputs to each other. We present Cerebro, an *end-to-end* collaborative learning platform that enables parties to compute learning tasks without sharing plaintext data. By taking an end-to-end approach to the system design, Cerebro allows multiple parties with complex economic relationships to safely collaborate on machine learning computation through the use of release policies and auditing, while also enabling users to achieve good performance without manually navigating the complex performance tradeoffs between MPC protocols.

4.1 Introduction

Recently, there has been increased interest in collaborative machine learning [244, 245], where multiple organizations run a training or a prediction task over data collectively owned by all of them. Collaboration is often advantageous for these organizations because it enables them to train models on larger datasets than what is available to any one organization, leading to higher quality models [246]. However, potential participants often own sensitive data that cannot be shared due to privacy concerns, regulatory policies [247, 248], and/or business competition. For example, many banks wish to detect money laundering by training models on customer transaction data, but they are unwilling to share plaintext customer data with each other because they are also business competitors.

Enabling these use cases requires more than a traditional centralized machine learning system, since such a platform will require a single trusted centralized party to see all of the other parties' plaintext data. Instead, the goal is to develop techniques through which participants can collabora-

System	**Multi-party**	**DSL & API**	**Policies**	**Automated optimization**	**Multiple backends**	**Auditing**
Specialized ML protocols	✓/ ✗	✗	✗	✗	✓/ ✗	✗
Generic MPC	✓	✗	✗	✗	✗	✗
MPC compilers	✓/ ✗	✓	✗	✓	✓/ ✗	✗
This paper: Cerebro	✓	✓	✓	✓	✓	✓

Table 4.1: Comparison with prior work in categories on properties necessary for collaborative learning. There are a number of works in specialized MPC protocols [71, 249–257], generic MPC [82, 188, 206, 207, 258], and MPC compilers [210, 217, 259–268]. Since the work space is so broad, we use "✓/ ✗" to indicate that only some systems in this category support that feature.

tively compute on their sensitive data *without revealing* this data to other participants. A promising approach is to use *secure multi-party computation* (MPC) [82, 207], a cryptographic technique that allows P parties to compute a function f on their private inputs $\{x_1, \ldots, x_P\}$ in such a way that the participants only learn $f(x_1, \ldots, x_P)$ and nothing else about each other's inputs.

While there is a vast amount of prior work on MPC for collaborative learning, none take *an end-to-end approach*, which is essential for addressing two major obstacles encountered by these organizations. The first obstacle is the tussle between generality and performance. Many recent papers on MPC for collaborative learning [71, 249–257] focus on hand-tuning MPC for specific learning tasks. While these protocols are highly optimized, this approach is not generalizable for a real world deployment because every new application would require extensive development by experts. On the other hand, there exist *generic* MPC protocols [82, 188, 206, 207, 258] that can execute arbitrary programs. However, there are many such protocols (most of which are further divided into sub-protocols [269, 270]), and choosing the right combination of tools as well as optimizations that result in an efficient secure execution is a difficult and daunting task for users without a deep understanding of MPC.

The second obstacle lies in the tussle between privacy and transparency. The platform needs to ensure that it addresses the organizations' incentives and constraints for participating in the collaborative learning process. Take the anti-money laundering use case as an example: while MPC guarantees that nothing other than the final model is revealed, this privacy property is also problematic because the banks effectively lose some control over the computation. They cannot observe the inputs or the computation's intermediate outputs before seeing the final result. In this case, some banks may worry that releasing a jointly trained model will not increase accuracy over their own models, but instead help their competitors. They may also have privacy concerns, such as whether the model itself contains too much information about their sensitive data [271–276] or whether the model is poisoned with backdoors [277].

In this paper, we present Cerebro, a platform for multi-party cryptographic collaborative learning using MPC. Cerebro's goal is to address the above two obstacles via a holistic design of an *end-to-end* learning platform, as illustrated in Figure 4.1.

To address the first challenge, Cerebro develops a compiler that can automatically compile any program written in our Python-like domain specific language (DSL) into an optimized MPC protocol. While there is prior work on MPC compilers [210, 217, 259–262, 264–268], none provides a *holistic* tool chain for the machine learning setting, which is the focus of our work. Beyond the compiler, the ML APIs provided by Cerebro abstract away the complexities of MPC from users, while keeping information needed for our compiler to do further optimizations. Our compiler also uses novel physical planning and considers the deployment environment to further choose the best MPC algorithms for executing machine learning workloads. We note here that Cerebro's goal is to provide a generic platform where users can write *arbitrary* learning programs, and not to compete with hand-optimized protocols (we compare Cerebro's performance against such protocols in Section 4.7.5).

We address the second challenge by introducing a set of mechanisms for organizations to ensure that their incentives and constraints are met before the result of a learning task is released, and also for participants to identify the source of malicious and ill-formed input data. Our insight is that we can leverage cryptographic primitives to enable this functionality without leaking additional data in the process. Based on this observation we define two important mechanisms: *compute policies* and *cryptographic auditing*. Compute policies allow parties to provide code that controls when and how the result of a learning task is released, while cryptographic auditing allows parties to backtrack and audit the inputs used during private computation, thus holding all parties accountable for their actions.

We implemented and evaluated Cerebro on common learning tasks—decision tree prediction, linear regression training, and logistic regression training. Our evaluation (Section 4.7) shows that our compiler generates optimized secure computation plans that are 1-2 orders of magnitude faster than an incorrect choice of a state-of-the-art generic MPC protocol (that is also un-optimized for machine learning), which is what a user might use without our system. Even with these performance gains, we want to remark that secure computation is not yet practical for all learning tasks. Nonetheless, we believe that a careful choice of protocols is practical for a number of useful learning tasks as our evaluation shows. Moreover, cryptographers have been improving MPC techniques at an impressive pace, and we believe that new MPC tools can be incorporated into the Cerebro compiler.

4.2 Background

Machine learning. Machine learning pipelines consist of two types of tasks: training and prediction. Training takes in a dataset and uses a training algorithm to produce a model. Prediction (or inference) takes in a model and a feature vector, and runs a prediction algorithm to make a prediction.

Secure multi-party computation (MPC). In MPC, P parties compute a function f over their private inputs $x_{i\in[1\dots P]}$, without revealing x_i to any other parties. In this paper, we consider that the final result is released in plaintext to every party.

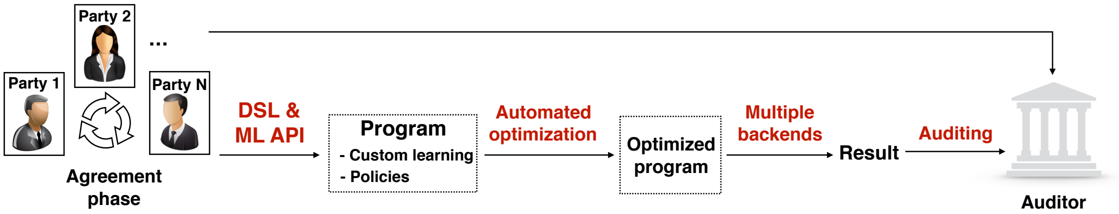


Figure 4.1: The Cerebro workflow.

There are two main MPC paradigms for generic computations: *arithmetic MPC* [188, 206, 207] and *boolean MPC* (based on garbled circuits). In arithmetic MPC, data is represented as finite field elements, and the main operations are addition and multiplication (called "gates"). In boolean MPC, data is represented as boolean values, and the main operations are XOR and AND.

One interesting commonality in these two frameworks is that they can often be split into two phases: preprocessing and online execution. At a high level, both frameworks use preprocessing to improve the online execution time for certain gates. In arithmetic circuits, addition gates can be computed locally without communication, while multiplication gates are more expensive to compute. Similarly, in boolean circuits, XOR is fast to compute while AND is much slower. The preprocessing phase for these frameworks pre-computes a majority part of executing multiplication/AND gates. And the preprocessing phase can execute without knowing the *input*; it only needs to know the *functionality*. The online execution for both arithmetic MPC and boolean MPC requires the parties to input their private data. At the end of this phase, the MPC protocol releases the output in plaintext to all the parties.

4.3 Overview of Cerebro

4.3.1 Threat model

We consider P parties who want to compute a learning function on their sensitive data. The parties are unwilling or unable to share the plaintext data with each other, but want to release the result of the function (e.g., a model or a prediction) according to some policies. We assume that the parties come together in an *agreement* phase during which they decide on the learning task to run, the results they want to disclose to each other, and the policies they want to implement. We assume this agreement is enforced by an external mechanism, e.g., through a legal agreement.

Cerebro allows the parties to choose what threat model applies to their use case by supporting both semi-honest and malicious settings. In the semi-honest setting, Cerebro can protect against an adversary who does not deviate from protocol execution. This adversary can compromise up to $P-1$ of the parties and analyze the data these parties receive in the computation, in hopes of learning more information about the honest party's data beyond the final result. In the malicious setting, the adversary can cause compromised participants to deviate from the protocol. The misbehavior includes altering the computation and using inconsistent inputs. Cerebro can support both settings by using different generic cryptographic backends. We believe that it is useful to support a flexible threat model because different organizations' use cases result in different assumptions about the adversary. Moreover, as we show in Section 4.7, the semi-honest protocol can be 61-$3300\times$ faster than the malicious counterpart, so the participants may not wish to sacrifice performance for malicious security.

Recent work has described many attacks for machine learning. One category is data poisoning [277] where the parties inject malicious data into the training process. Another category is attacks on the released result, where an attacker learns about the training dataset from the model [276, 278, 279] or steals model parameters from prediction results [272–275]. By definition, MPC does not protect against such attacks, and Cerebro similarly cannot make formal guarantees about maliciously constructed inputs or leakage from the result. However, we try to mitigate these issues via an end-to-end design of the system, where Cerebro provides a platform for users to program compute policies and add cryptographic auditing (explained in Section 4.5).

4.3.2 System workflow

Cerebro's pipeline consists of multiple components, as shown in Figure 4.1. In the rest of this section, we provide an overview of a user's workflow using Cerebro.

Agreement phase. This phase is executed before running Cerebro. During the agreement phase, potential participants come together and agree to participate in the computation. We assume that the number of participants is on the order of tens of parties. Parties need to agree on the computation (including the learning task and any compute policies) to run and agree on the threat model. Parties should also establish a public key infrastructure (PKI) to identify the participants.

Programming model. Users make use of Cerebro's Python-like domain-specific language (DSL) to write their programs. Users can easily express custom learning tasks as well as policies using our DSL and APIs. Cerebro also allows users to specify the configuration, such as the number of parties and how much data each party should contribute.

Compute policies. Cerebro supports user-defined compute policies via our DSL to handle concerns arising from the complex economic relationships among the parties. Compute policies can be generic logic for how results are obtained, or special *release policies* such that the result of a computation is only revealed if the policy conditions are satisfied.

Cryptographic compiler. Cerebro's cryptographic compiler can generate an efficient secure execution plan from a given program written in the Cerebro DSL. Our compiler first applies *logical optimization* directly on the program written in our DSL (see Section 4.4.2). Next, this optimized program is input to the *physical planning* stage (see Section 4.4.3) to generate an efficient physical execution plan.

Secure computation. In this phase, Cerebro executes the secure computation using the compiler's physical plan. When it finishes, the parties can jointly release the result.

```
# set_params() initializes parameters
# for an MPC execution, such as fixed-point
# parameters (p, f, k) and
# the number of parties (num_parties)
Params.set_params(p=64, f=32, k=64, num_parties=2)
# Decision tree prediction
# Reads in the tree model from party 0
tree = p_fix_mat.read_input(tree_size, 4, 0)
# Party 1 provides the features
x = p_fix_array.read_input(dim, 1)
...
for i in range(LEVELS-1):
    # Store user information in variables
    # like index and split
    ...
    cond = (x[index] < split)
    # This is a fused operation
    root = secret_index_if(cond, tree, left_child, right_child)
# Reveal prediction results
reveal_to_all(root[1], "Prediction")
```

Figure 4.2: A sample program written in Cerebro's DSL

Cryptographic auditing. Even after the result is released, the learning life cycle is not finished. Cerebro gives the parties the ability to audit each other's inputs with a third-party auditor in a post-processing phase (see Section 4.5.2).

4.4 Programming model and compiler

In this section we describe Cerebro's programming model. Similar to prior work [210, 217, 259–262, 264–268], and based on the DSL implementation by SCALE-MAMBA (see Section 4.6), users specify programs that Cerebro can execute using a domain-specific language (Section 4.4.1), which is then used as input to the Cerebro compiler (Figure 4.3).

The Cerebro compiler implements two logical optimization passes, which *reduce the amount of computation expressed in MPC* while preserving security guarantees. Finally, the Cerebro physical planner (Section 4.4.3) takes the logical plan generated by the compiler, and uses information about the physical deployment to instantiate and execute the plan.

4.4.1 Cerebro DSL

In Cerebro, users express training and inference algorithms, compute policies, and auditing functions using a Python-like domain specific language (DSL). Our DSL supports a variety of numerical data types that are commonly used in machine learning, data analytics, and generic

Input 1	Input 2	Output	Compute
public	public	public	local at all
public	private(i)	private(i)	local at i
private(i)	private(i)	private(i)	local at i
private(i)	private(j)	secret	global
any	secret	secret	global

Table 4.2: Rules for defining a function's execution mode

functions and are useful for expressing training and inference algorithms. Figure 4.2 shows an example program.

Data types. Each variable in a Cerebro program is automatically tagged with a type (integer, fixed-point, etc.) and a *security level*. The security level indicates which parties can access the raw value of the variable. Cerebro currently supports three security levels:

- *Public*: the value is visible to all parties
- *Private*: the value is visible to a single party
- *Secret*: the value is hidden from all parties

Our current implementation restricts that private variables are owned and visible to a single party, and we represent a private value visible to the party i as `private(i)`. The security level of variables is automatically upgraded based on type inference rules, described in Section 4.4.2. Programs can explicitly downgrade security levels by calling `reveal`.

Functions. Our DSL provides a set of mathematical and logical operators to process tagged data. Each operator can accept inputs with any security tag, and the output tag is determined using a set of type inference rules (explained more in Section 4.4.2). Security annotations also play an important role in enabling several of the optimizations employed by Cerebro.

Cerebro provides a variety of basic operators over data types including arithmetic operations and comparisons. Users can compose these basic operators to implement user-defined learning algorithms. Cerebro also provides a set of higher-level mathematical operators common to machine learning tasks (e.g., linear algebra operators, sigmoid), a set of functions for efficiently indexing into arrays or matrices, a set of branching operators, and a set of more complex fused operators. Fused operators (explained in Section 4.4.2) provide Cerebro with more opportunities to optimize complex code patterns.

4.4.2 Logical optimization

Given a program written in the Cerebro DSL, the Cerebro compiler is responsible for generating a logical execution plan that minimizes runtime. Our programming model also allows us to easily apply logical optimizations. More specifically, Cerebro includes two new optimizations that are particularly useful for machine learning tasks: the first is *program splitting*, where a program Q is split into two portions Q_1 and Q_2 such that Q_1 can be executed in plaintext, while Q_2 is executed using secure computation. The second optimization is operator fusion, where the compiler tries to detect pre-defined compound code patterns in Q_2 and transforms them to more efficient fused operations.

Program splitting. Program splitting is a type of logical optimization that delegates part of the secure computation to one party which computes *locally in plaintext*. We can illustrate this optimization by applying to sorting. If a program needs to sort training samples from all parties (e.g., in decision tree training), then parties can instead pre-sort their data. In this way, MPC only needs to merge pre-sorted data, providing a significant speedup over the naive solution in which it executes the entire sorting algorithm in the secure computation.

In the semi-honest setting, Cerebro can *automatically identify* opportunities for local computation within the code. As explained in Section 4.4, users write their programs using Cerebro's API, and the compiler automatically tags their data using Cerebro's secure types. Cerebro uses a set of rules (see Table 4.2) to infer a function's security level. If a function only has public input, then the output should also be public since it can be inferred from inputs. This type of computation can be executed in plaintext by any party. Similarly, if a function only takes input from a single party i, party i can compute this function locally in plaintext. However, if a function's input includes private data from different parties or secret data, then the function needs to be executed using MPC, and the output will also be tagged as secret.

However, in a malicious setting the criteria for secure local plaintext execution are more complex because a compromised participant can arbitrarily deviate from the protocol and substitute inconsistent/false data and/or compute a different function. Thus, we cannot assume that a party will compute correct values locally. As in the sorting example, we cannot trust the parties to correctly pre-sort their inputs. Therefore, secure computation must add an extra step to ensure that the input from each party is sorted.

In general, automatically finding efficient opportunities for local plaintext computation in the malicious setting is challenging. In Cerebro, we approach this problem by designing pre-defined APIs with this optimization in mind. If a user uses our API, Cerebro will apply program splitting appropriately while guaranteeing security in the malicious threat model. For example, our `sort` API will automatically group the inputs into private inputs from each party, followed by a local plaintext sorting in plaintext at each party. However, since a malicious party can still try to input unsorted data into the secure computation, the global sorting function will first check that the inputs from each party are sorted.

This optimization allows Cerebro to automatically generate an efficient MPC protocol that has similar benefits to prior *specialized* work. For example, in [255], one of the techniques is to have the parties pre-compute the covariance matrix locally, then sum up these matrices using linearly homomorphic encryption. While Cerebro's underlying cryptography is quite different—hence resulting in a very different overall protocol—we are able to automatically discover the same local computation splitting as is used by a specialized system written for ridge regression. We note that program splitting is compatible with cryptographic auditing mentioned in Section 4.5.2 by committing to the precomputed local data instead of the original input data.

Fused operations. Recognizing compound code patterns is crucial in MPC, since many com-

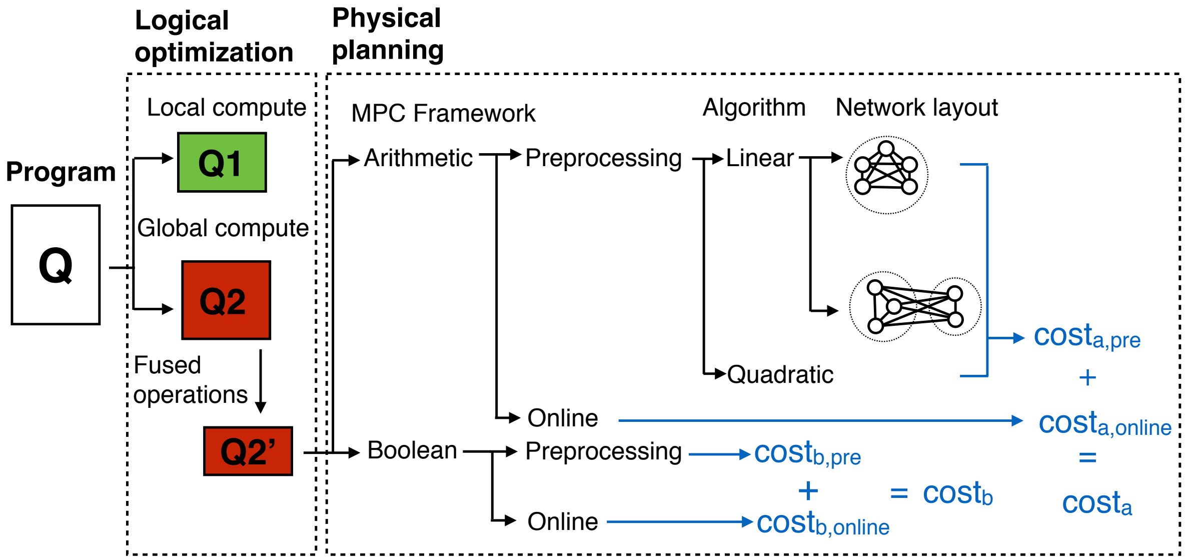


Figure 4.3: Cerebro architecture, showing choices we can make under the semi-honest threat model.

pound operations that are cheap in plaintext incur significant performance penalties when executed securely. For example, plaintext array indexing under the RAM model has a constant cost. In MPC, while array indexing using a public index has a constant cost, array indexing using a *secret variable* takes time that is proportional to the length of the array. This is because when executing secure computation, the structure of the function cannot depend on any private or secret value, otherwise a party may infer the value from the structure of the computation. Therefore, it is impossible to index an array using a secret value in constant time.

In Cerebro, as is common, we index arrays by linearly scanning the entire array, which is an $O(n)$ operation.[1] Next, consider a compound code pattern that occurs in programs like decision tree prediction (see Figure 4.2): an if/else statement that wraps around multiple secret accesses to the same array. In a circuit-based MPC, all branches of an if/else statement need to be executed. Therefore, conditionally accessing an index can require several scans through the same array.

For this scenario, Cerebro will combine the operators into a single fused operation represented by `secret_index_if` that can be used to represent such conditional access and minimizes the number of array scans required during computation. Fused operators in Cerebro play the same role as level 2 and level 3 [280] operations in BLAS [281] and MKL [282], and fused operations generated by systems such as Weld [283]; i.e., they provide optimized implementations of frequently recurring complex code patterns. Since operator fusion only happens on code expressed in MPC and preserves the functionality, it works for both the semi-honest and the malicious settings.

[1] Cerebro can be augmented to use oblivious RAM (ORAM) for secret indexing, which has $O(polylogn)$ overhead for an array of size n. Prior work has shown that for smaller arrays, linear scanning is faster [100] because ORAM needs to keep a non-trivial amount of state.

4.4.3 Physical planning

Once a logical plan has been generated, Cerebro determines an efficient physical instantiation of the computation, which can then be executed using one of Cerebro's MPC backends. We call this step physical planning (illustrated in Figure 4.3 on the right side) and describe it in this section. When converting logical plans into physical implementations, Cerebro must decide whether to use operations provided by existing boolean and arithmetic MPC protocols or to use our special vectorized primitives (Section 4.4.3). To choose between these implementation options, Cerebro uses a set of cost models (Section 4.4.3) to predict the performance of different implementation choices and picks the best among these choices. Finally, once a physical implementation has been selected, Cerebro decides where to place (Section 4.4.3) computation among available nodes—this choice can significantly impact performance in the wide area network.

Notation. Let P denote the number of parties, and let $\mathcal{P}_i$ denote the i-th party. We use N to represent the total number of gates in a circuit. N_m is the number of **m**ultiplication gates in an arithmetic circuit; N_a is the number of **A**ND gates in a boolean circuit. $B_{(\cdot)}$ represents network bandwidth parameters and $l_{(\cdot)}$ represents latency parameters. For a given type of encryption algorithm $C_{(\cdot)}$, we use $|C_{(\cdot)}|$ to represent the number of bytes in a single ciphertext.

We use c to capture any constant cost in a cost model, like an initialization cost. The rest of the cost can be categorized as *compute* (represented using f_i functions) and *network* costs (represented using g_i functions).

Vectorization. Cerebro supports compilation to two main MPC backends: arithmetic [188, 206] and boolean [258]. Both backends consist of two phases: preprocessing and online. During the preprocessing phase, random elements are computed and can be used later during the online phase. Preprocessing is especially useful because it can be executed before the parties' private inputs are available.

In arithmetic MPC preprocessing, parties need to compute multiplication triples, which are used to speed up multiplication operations during the online phase. However, many common machine learning tasks contain matrix multiplication, which is especially costly because of the large number of multiplication operations. In this section, we describe an optimization for arithmetic MPC preprocessing that allows us to vectorize multiplication triple generation. This idea was introduced in prior work for the semi-honest two party setting [71], and here we generalize the algorithm to the n-party semi-honest setting.

The two-party vectorized protocol happens in the preprocessing phase where it computes random matrix multiplication triples such that each $\mathcal{P}_i$ holds $A_j^{(i)}, B_j^{(i)}, C_j^{(i)}$ where $\sum_i (A_j^{(i)} \cdot B_j^{(i)}) = \sum_i C_j^{(i)}$. For the sake of a simpler analysis, we assume that B_j is a vector $\vec{b}$, and that the relation is $\vec{c} = A\vec{b}$. To generalize this to the multi-party setting, we can apply the two-party protocol in a pairwise fashion to generate the triples. To compute the triple, it suffices for each party to first sample random $A^{(i)}$ and $\vec{b}^{(i)}$, then use the two-party protocol to compute the pairwise products $A^{(i)} \cdot \vec{b}^{(j)}$.

Cost models. In this section, we provide two examples of the different cost models in Cerebro (see Section 4.9.1 for more).

• **Preprocessing planning.** As previously stated, Cerebro's MPC backends consist of preprocessing and online phases. Semi-honest arithmetic MPC has two different preprocessing protocols: *linear preprocessing* and *quadratic preprocessing* [188, 206, 270]. We describe the high-level protocols in Section 4.9.2. These two methods can behave quite differently under different setups, and we illustrate this by presenting their cost models. We define C_l to be the encryption algorithm used in linear preprocessing, and C_q to be encryption algorithm used in quadratic preprocessing. The per-party cost model for linear preprocessing is given by:

$$c + N_m(f_1(|C_l|) + \frac{1}{P}[f_2(|C_l|)(P-1) + f_3(|C_l|) + g(B, |C_l|)(P-1)]) \tag{4.1}$$

The per-party cost model for quadratic preprocessing is:

$$c + N_m(P-1)(f(|C_q|) + g(B, |C_q|)) \tag{4.2}$$

In terms of the scaling in the number of parties, linear preprocessing is much better than quadratic preprocessing. However, since $|C_q| < |C_l|$, quadratic preprocessing's encryption algorithm uses less computation and consumes less bandwidth.

• **Cost of vectorization.** The cost model to preprocess a matrix-vector multiplication for $(m, n) \times (n, 1)$ is:

$$c + f_1(|C_q|)(n + m(P-1)) + f_2(|C_q|)m(P-1) + g(B, |C_q|)(m+n)(P-1) \tag{4.3}$$

Comparing this cost model to Equation (4.2) (where we replace N_m with mn), the triple generation load is reduced from mn to m or $m + n$. We note that vectorization not only speeds up triple generation, but also introduces another planning opportunity if a program has a mix of matrix multiplication and regular multiplication.

Layout optimization. In the wide area network setting, different physical layouts can significantly impact the performance of a protocol. In this section, we give an example of *layout optimization*, where Cerebro plans an alternative communication pattern for parties that span multiple regions.

In the semi-honest setting, linear preprocessing requires a set of coordinators that aggregate data from all parties. The coordinators can be trivially load-balanced among all parties by evenly distributing the workload. However, this only works when the pairwise communication costs are similar, and no longer works when the parties are located in different regions.

We make the observation that the underlying algorithm requires coordinators to perform an aggregation operation. Therefore, we introduce *two-level hierarchical layout*, where the coordination happens at both the intra-region and the inter-region levels. Each triple is still assigned to a single global coordinator, and is also additionally assigned a *regional coordinator* that is in charge of partially aggregating every party's data from a single region and sending the result to the global coordinator.

Assumptions. We assume that the regions are defined by network bandwidth. The regions can be manually determined based on location, or automatically identified by measuring pairwise bandwidth and running a clustering algorithm. For a more detailed analysis (including a walkthrough for the case of two parties), see Section 4.9.3.

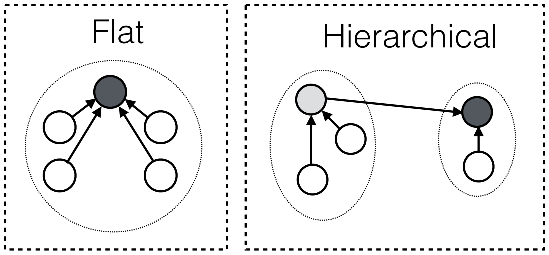


Figure 4.4: Communication pattern for a single multiplication triple. Shaded nodes are coordinators.

Given k regions, let B_{ij} denote the bandwidth between regions i and j and let B_i denote the bandwidth within region i. Let n_i denote the number of triples assigned to each party in region i and $\mathcal{P}_i$ be the number of parties in region i. There, we have $\sum_{i=1}^{k} n_i \cdot P_i = N_m$. The cost function can now be formulated as $C = L'_1 + L'_2 + L'_3$ where the constants are analogous to those in the previous example of two regions. We generalize the constants as follows:

$$\begin{aligned} L'_1 &= \max\left(\textstyle\sum_{j\neq i}\left(\frac{n_j\cdot P_j(P_i-1)}{P_iB_i}\right)\right) && i=1,2,...,k,\\ L'_2 &= \max\left(\frac{n_i\cdot(P_i-1)}{B_i}\right) && i=1,2,...,k,\\ L'_3 &= \max\left(\textstyle\sum_{j\neq i}\left(\frac{n_j\cdot P_j}{B_{ij}}\right)\right) && i=1,2,...,k \end{aligned}$$

We solve this optimization problem in cvxpy [284] by transforming it into a linear program, described in Section 4.9.3. As an example, a setting with five regions is solved in roughly 100 milliseconds on a standard laptop computer.

4.5 Policies and auditing

In the collaborative learning setting, an end-to-end platform needs to take into account the incentives and constraints of the participants. This is critical when competing parties want to cooperate to train a model together. For example, the participants may be concerned about each other's behavior during training, as well as the costs and benefits of releasing the final model to other parties. A party may want to make sure that the economic benefits accrued by its competitors do not greatly outweigh its own benefits. Thus, a collaborative learning platform needs to allow participants to specify their incentives and constraints and also needs to ensure that both are met.

Cerebro addresses this problem by introducing the notion of user-defined *compute policies* and a framework for enabling *cryptographic auditing*. Compute policies are executed as part of the secure computation and are useful for integrating extra pre-computation and post-computation checks before the result is released. Auditing is executed at a later time after the result is released and can make parties accountable for their inputs to the original secure computation. In the rest

```
def release_policy(prediction_fn, test_data, weights, tau):
    score = prediction_fn(data, weights)
    return (score > tau)
# Make a call to release_policy
if_release = release_policy(lr_prediction, vdata, weights, min_score)
# Set weights to 0 if if_release is false
final_weights = release(if_release, weights)
return final_weights
```

Figure 4.5: Example validation-based release policy.

of this section, we give an overview of how users can use our system to encode policies and audit cryptographically.

4.5.1 Compute policies

We first make the observation that secure computation can enable *user-defined compute policies* that can be used to dictate how the result of a computation is released. In fact, MPC's security guarantees means that it can also be used to *conditionally release the computation result*. Cerebro provides an easy way for users to write an arbitrary release policy by first writing as a function that returns a boolean value `if_release` and calls our `release` API on this boolean value and the result of the learning task. If `if_release` is true, then `release` will return the real result; otherwise it will return 0 values, thus un-releasing the result. Figure 4.5 shows an example policy written in Cerebro.

We assume that policy *functions* are public, and that all participants must agree on them during the agreement phase. This workflow allows participants to verify that each other's policy conforms to some constraints before choosing to input private data and dedicate resources for the secure computation. However, the constants/inputs for these policies can be kept private using MPC (e.g., a training accuracy threshold).

Since our DSL is generic, the participants can program any type of policy. We focus on two major categories of policies—validation-based policies and privacy policies—and how they can be encoded in our DSL.

Validation-based policy. In training, model accuracy can be a good metric of economic gains/losses experienced by a participant since it is usually the objective that a party seeks to improve via collaborative learning. In a single-party environment, the metric is commonly computed by measuring the prediction accuracy on the trained model using a held-back dataset. When constructing validation-based policies in Cerebro, each party provides a test dataset in addition to their training dataset and provides a prediction function. We now describe some examples.

- **Threshold-based validation.** In this policy, party i wants to ensure that collaborative training gives better accuracy than what it can obtain from its local model. The policy takes in the model w, a test dataset $X_{t,i}$, as well as a minimum accuracy threshold τ_i. This policy runs prediction on

$X_{t,i}$ and obtains an accuracy score. If this score is greater than τ_i, then the policy returns true. See example code in Figure 4.5.

• **Accuracy comparison with other parties.** In this policy, party i's decision to release depends on how much its competitors' test accuracy scores improve. Therefore, the inputs to this policy are: the model w, every party's test dataset $X_{t,j}$, every party's local accuracy scores a_j, and a percentage x. The policy runs prediction on every party's test dataset and obtains accuracy scores b_j. Then it checks b_j against a_j, and will only return true if $b_j - a_j < x(b_i - a_i)$ for all $j \neq i$.

Cross validation. Since the parties cannot see each other's training data, it is difficult to know whether a party has contributed enough to the training process. All parties may agree to implement a policy such that if a party does not contribute enough to training, then it also does not receive the final model. Such a party can be found by running *cross validation*, a common statistical technique for assessing model quality. In this setting, Cerebro treats the different parties as different partitions of the overall training dataset and takes out a different party every round. The training is executed on the leftover $P - 1$ parties' data, and an accuracy is obtained using everyone's test data. At the end of P rounds, the policy can find the round that results in the highest test accuracy. The party that is *not* included in this round is identified as a party that contributed the least to collaborative training.

Privacy policy. For training tasks, the secure computation needs to compute and release the model in plaintext to the appropriate participants. Since the model is trained on everyone's private input, it must also embed *some* information about this private input. Recent attacks [276] have shown that it is possible to infer information about the training data from the model itself. Even when parties do not actively misbehave (applicable in the semi-honest setting), it is still possible to have *unintended leakage* embedded in the model. Therefore, parties may wish to include *privacy checks* to ensure that the final model is not embedding too much information about the training dataset. We list some possible example policies that can be used to prevent leakage from the model.

• **Differential privacy.** Differential privacy [285] is a common technique for providing some privacy guarantees in the scenario where a result has to be released to a semi-trusted party. There are differential privacy techniques [286–288] for machine learning training, where some amount of noise is added to the model before release. For example, one method requires sampling from a public distribution and adding this noise directly to the weights. This can be implemented in Cerebro by implementing the appropriate sampling algorithm and adding the noise to the model before releasing it.

• **Model memorization.** Another possible method for dealing with leakage is to measure the amount of training data memorization that may have occurred in a model. One particular method [271] proposes injecting some randomness into the training dataset and measuring how much this randomness is reflected in the final model. This technique can be implemented by altering the training dataset X_i and programming the measurement function as a release policy.

4.5.2 Cryptographic auditing

In the malicious setting, Cerebro can use a maliciously secure MPC protocol to protect against deviations during the compute phase. However, even such an MPC protocol cannot protect against any attack that happens *before* the computation begins; namely, an adversarial party can inject carefully crafted malicious input into the secure computation in order to launch an attack on the computed result.

For example, prior work has shown that a party can inject malicious training data that causes the released model to provide incorrect prediction results for any input with an embedded backdoor [277]. If multiple self-driving car companies wish to collaboratively train a model for better object detection, a malicious participant can embed a specific backdoor pattern into non-malicious training samples and also change the corresponding prediction labels. If there are enough poisoned training samples, then the trained model will associate the backdoor pattern with a specific prediction label. If this poisoned model is deployed in a real world application by the victim in their self driving cars, the same adversary can attack the poisoned model by embedding the backdoor pattern—perhaps detecting a stop sign as a speed limit sign—thus triggering a malicious behavior that could cause a crash.

Overview. The previously proposed compute policies may be insufficient to detect such attacks since either the policy writer has to be aware of the chosen backdoor—which is unlikely—or the policies have to exhaustively check the input domain—which is infeasible. Therefore, we propose an auditing framework that instead aims to hold all parties accountable for their original inputs even after the result has been released. Auditing allows parties to execute an *auditing function* on the same inputs that were used during the compute phase—Cerebro guarantees that no party can maliciously substitute an alternative sanitized input during auditing without being detected as cheating. Using the previous attack as an example: if a poisoned model is triggered during inference, the victim can request an auditing phase. During the auditing phase, all parties must first agree on a public auditing function, then undergo an audit on their input training data. If the auditing function is correctly constructed, the auditing phase should be able to identify the parties that input the malicious training samples.

We note that Cerebro's aim is to provide a *framework* for auditing instead of specific auditing functions. Therefore, we rely on the participants to formulate auditing functions for specific attacks that they wish to protect against. In the above example, the self-driving car companies will need to design an auditing function that finds similarities in the malicious samples that trigger a misprediction and the training samples from each party. If an auditing function is not correctly formulated, then the auditing process cannot detect wrongdoing. The goal of auditing is to ensure that either the auditing function is successfully executed to completion, or the participant who causes an abort during auditing is identified (addressed in more detail in Section 4.5.2).

Finally, the type of threat that Cerebro is attempting to address is one where the result of the computation is attacked by constructing malicious input to the computation. Consequently, we assume that the attacker wants to get the result of the computation, and therefore do not address aborts during the compute time.

Auditing framework design. When auditing a computation in Cerebro, we need to ensure that the audit procedure has access to the same inputs as were used in the original computation. Otherwise, we run the risk of allowing a malicious participant to provide sanitized inputs during the audit, thus avoiding detection. Cerebro enforces that the same input from the compute phase is used in the auditing phase as well by using *cryptographic commitments* [289, 290], a cryptographic tool that ties a user to their input values without revealing the actual input. A participant commits to its input data by producing a randomized value that has two properties: *binding* and *hiding*. Informally, binding means that a party who produces a commitment from its malicious dataset will not be able to produce an alternate sanitized version later and claim that the commitment matches this new dataset. At the same time, hiding ensures that the commitments do not reveal information about the inputs.

• **Auditing API.** In order to abstract away the cryptographic complexity and to provide users with an intuitive workflow, we design the following API:

- `c, m = commit(X)`: returns `c`, the actual commitment, as well as `m`, the metadata used in generation of the commitment. `c` is automatically published to every other party, while `m` is a private output to the owner of `X`.
- `audit(X, c, m)`: this function returns a boolean value showing whether the commitment matches input data `X`.

• **Handling malicious aborts.** A serious concern during auditing is that a participant might cause the secure computation to abort since maliciously secure MPC generally does not protect against parties aborting computation. There are two types of aborts: a malicious party can refuse to proceed with the computation or can maliciously alter its input to MPC so that the computation will fail. The first type of abort is easy to catch, but the second is sometimes impossible to detect. For example, an arithmetic MPC that uses information theoretic MACs to check for protocol correctness cannot distinguish which party incorrectly triggered a MAC check failure. Therefore, a party can maliciously fail during the auditing phase and make it impossible to run an auditing function to track accountability.

To resolve this challenge, we introduce a *third-party auditor* into our auditing workflow. We do not believe this is an onerous requirement, since audit processes often already involve third-party arbitrators, e.g., courts, who help decide when to audit and how to use audit results. We *do not* require the third-party to be completely honest, but instead assume that it is honest-but-curious, does not collude with any of the participants, and does not try to abort the computation. Under this assumption, we enable the auditor to audit a party without forcing the party to release its data. This means that the auditor will not see any party's data in plaintext, since we still require the auditor to run the auditing process using MPC. During auditing, we require all parties to be online, and any party who is not online or aborts is identified as malicious.

• **Auditing workflow.** Let $\mathcal{A}$ denote a separate auditor entity, and let $\mathcal{P}_i$ denote the parties running the collaborative computation. We construct the following auditing protocol.

1. Using the established PKI, $\mathcal{P}_i$'s have public keys corresponding to every participant in the secure computation. $\mathcal{P}_i$'s agree on the same unique number qid.
2. $\mathcal{P}_i$ computes a commitment of its data. Let the commitment be $\vec{c}_i$. $\mathcal{P}_i$ hashes the commitments $h_i = \mathsf{hash}(\vec{c}_i)$ and generates a signature $\sigma_i = \mathsf{sign}(\mathsf{qid}, h_i)$ using its secret key. $\mathcal{P}_i$ publishes

$(\vec{c}_i, \sigma_i)$ to $P_{j \neq i}$.

3. All $\mathcal{P}_i$'s run the secure computation, which encodes the original learning task and a preprocessing stage that checks that $\mathcal{P}_i$'s input data indeed commits to the public commitments received by every party from $\mathcal{P}_i$. If the check fails, then the computation aborts. Note that we won't know who is cheating in this stage, but the parties also won't get any result since the computation will abort before any part of the learning task is executed.
4. During auditing, $\mathcal{P}_i$ will publish its signed commitments, along with the $(\vec{c}_j, \sigma_j)$ received from $\mathcal{P}_j$, to $\mathcal{A}$. $\mathcal{A}$ checks that all commitments received from $\mathcal{P}_j$ about $\mathcal{P}_i$ match. If they do not, then $\mathcal{P}_i$ is detected as malicious.
5. $\mathcal{A}$ runs a two-party secure computation with each $\mathcal{P}_i$ separately. $\mathcal{P}_i$ inputs its data, and $\mathcal{A}$ checks the data against the corresponding commitment. If there is a match, continue with the auditing function. If this computation aborts, $\mathcal{P}_i$ is also detected as malicious. Since the auditing is in secure computation, $\mathcal{A}$ will not directly see $\mathcal{P}_i$'s input data.

Using the same training example from above, we can see that any $\mathcal{P}_i$ who cheats by substituting input can only avoid detection via a badly formulated auditing function. A cheating party will be detected and identified by the auditor if it attempts to substitute an alternative copy of the input or if it attempts to abort during auditing.

Commitment schemes. Our auditing protocol is generic enough to be implemented with any commitment and MPC design. In practice, there are ways of constructing efficient commitments that can also be easily verified in MPC. In this section, we describe some commitment schemes that integrate well with MPC, and how to efficiently check these commitments.

• **SIS-based commitment.** Based on the short integer solution (SIS) problem in lattices, there is a class of collision-resistant hash functions [291–296], from which we can instantiate commitment schemes that are efficient in MPC. This has been used in zero-knowledge proof systems [297, 298].

• **Pedersen commitment.** In this section, we additionally provide a way of batch checking commitments in MPC using a homomorphic commitment such as Pedersen [290]. We utilize the fact that our arithmetic framework is reactive to construct such a scheme.

Denote $\mathsf{com}(x; r)$ as the Pedersen commitment. The protocol is as follows:

1. As before, each $\mathcal{P}_i$ commits and publishes its commitments.
2. $\mathcal{P}_i$'s start a SPDZ computation and input both their input data $\vec{x}_i$, as well as the randomness used $\vec{r}_i$ for the commitments.
3. Everyone releases a random number s from SPDZ.
4. Each $\mathcal{P}_i$ computes $\tilde{c}_j = \sum_k s^k \otimes \vec{c}_j[k]$ for every $\mathcal{P}_j$.
5. $\mathcal{P}_i$'s input s as well as $\tilde{c}_j$ into the same SPDZ computation computed in step 2.
6. The secure computation calculates $\tilde{x}_i = \sum_k s^k \cdot \vec{x}_i[k]$ and $\tilde{r}_i = \sum_k s^k \cdot \vec{r}_i[k]$. Then it checks that $\mathsf{com}(\tilde{x}_i; \tilde{r}_i) = \tilde{c}_i$.

For elliptic curve groups, the prime modulus will need to be on the order of at least 256 bits.

Tradeoffs. While SIS-based commitments work with our standard benchmarking prime field of 170 bits, Pedersen commitments need a minimum prime field size of 256 bits. Thus, while Pedersen commitments are more efficient because they enable triple batching, the larger bit size means offline generation can be more expensive. Of course, if the application already needs a larger field size (e.g., more precision for fixed-point representation), then Pedersen commitments would not have extra overhead. Additionally, Pedersen commitments require a reactive framework such as SPDZ in order for the batching to work properly in the secure computation phase. Cerebro's planner takes these circumstances into account, and chooses the best plan accordingly.

4.6 Implementation

We implemented Cerebro's compiler on top of SCALE-MAMBA [217], an open-source framework for arithmetic MPC. Our DSL is inspired by and quite similar to that of SCALE-MAMBA, though we have the notion of private types. In order to support both arithmetic and boolean MPC, we added a boolean circuit generator based on EMP-toolkit [259]. Both of these circuit generators are plugged into our DSL so that a user can write one program that can be compiled into different secure computation representations.

Cerebro uses different cryptographic backends that support both semi-honest and malicious security. We implemented Cerebro's malicious cryptographic backend by using the two existing state-of-the-art malicious frameworks—SPDZ [217] and AG-MPC [259]. Additionally, we implemented Cerebro's semi-honest cryptographic backend by modifying the two backends to support semi-honest security.

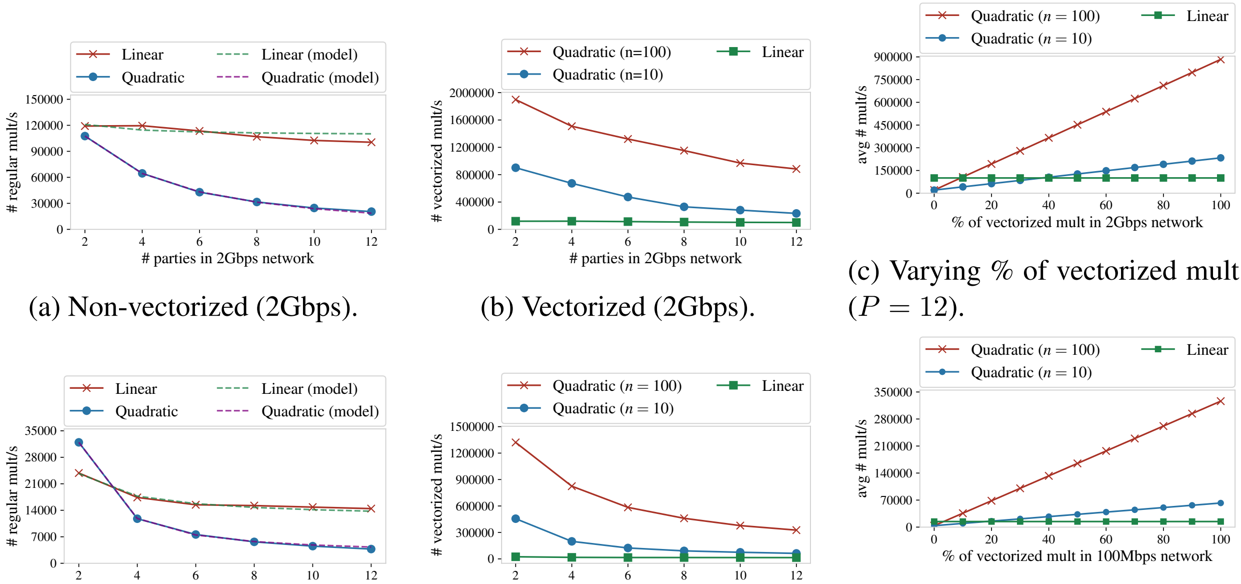
4.7 Evaluation

We evaluate the effectiveness of Cerebro's cryptographic compiler in terms of the performance gained using our techniques. We use the two generic secure multiparty frameworks that Cerebro uses as a baseline for evaluation, in both semi-honest and malicious settings. We compare to what users would be doing today without our system, which is choosing a generic MPC framework and implementing a learning task using it. Our goal is to show that, without Cerebro's compiler, users can experience *orders of magnitude* worse performance if they choose the wrong framework and/or do not have our optimizations.

We also do not experimentally compare performance against a traditional centralized machine learning system. Such a system can greatly outperform an MPC-based system because it can operate directly on the parties' plaintext training data, but is also *insecure* under our definition because it requires a centralized party that sees all of the plaintext training data. Due to the lack of security, the applications we are tackling cannot be realized with a centralized learning system.

4.7.1 Evaluation setup

Our experiments were run on EC2 using r4.8xlarge instances. Each instance has 32 virtual CPUs and 244GB of memory. In order to benchmark in a controlled environment, we use `tc` and `ifb` to fix network conditions. Unless stated otherwise, we limit each instance to 2Gbps of upload bandwidth and 2Gbps of download bandwidth. We also adjust latency so the round trip time



(a) Non-vectorized (2Gbps).

(b) Vectorized (2Gbps).

(c) Varying % of vectorized mult. ($P = 12$).

(d) Non-vectorized (100Mbps).

(e) Vectorized (100Mbps).

(f) Varying % of vectorized mult. ($P = 12$).

Figure 4.6: Choosing linear vs. quadratic protocol for arithmetic MPC preprocessing; y-axis shows triple generation throughput.

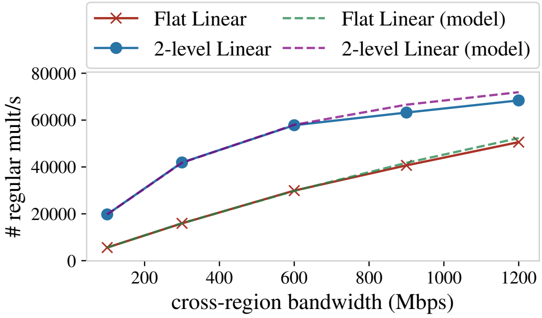


Figure 4.7: Flat vs. two-level linear protocol for 9-party vs. 3-party bipartite network layout with varied cross-region total bandwidth (2Gbps intra-region per-party bandwidth).

(RTT) is 80ms between any two instances. According to [299], this is roughly the RTT between the east-coast servers and west-coast servers of EC2 in the U.S.

4.7.2 Compiler evaluation

We evaluate Cerebro's compiler by answering these questions:

1. Are Cerebro's cost models accurate?
2. How do logical optimizations impact performance?
3. For realistic setups, does Cerebro's physical planning improve performance?

To answer these questions, we run a series of microbenchmarks as well as end-to-end application-level benchmarks. We first curve-fit our cost models and extrapolate against experimental results. We then evaluate different planning points to show Cerebro's gain in performance. We focus our evaluation on planning in the semi-honest setting, but our planning also supports the malicious setting, though a number of optimizations would be unavailable.

Microbenchmarks.

• **Cost models.** Our first microbenchmark compares the two methods for semi-honest arithmetic MPC preprocessing (see Section 4.4.3): linear and quadratic preprocessing.

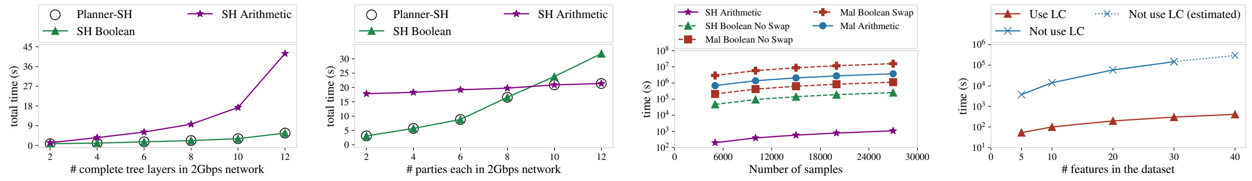
For both of the following experiments, we fit the constants of our cost model to the first four points of the graph and then extrapolate the results for the remaining two points. The dotted lines of the graph indicate the cost model's predictions and we can see that it closely matches with the experimental results. Figure 4.6a shows the preprocessing throughput of the linear and the quadratic protocols on high-bandwidth network. When the number of parties is small, the two protocols have similar throughput. However, as the number of parties increases, the quadratic protocol becomes slower than the linear protocol, mainly due to the increased communication.

Figure 4.6d compares the same protocols when the network is slow and becomes the bottleneck. When the number of parties is small, the quadratic protocol is faster than the linear protocol because it uses smaller ciphertexts, but it performs worse than the linear protocol as the number of parties increases.

• **Vectorization.** Figures 4.6b and 4.6e show the preprocessing throughput of a single matrix-vector multiplication—where the matrices are of sizes $(m \times n)$ and $(n \times 1)$—under different network conditions. We test with a fixed $m = 128$ and vary n in our experiments. On a high-bandwidth network, when there are two parties and $n = 100$, the quadratic protocol achieves a $16\times$ speedup over the linear protocol. Even when the number of parties is increased to 12, these two protocols still have an $8.8\times$ gap. On a slower network, the matrix-vector technique has a larger performance gain since it mainly saves communication, with up to a $55\times$ speed up.

Next, we evaluate the two protocols when there is a mix of matrix multiplication and regular multiplication. The results are shown in Figures 4.6c and 4.6f. The planning decision will be different based on the percentage of multiplication gates that can be substituted with matrix-vector multiplications, the shape of such matrices, the number of parties, and the network bandwidth. For example, in 2Gbps network with 12 parties and $n = 10$, if 40% of the multiplication gates can be vectorized, then Cerebro will pick quadratic. If the network bandwidth drops to 100Mbps, then 20% of such computation is enough for the compiler to pick quadratic.

• **Layout planning.** We evaluate the hierarchical layout preprocessing against a flat one for 12 parties across two regions: 9 are located in one region, and 3 are located in the other. Each party has 2Gbps bandwidth for intra-region communication, and we vary the *total* cross-region bandwidth shared by parties in the same region. Figure 4.7 shows the throughput comparison as well as our fitted cost models. Similar to before, we fit the constants of our cost model to the first



(a) Decision tree prediction varying tree sizes.

(b) Decision tree prediction varying number of parties.

(c) Logistic regression training in different dataset size.

(d) ADMM with/without local compute.

Figure 4.8: Experiments on machine-learning applications (2Gbps network).

three points of the graph and then extrapolate the results. The flat layout throughput scales linearly to the cross-region total bandwidth.

To evaluate the hierarchical layout, we need to first determine the workload of each coordinator using cvxpy. From the graph we can see that the hierarchical layout achieves a speed up of $4\times$ to $4.5\times$ over the flat layout.

Machine learning applications. In this section we evaluate Cerebro using decision tree prediction, logistic regression training via SGD, and linear regression training via ADMM [250, 257, 300]. We estimate the network cost for the preprocessing phase of the arithmetic protocol using the throughput gathered in the previous benchmarks.

Decision tree prediction. We implement decision tree prediction using Cerebro's DSL, which evaluates a complete h layer binary decision tree, where the i^{th} layer has 2^{i-1} nodes. We evaluate a scenario where there are P parties, one of which has the input feature vector and all P parties secret-share a model. If $P = 2$, we assume that we are doing a two-party secure prediction, where one party has the feature vector and the other has the model.

We show the prediction performance in the 2-party semi-honest setting in Figure 4.8a. In this experiment, we varied the number of layers in the decision tree. We fit the data points involving $3, 6, 9$ layers and then extrapolate the cost model to estimate the performance of our graphed points. Cerebro always picks the protocol that has the lower estimated cost from our model. In the 2-party scenario, Cerebro always chooses to use a boolean protocol since evaluating the decision tree requires many comparisons and data selection. In a 12-layer tree, the semi-honest boolean protocol takes $7.5\times$ less time than the semi-honest arithmetic protocol. In Figure 4.8b, we vary the number of parties, and plot the inference runtime for a 10 layer tree. We observe that the total execution time for the boolean protocol grows linearly with number of parties, and sublinearly for the arithmetic protocol. Therefore, with 9 or more parties, Cerebro chooses to use the arithmetic protocol.

As noted previously, Cerebro also supports the malicious setting, and we exclude those results for brevity.

• **Logistic regression.** We implemented and evaluated Cerebro on logistic regression training using SGD. In this experiment, we evaluated training in both the semi-honest and the malicious settings to show a difference in the performance for different variants of the protocols. For the semi-honest and malicious boolean protocols, we ran logistic regression for one iteration of SGD and extrapolated the remaining results. First, we compare the performance between the semi-honest boolean and semi-honest arithmetic protocol in Figure 4.8c. We run one epoch over the dataset in these experiments with a batch size of 128. As expected, the arithmetic protocol significantly outperforms the boolean protocol in this case, both because it is better suited for this task and because it enables vectorization. Using these results we see that for a 27000 record training set the arithmetic protocol is 67× faster than the boolean protocol, taking an hour instead of three days.

However, in the malicious setting, the arithmetic protocol does not always perform better. The amount of memory used by the malicious boolean protocol is linear in the number of parties and the number of gates. As a result, we run out of memory when trying to benchmark larger circuits. We estimate the malicious boolean protocol on machines with enough memory as well as on the original machines with swap space to use as additional memory. As shown in Figure 4.8c, if the machines have enough memory, then the malicious boolean protocol is 3× faster than the malicious arithmetic protocol, but if swap space is used instead, then the malicious boolean protocol is 4× slower than the malicious arithmetic protocol. Overall, the malicious boolean protocol is up to 61× slower than its semi-honest counterpart and the malicious arithmetic protocol is up to 3300× slower than its semi-honest counterpart, indicating a significant tradeoff between performance and security.

• **ADMM.** We evaluate ADMM in the semi-honest setting to show Cerebro's automated planning of local computation. Cerebro automatically detects that the parties can locally compute much of the ADMM algorithm, thus minimizing the number of MPC operations required as described previously in Section 4.4. We evaluate these benefits in Figure 4.8d and find that the use of local computation allows Cerebro to improve ADMM performance by up to 700× when training a 40-feature model using 10000 records per party for 6 parties. We estimate the preprocessing and run the online phase for the first four data points, but estimate the fifth. Beyond this we also find that the use of arithmetic circuits is beneficial here for the same reasons as in the case of logistic regression, i.e., it allows vectorization and is better suited to expressing matrix operations.

4.7.3 Policy evaluation

We evaluate the performance of Cerebro's release policies in the semi-honest setting. Specifically, we evaluate logistic regression that uses both differential privacy and the threshold-based validation policies. Our differential privacy policy is output perturbation-based [286, 287], which simply requires each party to locally sample noise. The secure computation will sum every party's noise and add the noise to the weights. As Table 4.6 shows, the time for adding this noise is independent of the number of training samples, and is insigificant compared to the training time.

The threshold-based validation policy requires the model to achieve a sufficient level of accuracy in order to be released. To see how much time is needed for validation, we split the dataset with 30000 records into a training set of 27000 records and a validation set composed of the remaining 3000 records. We train the model using a subset of the training set and validate the trained model using part of the validation set which is 10% the size of the used training set. From Table 4.6, we can see that the validation time grows linearly to the used validation set. Compared with training in logistic regression, the time taken by validation is equivalent to training another 10% of the training samples, which matches the training behavior of logistic regression.

4.7.4 Auditing evaluation

Next, we present the overheads from enabling auditing support for logistic regression. There are two main costs in this case. The first cost is producing and signing a commitment, which takes 24.4 seconds, of which 8 milliseconds are spent generating a signature for user input (which is a 27000×23 matrix in our case). The second cost is spent on the commitment protocol described in Section 4.5.2. Checking the commitment within MPC using a non-batching commitment scheme such as subset sum takes approximately 4.5 days while checking the commitment using a batching commitment scheme such as Pedersen commitments takes approximately 2.23 hours. The speedup is roughly $53\times$, which only grows as the number of samples increases as the batched commitment scheme scales better with respect to the number of samples. Overall we find that enabling auditing has reasonable overhead.

4.7.5 Comparison with hand-tuned protocols

We compare with three hand-tuned protocols: SecureML's logistic regression [71], EzPC's decision tree [262], and secure ridge regression [255] (see Tables 4.3 to 4.5). Since [71] and [255] are not open sourced, we compare to the reported numbers; we ran EzPC since they provide an open source repository. These works also only support two parties who are semi-honest whereas Cerebro supports an arbitrary number of parties under different threat models. Compared to SecureML, Cerebro has 10–92× performance overhead. Cerebro performs better in the WAN setting than the LAN setting due to better batching. Compared to EzPC, Cerebro has an overhead of $3\times$. Compared to ridge regression, our compiler discovers similar insights as the hand-tuned protocol, except we can *automatically* split a program into plaintext precomputation and MPC. Cerebro is $2.5\times$ slower on a dataset with 20 features and 1 million samples, and $2.5\times$ faster for a dataset with 10 features and 1 million samples. We also tested Cerebro's performance with and without automatic optimization on a dataset with 20000 samples and 10 features, and Cerebro with precomputation is $25\times$ faster. We did not test larger circuits for the baseline because it could not run.

4.7.6 Discussion on automatic optimization

Based on these evaluation results, we believe that automatic compilation and optimization of MPC protocols has a lot of potential. Compared to hand-tuned MPC protocols, Cerebro's performance comes close or even exceeds that of protocols specifically tailored to a particular threat model and application. Though Cerebro cannot always compile a protocol that is as efficient as a hand-tuned version (which is true for regular compilers as well), our compiler can generalize

# Training samples	# Training features	Cerebro time (s)	Secure Ridge Regression time (s)
1000000	10	51.23	80
1000000	15	247.88	180
1000000	20	767.89	330

Table 4.3: Comparison with secure ridge regression [255].

# Training samples	# Training features	Network type	Cerebro time (s)	SecureML time (s)
10000	100	LAN	825.17	8.9
10000	500	LAN	2563.39	63.37
10000	100	WAN	3941.28	12.59
10000	500	WAN	10345	950.2

Table 4.4: Comparison with SecureML's logistic regression [71].

# Nodes	# Dims	Cerebro time (s)	EzPC time (s)
3095	13	7.15	3.67
2048	64	7.22	3.41

Table 4.5: Comparison with EzPC [262].

to any learning task, hence obviating the need for users to consult an expert for every new functionality. For experts who wish to hand-optimize a learning task, Cerebro's compiled program can also act as a starting point upon which more efficient MPC protocols can be built. We hope that Cerebro can also act as a standard platform for researchers to continue to improve automatic MPC optimization. One area for research is how an MPC compiler handles memory's impact on performance. Cerebro could easily be extended to model memory usage directly for MPC backends, or work with runtime cost models with memory size as an input parameter.

4.8 Related work

Related plaintext systems. There is a large body of prior work on distributed linear algebra systems [301–303] and machine learning training/prediction [304–309].

While some of these systems are general and can be adapted to the distributed setting, they do not provide security guarantees and cannot be used in the collaborative machine learning on sensitive data. Some of these systems provide interesting linear algebra optimizations that are

# Training samples	D.P. time (s)	Validation time (s)
1000	1.192	14.19
5000	1.192	48.66
15000	1.192	140.34
25000	1.192	238.01
27000	1.192	257.05

Table 4.6: Time for applying policies to logistic regression.

similar to Cerebro's optimizations at a very high level, but Cerebro additionally must consider the effects of optimizing a *cryptographic* protocol. This means that Cerebro has different rules for transformation and a very different cost model. The idea of "physical planning" is similar to prior systems and database work [310–316]. The main difference is that we instantiate this idea to the MPC setting and work closely with the underlying cryptography.

MPC compilers. Cerebro draws inspiration from a body of work on MPC compilers [210, 216, 217, 259–268, 317–324]. Compared to prior work, Cerebro's compiler differs in two important aspects. First, we provide n-party compilation supporting two MPC frameworks under different threat models. There is prior work providing n-party compilation supporting a single framework [210, 217, 259, 260, 264–266] and two-party compilation supporting multiple frameworks [262, 267, 317]. Second, Cerebro adds optimization in *both* the logical and the physical layers, which allows us to consider a multitude of factors like computation type, network setup, and others. Conclave [268] is a recent system that is similar to Cerebro because it handles multiple frameworks and does optimization. However, it is designed for SQL, and does not consider physical planning or release policies. Finally, Cerebro itself is an end-to-end platform for collaborative learning and supports policies and auditing.

Secure learning systems. There is prior work that uses hardware enclaves to execute generic computation, analytics, or machine learning [325–330]. Compared with Cerebro, the threat model is quite different. While hardware enclaves support arbitrary functionality, the parties have to put trust in the hardware manufacturer. We have also seen that enclaves are prone to leakages [331–334].

There has been much work on secure learning using cryptography, both in training and prediction [71, 249–257, 335–366]. A few of these systems make advances in the N-party collaborative learning setting, such as Abspoel et al. [353], Chase et al. [367], Chen et al. [348], Chen et al. [361], CaPC [364], Helen [250], POSEIDON [350], and SPINDLE [335]. However, Cerebro differs from these works in several aspects. First, they mostly focus on optimizing specific training/prediction algorithms and do not consider supporting an interface for programming generic models. Second, they do not automatically navigate the tradeoffs of different physical setups. Finally, these frameworks also do not take into account the incentive-driven nature of secure collaborative learning, while Cerebro supports policies and auditing.

Other related work. A recent paper by Frankle et. al. [368] leverages SNARKs, commitments, and MPC for accountability. However, the objective is to make the government more accountable to the public, so the setting and the design are both quite different from ours. Other papers [369–371] explore identifying cheating parties in maliciously secure MPC. However, these papers are either highly theoretical in nature, or require proof that each party behaved honestly during the *entire protocol execution*, which can be quite expensive. Cerebro is mainly concerned with holding the users accountable for their input data, and our scheme both works with multiple MPC frameworks and does not need to require proof of honest behavior for the entire protocol execution. With regards to the logical optimizations that Cerebro performs, there has been work [261] that also performs partitioning of computation into local and secure modes. However, Cerebro does not require the user to specify the mode of computation for every single operation and instead automatically partitions the source code into local and secure components.

4.9 More details on physical planning

4.9.1 Cost models

Boolean MPC preprocessing cost. For boolean MPC, there are two phases within the preprocessing phase. The first phase is very similar to the preprocessing generation phase for arithmetic MPC, except that this step now generates AND triples instead of multiplication triples. For this phase, Cerebro only provides one method, which is similar to the quadratic preprocessing, and has the same cost model as Equation (4.2). The second phase is a circuit generation phase, where each party creates a copy of the final circuit and sends it to a *single* party. This "evaluator" party will be in charge of executing the circuit during the online phase.

Therefore, the cost model for the boolean MPC preprocessing phase is:

$$c + N_a(P-1)(f_1(\lambda) + g_1(B,\lambda)) + g_2(B,\lambda)N(P-1) \tag{4.4}$$

where λ is the security parameter, g_1 refers to the cost of preprocessing AND gates, and g_2 refers to the cost for a single evaluator to receive $P-1$ copies of the garbled circuit.

Online execution cost. The online phases for arithmetic and boolean MPC have quite different behaviors, which in turn result in different cost models. Arithmetic MPC requires interaction (hence network round trips) among all parties for multiplication gates throughout the entire computation. The number of round trips is proportional to the *depth* of the circuit. Boolean MPC, on the other hand, is able to evaluate the online phase in a *constant* number of rounds. Therefore, arithmetic MPC's online phase can be modeled as $c+g(l)R$, where R indicates the number of communication rounds. We do not consider the compute cost because it should be very insignificant compared to the cost of the round trips.

Boolean MPC's online phase is modeled as $c + f(\lambda)N$, where λ is the security parameter. This captures the compute cost of evaluating the entire boolean circuit. There is interaction at the beginning of the protocol because the evaluator needs to receive encrypted inputs, and at the end of the protocol because the evaluator needs to publish the output.

4.9.2 Linear vs. quadratic preprocessing

Without diving into the cryptography, we describe these two methods at a very high level. Both methods are *constant round*, which means that they only need 1–2 roundtrips. In linear preprocessing, each party independently generates data for each triple and sends this data to a set of *coordinators*. The coordinators then aggregate this data, compute on it, and send the results back to each party. A similar pattern repeats for a second round. Since the triples can be generated independently, we distribute the coordination across all parties. In quadratic preprocessing, each party interacts with every other party in constant round to compute the triples.

4.9.3 Extended description of layout optimization

In this section, we give an extended analysis of the layout optimization problem. For an easier analysis, we assume that there are at most two regions (see Figure 4.4). In order to explain our cost model, we first define some preliminary notation as follows. The two regions are denoted as L and R. P_L parties are located in region L, and P_R parties are located in region R. We assume that each party has roughly the same computation power, that each party has a fixed inbound and outbound bandwidth limit for in-continent data transfer, and that between the two regions there is another inbound and outbound bandwidth limit shared by all the parties. Let n_L be the number of triples that a single global coordinator in L handles; n_R is similarly defined for region R. Hence we have the following relation $n_L \cdot P_L + n_R \cdot P_R = N_m$. The cost (i.e., the wall-clock time) for preprocessing arithmetic circuits is:

$$T = g_1(B_1)(L_1 + L_2) + g_2(B_2)L_3 + f_1(|p|)L_4 + f_2(|p|)(L_1 + L_3) \tag{4.5}$$

B_1 is the intra-region bandwidth per party, while B_2 is the *total* inter-region bandwidth between the two regions. Therefore, the g_1 and g_2 terms capture the network cost. The f_1 and f_2 terms correspond to the compute cost, where f_1 captures ciphertext multiplication, and f_2 captures the other ciphertext operations. L_1–L_4 are scaling factors that are functions of $n_L,\ n_R,\ P_L,$ and P_R:

$$\begin{aligned}
&L_1 = \max\left(n_R \cdot \tfrac{P_R \cdot (P_L - 1)}{P_L}, n_L \cdot \tfrac{P_L \cdot (P_R - 1)}{P_R}\right), \\
&L_2 = \max\left(n_L \cdot (P_L - 1),\ n_R \cdot (P_R - 1)\right), \\
&L_3 = \max\left(n_L \cdot P_L,\ n_R \cdot P_R\right),\ L_4 = \max\left(n_L,\ n_R\right).
\end{aligned}$$

The intra-region communication cost is captured by the g_1 term. Because of hierarchical planning, each node needs to act as both an intra-region coordinator and an inter-region coordinator. Without loss of generality, we analyze region L. The intra-region coordination load is $n_L \cdot (P_L - 1)$, because each node receives from every other node in the region. The inter-region coordination load can be derived by first summing the total number of triples that need to be partially aggregated within L, which is equal to the total number of triples handled by region R: $n_R \cdot P_R$. Since there are P_L parties, each party handles $n_R \cdot P_R / P_L$ triples. Finally, since each party only needs to receive from the other parties in L, the cost per party is $nR \cdot P_R(P_L - 1)/P_L$. The g_2 term captures the inter-region communication cost. Since we are doing partial aggregation, we found that the best way to capture this cost is to sum up the total number of triples per region (see L_3) and scale that according to the total inter-region bandwidth B_2. The f_1 term captures the ciphertext multiplication cost. Since that happens only once per triple at the intra-region coordinator, we have the scaling in L_4. Finally, the rest of the ciphertext cost is attributed to ciphertext addition. This can be similarly derived using the logic for deriving g_1, so we omit this due to space constraints.

Finally, for the k region case, we can transform the optimization problem described in Section 4.4.3 into a linear program by moving the $\max$ into the constraints as follows:

$$\begin{array}{lll}\min(L'_1 + L'_2 + L'_3) \text{ s.t.} & \sum_{i=1}^{k} n_i \cdot P_i \geq N_m & \\ & L'_1 \geq \sum_{j \neq i} \left(\frac{n_j \cdot P_j (P_i - 1)}{P_i B_i} \right) & i = 1, 2, ..., k, \\ & L'_2 \geq \frac{n_i \cdot (P_i - 1)}{B_i} & i = 1, 2, ..., k, \\ & L'_3 \geq \sum_{j \neq i} \frac{n_j \cdot P_j}{B_{ij}} & i = 1, 2, ..., k \end{array}$$

We loosen the first constraint to be an inequality rather than an exact equality to make it easier to find feasible solutions since we require the n_i's to be integral. Therefore, the equations above formulate the linear program we solve to obtain the optimal assignment of triple generation tasks.

Chapter 5

HOLMES: Efficient Distribution Testing for Secure Collaborative Learning

Many organizations in healthcare, finance, law enforcement want to train a model over a large amount of *high-quality* data, and one way to find such data is through a collaboration between organizations in the same industry. Using *secure multiparty computation (MPC)*, they can train the model over the joint dataset without revealing the data to each other.

However, secure computation makes it difficult for organizations to assess the quality of each other's data. Without such an assessment, it is *irresponsible and dangerous* to deploy the model in practice, especially in healthcare, finance, or law enforcement where humans can be significantly affected by the model's potentially biased decision. Regulations such as European Union's General Data Protection Regulation (GDPR) require organizations not only to take steps to prevent errors, bias, and discrimination in such models but also to be legally responsible for the damage. How can an organization take the risks of another organization's data that is largely untested?

Hence, we believe that testing the quality of data is an *indispensable* step in secure collaborative learning. The bottom line is that every organization should check if the datasets are biased or severely unbalanced. However, performing such distribution testing is very expensive in MPC, as MPC needs to hide the data by performing costly oblivious operations over encrypted data.

We present HOLMES, a protocol for performing distribution testing *efficiently*. In our experiments, compared with a nontrivial baseline, HOLMES achieves a speedup of $270\times$ for various tests and up to $10^4\times$ for multidimensional tests. The core of HOLMES is a hybrid protocol that integrates MPC with zero-knowledge proofs and a new ZK-friendly and naturally oblivious sketching algorithm for multidimensional tests, both with *significantly lower* computational complexity and concrete costs.

5.1 Introduction

An article from MIT Technology Review, "AI is sending people to jail–and getting it wrong" [372], again reminds us that machine learning models have not only been deployed everywhere but also started to determine people's fate. For example, AI models have been used to detect traditionally unrecognized anxiety and depression from speech [373, 374] and diagnose attention deficiency [375], in which accuracy is crucial for people to get the right treatment. Mortgage approval is often decided by models, which have been shown to more likely deny loans to people of color [376–378]. The most infamous example is probably COMPAS[1] [379], a system that rates people's risk of future crime and decides if one should be held in jail before trial and the severity of the sentences, has been shown to disproportionately target the low-income and minority [372].

Underlying data rather than the algorithm is most often the main source of the issue [380]. When the training data is inaccurate, skewed, or systemically biased (due to how data is collected, for example), the model, which we train to *fit* the training data, becomes biased. This implies a potential solution: to avoid a biased model, we should, at the very least, attempt to avoid learning from unbalanced data [381].

In the machine learning community, there have been many approaches (e.g., [382–384]) to protect fairness, starting from detecting and removing unbalances from the training data. For example, if one finds a dataset with an overwhelming number of negative records toward a protected group, one can adjust the weight or reduce the number by subsampling. Yet, the situations with respect to *secure collaborative learning*, where the model is trained in a way that none of the parties can see the data, are very different.

In secure collaborative learning, many organizations train a model over their joint dataset using MPC. If the data is of high-quality, by having more data, we expect the model to be better [385–389], but we really do not know if the data is indeed of high quality. This is crucial because the model may later be used to affect people. GDPR [247] requires organizations using such models to prevent errors, bias, and discrimination and take liability of the models [390]. Due to MPC, the organization that uses the model does not actually know the training data, but, however, the organization is *responsible* for the mistakes of the model, which can be caused by unbalanced data from another party in the collaborative learning. How can an organization take the (unknown) risk for another organization's *untested* data?

We believe that, in secure collaborative learning, different parties should not just focus on data confidentiality and integrity, but also *data quality*. Specifically, we think the first step toward this direction is to perform *distribution testing* over the data used in the collaborative learning, to examine:

- **One-dimensional distributions,** such as histogram that describes the distributions of values in one dimension (e.g., income), or as basic as whether data in the entry is in the right range (e.g., age should be ≤ 1000).
- **Multidimensional distributions,** such as whether the joint distributions of several dimensions *fit* into a desired distribution, such as a *balanced* demographic composition.

Distribution testing has been used to understand data quality and bias in clinical trials. The NIH Collaboratory [391] recommends comparing the distributions of different datasets to detect data discrepancies. Such a useful tool is, however, missing in prior works in secure collaborative learning (e.g., [250, 337, 365, 392]), often left as an open problem. This is likely due to the *extremely*

[1] Correctional Offender Management Profiling for Alternative Sanctions.

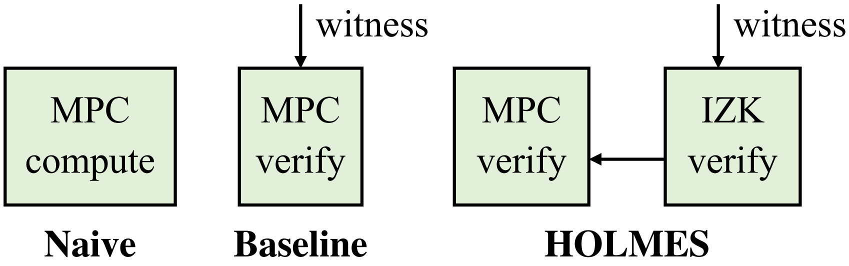


Figure 5.1: Three computation models for distribution testing.

high overhead of distribution testing. Checking if a number x is in a range $[a, b]$, for example, already requires a very complicated protocol in MPC [217].

We present HOLMES, a protocol that performs such distribution testing efficiently, often at only a small fraction of the cost of secure collaborative learning. In our experiments, compared with a nontrivial baseline that we describe below, HOLMES achieves a speedup of $270\times$ for various distribution tests and up to $10^4\times$ for multidimensional tests. The core of HOLMES is a hybrid protocol that integrates MPC with zero-knowledge proofs and a new ZK-friendly, naturally oblivious sketching algorithm for multidimensional tests. HOLMES is already open-sourced in GitHub (anonymously): `https://github.com/holmes-anonymous-submission/`

5.1.1 Finding the right computation model

Our first task is to bypass the inefficiency of MPC. This seems to be impossible, as for security, data cannot leave MPC. Our insight is that distribution testing is theoretically a decision problem [393], i.e., yes or no. It is *verification*, not computation. This brings us to the baseline described as follows.

Baseline: verification vs computation. There is a nontrivial baseline that significantly improves the performance from the naive solution that computes the distribution testing in MPC. In this model, each party provides some auxiliary information, called *witness*, for the distribution testing of their part of the dataset, which the party knows the data. For example, checking that x is in range $[a, b]$ becomes easier given bit decomposition of $(x - a)$ and $(b - x)$ as the witness. MPC only needs to perform binary testing over the bit decomposition and perform equality checks. This solution comes with a cost, as the party needs to input the witness to MPC.

This is, however, still expensive, especially when the number of parties N is high. Assuming that each party's dataset is of the same size, the amount of data to be tested is $O(N)$.

- In the best case, if all parties agree on a single distribution testing (of computation overhead C) to be applied to all individual datasets, the amount of computation is $O(CN)$. Running this computation in N-party MPC leads to a wall-clock time of $O(CN^2)$.

- In the worst case, each party requires a different distribution testing (assuming each is of overhead C) to other parties, the amount of computation is $((NC) \cdot N)$. Running this in N-party MPC leads to a wall-clock time of $O(CN^3)$.

In this paper, we present a third computation model that reduces the wall-clock time to $O(CN)$ even in the worst case.

HOLMES: low-complexity via IZK. We make use of interactive zero-knowledge (IZK) proofs, which involve a prover and a verifier as follows. Each party in the collaborative learning takes turns to prove to and verify each other in respect to distribution testing. Concretely, each party proves to $(N-1)$ parties and verifies $(N-1)$ parties. For a distribution testing of overhead C, the wall-clock time for IZK is $O(CN)$. This applies to the worst case as well: a prover simply proves different statements to different verifiers.

We can see that IZK immediately reduces the complexity as it avoids the penalty due to having more parties in the MPC. In essence, the distribution testing is not a computation that requires N-party data, it is about one party's data, and it does not actually need N-party MPC. We feel that many secure distribution testing would benefit from this model.

This new model leaves two questions: First, how can we ensure that the data in MPC is exactly the same data proved in IZK? We present, in Section 5.3, a lightweight consistency checking protocol from DeMillo-Lipton-Schwartz–Zippel lemma [191–193] that solves this in $O(N)$ wall-clock time and is concretely efficient. Second, is this low-complexity IZK also concretely efficient? Thanks to the recent development of IZK from silent OT [394–400], the answer is yes. Our evaluation in Figure 5.7 confirms these gaps in complexity and concrete costs.

Remark: why not NIZK?. A natural question is why we use IZK instead of non-interactive ZK (NIZK), which can further reduce the wall-clock time to $O(C+N)$, if the NIZK is succinct and holographic. The reason we choose not to use NIZK is due to their extremely high prover overhead. IZK from silent OT, which cannot be NIZK by applying Fiat-Shamir transform [218] due to the use of private coins, is much more efficient. This gap could be in fact inherent, as an IZK can often reason about a statistical security parameter $\sigma \approx 2^{40}$, while NIZK needs a computational security parameter $\lambda \approx 2^{128}$. It is an open problem to find an NIZK that is concretely more efficient for our setting.

5.1.2 Testing multidimensional distributions

Our second task is to *efficiently* test multidimensional distributions. For example, for a financial dataset, instead of focusing on one dimension such as "debt", we actually want to understand the distributions of several dimensions, for example gender, age, race, income level, and debt, and we want to ensure that it is not too far away from a balanced distribution, in which different genders, ages, races, and income levels are fairly represented.

In plaintext systems, this can be done by putting the data into the buckets, counting the number of samples in the buckets, and then performing a Pearson's χ^2 test. The situations with respect to secure learning are very different because the bucketing must be *oblivious*, in that it cannot reveal which bucket a sample goes. This either requires a linear scan, as shown in Figure 5.2,

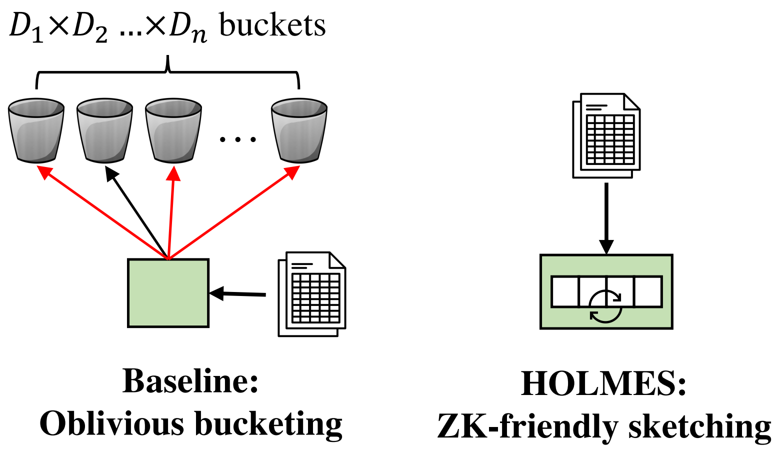


Figure 5.2: Methods for multidimensional distribution testing.

or oblivious RAM (ORAM) [**ORAM:goldreich1987towards**, **ORAM:goldreich1996software**, **ORAM:RevisitedPinkas:2010**, **ORAM:Shi:2011**, **ORAM:Stefanov:TowardsPracticalORAM:2012**, **ORAM:gentry2013optimizing**, 186, 187] in MPC, which has a lower complexity but is concretely expensive.

The curse of oblivious bucketing. The linear scan is slow because for each sample, it needs to "update" each bucket. The number of buckets is the product of the range of each dimension, i.e., $\prod_i D_i = D_1 \times D_2 \times ... \times D_n$. In one of our experiments in Section 5.6, $\prod_i D_i = 37500$. As shown in Figure 5.2, this means that beside one real update (indicated by a black line), there are $\prod_i D_i - 1$ dummy updates for padding (indicated by red lines). The amount of computation for m samples is therefore $O(m \prod_i D_i)$. Our experiment in Section 5.6 shows that such oblivious bucketing can take 10^5 seconds.

Streaming and sketching to the rescue. We find two concepts from data science research closely related to oblivious computation: *streaming* and *sketching* [401–405].

- **Streaming:** an algorithm that takes input as a sequence and only needs access to *limited* memory.
- **Sketching:** an algorithm that (approximately) performs the computation, using a *compressed* representation of the data.

Such algorithms naturally fit oblivious computation, because they have sequential memory accesses and, if a linear scan is needed to access certain locations, the amount of padding needed is small. There have already been research targeting specifically on oblivious sketching [406, 407]. In HOLMES, we present an oblivious sketching algorithm that, given pseudorandomness, compresses the training data. This algorithm applies a random linear projection that approximately preserves the distance to the data. The distribution testing can be still done with high accuracy, according to the Johnson-Lindenstrauss lemma [408].

The last challenge is how to efficiently obtain pseudorandomness in IZK. Running a classical

Ideal functionality $\mathcal{F}$

Running with N parties $\mathcal{P}_1, \cdots, \mathcal{P}_N$ and a simulator Sim, $\mathcal{F}$ proceeds as follows: Upon receiving a message $(sid, \mathcal{P}_i, \mathsf{input}_i, \mathsf{test}_i)$ from each of the N parties (or from Sim if that party is corrupted),

1. **Abort:** $\mathcal{F}$ awaits a message $(\mathsf{deliver}, sid)$ or (abort, sid) from Sim to decide whether the computation should move forward. $\mathcal{F}$ proceeds to the next step if the message is $(\mathsf{deliver}, sid)$. Otherwise, $\mathcal{F}$ sends (abort, sid) to each $\mathcal{P}_i$ and Sim, and halts.
2. **Distribution tests:** $\mathcal{F}$ runs the distribution tests specified in $(\mathsf{test}_1, \ldots, \mathsf{test}_N)$ with respect to the each input $(\mathsf{input}_1, \ldots, \mathsf{input}_N)$. For each distribution test, $\mathcal{F}$ sends either (accept, sid) or (reject, sid) to each $\mathcal{P}_i$ and Sim, and halts.

Figure 5.3: Ideal functionality for distribution testing.

pseudorandom function, such as SHA-256, is impractical. We also find that ZK-friendly hash functions [409, 410] are still costly. Instead, we strive to find a *tailored* way to obtain pseudorandomness for the specific random projection that we are performing. We call this overall construction "ZK-friendly sketching".

Finding a random function for random projection. The specific random projection that we are performing requires a pseudorandom map with an one-bit output $b \in \{-1, 1\}$. Although we can use any pseudorandom function and extract one bit from it through bit decomposition, this process is costly in IZK. Instead, we identify that Legendre PRF, which has been studied recently [411–414] and can be seen as a random universal one-way hash function (UOWHF) [415, 416] with a one-bit extractor, is a natural fit, and is extremely cheap—it only requires 8 input or multiplication gates.

With these techniques, HOLMES's multidimensional distribution testing only requires $O(m \log m)$ computation (as long as $\prod_i D_i$ is not too large). We show, in our experiments, that this solution is up to $10^4\times$ faster compared with the baseline.

5.1.3 Summary of contributions

In sum, HOLMES's contributions are the following:

- A new hybrid protocol that integrates MPC and IZK for distribution testing, which has lower complexity and concretely much more efficient (up to $270\times$ in our experiment) compared with a nontrivial baseline;
- A new efficient multidimensional distribution testing procedure via ZK-friendly sketching, which has lower complexity and concretely much more efficient (up to $10^4\times$ in our experiment) compared with the baseline;

- Implementation of HOLMES with well-known statistical tests including z-test, t-test, F-test, and χ^2-test;
- Extensive experimental evaluation that uses diverse datasets such as a bank-marketing [417, 418] dataset, a diabetes [419, 420] dataset, and an auctioning dataset [421].

5.2 Overview

We consider N parties who participate in a secure collaborative learning, as shown in Figure 5.4. In HOLMES, the parties perform two types of computation: MPC and IZK. All the N parties run the same instance of MPC, which has loaded each party's dataset. During the distribution testing, the parties will run IZK in a *pairwise* and *bidirectional* manner, where each party communicates with every other party, and both parties take turns as a prover and verifier to perform distribution testing on each other's individual dataset.

Tests. A distribution test is a predicate over a dataset or more than one datasets (from different parties). Examples include well-known statistical tests, such as mean equality tests z-test (known variance) and t-test (unknown variance), variance equality test F-test, and Pearson's χ^2-test. All these tests target the distances between two datasets, or between a dataset and a public distribution. HOLMES implements various distribution test gadgets, as shown in Table 5.1.

Workflow. HOLMES is a protocol to perform distribution testing for secure collaborative learning running in MPC, or in other words, a subroutine of the MPC that runs at the early stages of the secure collaborative learning when all the parties have inputted their data to MPC. Now, right before MPC starts to run the actual training algorithm, HOLMES is invoked to perform distribution tests, as follows.

1. **Loading the data to IZK:** Each party inputs their data to IZK. This prevents the party from changing the input adaptively after seeing the revealed distribution tests.
2. **Revealing the distribution tests:** After the data has been loaded, the parties reveal the distribution tests that they want to perform over other parties' datasets.
3. **Consistency check:** All parties perform the consistency check in Section 5.3 to check that the inputs to IZK and MPC are equal. The parties abort if the check fails.
4. **IZK verification:** Every pair of parties take turns to be the prover and the verifier in IZK. The prover proves the correct calculation of some specified statistics about the data, which is verified by the verifier without seeing the data. The verifier aborts if IZK fails.
5. **MPC finishing touch:** For distribution tests that involve more than one party's data (e.g., z, t, F-tests), we need MPC to look at the statistics of each party's data, compute the final test statistic, and decide whether the data passes the test. Here, MPC only needs to compute over *statistics*, which is small, and thus we call it the "finishing touch".

When the distribution testing is done, the original MPC protocol continues with the secure collaborative learning.

Table 5.1: Distribution test gadgets in HOLMES (α denotes the significance level).

Gadget	Description
$\mathsf{range}(\langle S\rangle, \mathsf{attr}, [a,b]) \to \{\mathsf{yes}, \mathsf{no}\}$	check that values for an attribute attr in population S fall in the range $[a,b]$
$\mathsf{histogram}(\langle S\rangle, (\mathsf{attr}_1, ..., \mathsf{attr}_n),$ $([a_1,b_1], \ldots, [a_D,b_D])) \to \overrightarrow{\mathsf{count}}$	count elements of S in $D = \prod_{i=1}^{n} D_i$ non-overlapping (multidimensional) buckets for attributes $(\mathsf{attr}_1, \ldots, \mathsf{attr}_n)$
$\mathsf{mean}(\langle S\rangle, \mathsf{attr}) \to \overline{x}$	mean of S for an attribute attr
$\mathsf{variance}(\langle S\rangle, \mathsf{attr}) \to s^2$	variance of S for an attribute attr
$\mathsf{trimmedMean}(\langle S\rangle, \mathsf{attr}, \theta) \to \overline{x}$	mean of elements in S belonging in the range $[0,\theta]$ for an attribute attr
$\mathsf{zTest}(\langle S_1\rangle, \langle S_2\rangle, \mathsf{attr}, \sigma_1, \sigma_2, \alpha) \to \{\mathsf{yes}, \mathsf{no}\}$	z-test for mean equality for an attribute attr of populations S_1 and S_2 with standard deviations σ_1, σ_2, respectively
$\mathsf{tTest}(\langle S_1\rangle, \langle S_2\rangle, \mathsf{attr}, \alpha) \to \{\mathsf{yes}, \mathsf{no}\}$	t-test for mean equality for an attribute attr of populations S_1 and S_2
$\mathsf{FTest}(\langle S_1\rangle, \langle S_2\rangle, \mathsf{attr}, \alpha) \to \{\mathsf{yes}, \mathsf{no}\}$	F-test for variance equality for an attribute attr of populations S_1 and S_2
$\mathsf{chiSquaredTest}(\langle S\rangle, (\mathsf{attr}_1, \ldots, \mathsf{attr}_n),$ $([a_1,b_1], \ldots, [a_D,b_D]), (p_1, \ldots, p_D), \alpha) \to \{\mathsf{yes}, \mathsf{no}\}$	χ^2 test for goodness-of-fit of attributes $(\mathsf{attr}_1, \ldots, \mathsf{attr}_\mathsf{d})$ for population S for multidimensional buckets $[a_j, b_j]_{j=1}^{D=\prod_{i=1}^{d} D_i}$ and public distribution $(p_1, ..., p_D)$

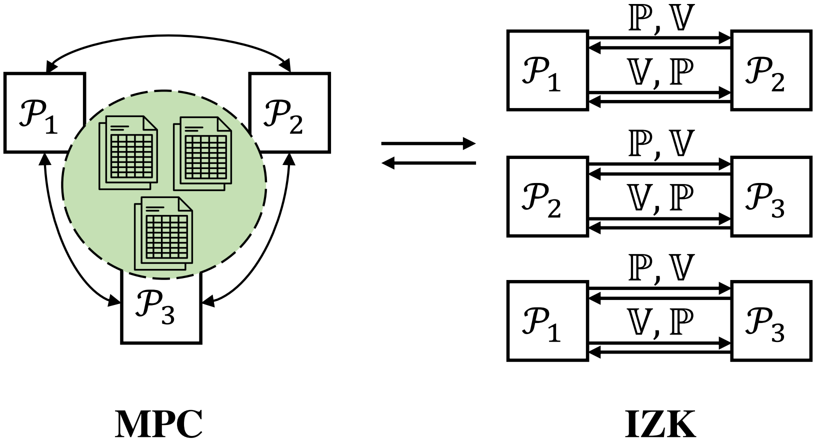


Figure 5.4: System model of HOLMES.

5.2.1 Threat model and assumptions

Up to $N-1$ of the parties can collude and arbitrarily deviate from the protocol. We define the security of HOLMES in the real/ideal world paradigm [422, 423].

The ideal functionality $\mathcal{F}$ (Figure 5.3) takes as input the list of distribution tests and the data; it outputs whether the data passes the tests or not. Based on $\mathcal{F}$, we define the security of HOLMES using a standard definition for (standalone) malicious security [200]. We provide the security proof sketch in Section 5.9.

Definition 3. A protocol Π is said to *securely compute* $\mathcal{F}$ in the presence of static malicious adversaries that compromise up to $N-1$ of the N parties, if, for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real world, there exists a non-uniform probabilistic polynomial-time simulator Sim in the ideal world, such that the outputs of the two worlds are computationally indistinguishable.

Leakage from distribution tests. Note that any distribution testing leaks one-bit information—passing or not passing the tests. We assume that, when each party reveals the distribution tests it wants to enforce on other parties' data, the other parties will refuse those tests that may leak sensitive information.

More assumptions. Parties in the secure collaborative learning are responsible for specifying the distribution tests, which are revealed after the same data is loaded to MPC and IZK. We informally assume those tests are *not predictable*, and the parties cannot effectively tune the input data accordingly before the distribution testing. In addition, parties can abort after knowing the distribution tests (e.g., pretending that the network is down) and request to redo the distribution testing, so that the parties can tune the data according to the tests. We assume that MPC can enforce that parties in the second execution must use the same data as the first execution, which is commonly supported in many MPC protocols.

5.3 Consistency check

HOLMES needs to ensure that data used in MPC is exactly the data being proved in IZK. We now describe a lightweight consistency check protocol that realizes this functionality.

Definitions. We define the consistency check as a procedure $\mathsf{CC}^{\mathcal{O}_{\mathsf{MPC}},\mathcal{O}_{\mathsf{IZK}}}(1^\lambda) \rightarrow b \in \{0,1\}$ that controls the MPC via an oracle $\mathcal{O}_{\mathsf{MPC}}$ and controls the $N(N-1)$ instances of pairwise IZK via an oracle $\mathcal{O}_{\mathsf{IZK}}$. Before running this procedure, the data must be already loaded in MPC and IZK. We want this consistency check to have the following security properties:

- **Completeness:** if the same data is loaded to MPC and IZK, $\mathsf{CC}^{\mathcal{O}_{\mathsf{MPC}},\mathcal{O}_{\mathsf{IZK}}}(1^\lambda) \rightarrow b = 1$.
- **Soundness:** if the data loaded to MPC and IZK is different, $\mathsf{CC}^{\mathcal{O}_{\mathsf{MPC}},\mathcal{O}_{\mathsf{IZK}}}(1^\lambda) \rightarrow b = 0$.
- **Zero-knowledge:** There is a simulator that can simulate the execution of CC without access to $\mathcal{O}_{\mathsf{MPC}}$ and $\mathcal{O}_{\mathsf{IZK}}$.

Construction. $\mathsf{CC}^{\mathcal{O}_{\mathsf{MPC}},\mathcal{O}_{\mathsf{IZK}}}(1^\lambda)$ assumes that both MPC and IZK work in the same field $\mathbb{F}$ and works as follows.

1. Each party $\mathcal{P}_i$ samples a random number r_i for blinding and inputs it to MPC and the $(N-1)$ instances of IZK where $\mathcal{P}_i$ serves as the prover.
2. All parties run a coin toss protocol [424] to obtain a random challenge $\beta \leftarrow\$ \mathbb{F}$.
3. Let $\{x_{\text{MPC}}[i,j]\}_{j=1}^{m_i}$ be the data from $\mathcal{P}_i$ in MPC. Invoke the MPC oracle to compute $r_i + \sum_{j=1}^{m_i} x_{\text{MPC}}[i,j] \cdot \beta^j$ for each party $\mathcal{P}_i$ and reveal the result.

4. Let $\{x_{\text{IZK}}[i,j]\}_{j=1}^{m_i}$ be the data from $\mathcal{P}_i$ in IZK. Invoke the IZK oracle to compute $r_i+\sum_{j=1}^{m_i} x_{\text{IZK}}[i,j]\cdot\beta^j$ for each party $\mathcal{P}_i$ and reveal the result in each of the $(N-1)$ instances of IZK where $\mathcal{P}_i$ serves as the prover.
5. CC outputs $b=1$ if the results about each party's data are all equal, or $b=0$ otherwise.

Security. Completeness is straightforward, so we focus on soundness and zero-knowledge.

- **Soundness:** If the consistency check passes, by DeMillo-Lipton-Schwartz–Zippel lemma [191–193] (which we include in Section 5.8), with probability $1-\max_i(m_i)/|\mathbb{F}|$, the data in MPC and IZK is the same.
- **Zero-knowledge:** Thanks to the random values r_i, all values being opened in CC are masked by r_i. There is a simulator that samples random values as the results of linear combination and invokes the MPC and IZK simulators to simulate the computation.

Cost analysis. The amount of work in MPC is $O(\sum_{i=1}^{N} m_i)$, and, in many MPC protocols where linear combination is a local operation that does not depend on the number of parties N, the wall-clock time is indeed only $O(\sum_{i=1}^{N} m_i)$. The wall-clock time in IZK is $O(N\max_i(m_i))$, and in many IZK protocols, linear combination is concretely very cheap.

5.4 One-dimensional distribution testing

We now provide more details about how IZK and MPC work together in the distribution testing. As a warm-up for multidimensional tests, and sometimes serving as a precondition for multidimensional tests, we first describe one-dimensional tests in this section.

We split the distribution tests into two steps: (1) compute the statistics of the data needed for the tests, which is done in IZK, as described in Section 5.4.1; and (2) compute over the statistics to see if the data passes the distribution tests. If the distribution test is over one party's data, the second step can be done within IZK, but if the distribution test is multiparty, then the second step is done in MPC, as described in Section 5.4.2.

5.4.1 IZK: Verifying the calculation of statistics

HOLMES verifies the calculation of some basic statistics in IZK that involve the input of one party: range, histogram, mean, variance, and trimmed mean. Such statistics can be used for some distribution tests with a public distribution, or be used for multiparty distribution tests described in Section 5.4.2.

Range. To prove that $x\in[a,b]$, the prover $\mathbb{P}$ proves that $0\leq x-a\leq b-a$ and $0\leq b-x\leq b-a$. The standard method for range checks of the form $0\leq x\leq\ell$ is to show that the bit decomposition of x takes $\leq\lceil\log_2\ell\rceil$ bits. We implement a gadget $\mathsf{range}(\langle S\rangle,\mathsf{attr},[a,b])$ that performs a range check on attribute attr for each entry of the population S.

Histogram. A histogram of a population S is generated by counting the number of entries in a set of non-overlapping buckets. The prover $\mathbb{P}$, for each entry, provides a one-hot vector $\vec{v}=(0,0,...,1,...,0,0)$. If $\vec{v}[i]$ is 1, the entry belongs to the i-th bucket. The prover $\mathbb{P}$ first proves that

$\vec{v}$ is one-hot, meaning that it has exactly one "1", and the rest are "0". Next, $\mathbb{P}$ shows that the entry belongs to the i-th bucket via a range check. This can be extended to the multidimensional case. The gadget $\mathsf{histogram}(\langle S\rangle, (\mathsf{attr}_1, ..., \mathsf{attr}_n), ([a_1, b_1], \ldots, [a_D, b_D])) \rightarrow \overrightarrow{\mathsf{count}}$ works with with n attributes $(\mathsf{attr}_1, ..., \mathsf{attr}_n)$ and produces a multidimensional histogram.

Mean and variance. Mean and variance are essential in many tests, such as z-tests and t-tests. To prove $\overline{x}$ is the mean of a population S over an attribute attr, the prover $\mathbb{P}$ shows that $m \cdot \overline{x} \approx \sum_{i=1}^{m} x_i$. In practice, we want to keep a few decimal places for $\overline{x}$ (e.g., $\overline{x} = 12.34$ with two decimal places). This is done by defining $\overline{x}' = 1234$, a fixed-point representation of $\overline{x}$, and $\mathbb{P}$ uses range proofs to show that $m \cdot \overline{x}' \leq 100 \cdot \sum_{i=1}^{m} x_i < m \cdot (\overline{x}' + 1)$. To prove the variance, $\mathbb{P}$ first proves the calculation of the mean $\overline{x}$ and of the mean of the squares $\overline{y}$. The variance can be verified by checking $s^2 \approx \frac{m}{m-1}(\overline{y} - \overline{x}^2)$.[2] We provide two gadgets $\mathsf{mean}(\langle S\rangle, \mathsf{attr}) \rightarrow \overline{x}$ and $\mathsf{variance}(\langle S\rangle, \mathsf{attr}) \rightarrow s^2$.

Trimmed mean. Trimmed mean is similar to mean, but it only considers entries with values within a certain range $[0, \theta]$. This statistic is useful as it can remove extreme values before computing the mean. For each entry x_i in the population, the prover $\mathbb{P}$ first proves the computation of a bit b_i, which indicates whether the value falls in $[0, \theta]$, using range checks. Next, the party $\mathcal{P}$ proves that the number of entries with $b_i = 1$ is n_θ and that the sum of these entries $S_\theta = \sum_{i \text{ s.t. } b_i = 1} x_i$. Finally, $\mathbb{P}$ can show the trimmed mean as S_θ / n_θ. HOLMES implements it in the gadget $\mathsf{trimmedMean}(\langle S\rangle, \mathsf{attr}, \theta) \rightarrow \overline{x}$.

5.4.2 MPC: Finishing touch for multiparty tests

Given the statistics, we can decide if the data passes the distribution tests. If the data is from a single party, this can already be done in IZK. If the data is from multiple parties, we need to perform the final computation in MPC over the statistics, which have been verified by IZK. We call it a "finishing touch" by MPC because the amount of computation is small.

As follows, we discuss how to perform well-known statistical tests: z-test, t-test, F-test, and Pearson's χ^2-test, in the multiparty case in MPC. The single-party case is similar, but the test is run in IZK, compared with a public distribution.

***z*-test.** This distribution test checks whether the mean of two populations S_1 and S_2 (of size m_1, m_2) for the attribute attr are close to each other, when the variances are known. This requires each party to provide the mean, here $\overline{x}_1$ and $\overline{x}_2$. The test checks if:

$$(\overline{x}_1 - \overline{x}_2) / \sqrt{\sigma_1^2/m_1 + \sigma_2^2/m_2} \overset{?}{\leq} T_{\alpha,m_1,m_2},$$

where T_{α,m_1,m_2} is the critical value that is determined by the significance level α, m_1, and m_2.

***t*-test.** This distribution test checks whether the mean of two populations S_1 and S_2 for the attribute attr are close, when the variances are not known. This requires each party to provide the mean, here $\overline{x}_1$ and $\overline{x}_2$, as well as the variance, here s_1^2 and s_2^2. The test checks if:

$$\overline{x}_1 - \overline{x}_2 / \sqrt{s_1^2/m_1 + s_2^2/m_2} \overset{?}{\leq} T_{\alpha,\mathrm{df}}$$

[2] Here, the term $m/(m-1)$ corrects the bias of the variance because $\overline{x}$ is computed from the data [425].

where $T_{\alpha,\mathrm{df}}$ is the critical value determined by the significance level α and the degree of freedom, which, for the t-test, is defined as follows, which involves secret variables s_1^2 and s_2^2.

$$\mathrm{df} = \frac{\left(s_1^2/m_1 + s_2^2/m_2\right)^2}{\left(s_1^2/m_1\right)^2/(m_1-1) + \left(s_2^2/m_2\right)^2/(m_2-1)},$$

This can be computed in MPC with the help of a lookup table. In our implementation, we use a lookup table for T_{df} for df ranging from 1 to 100. When $\mathrm{df} > 100$, T_{df} almost converges, and we can take $T_{\mathrm{df}} = 1.645$.

***F*-test.** This distribution test checks whether the variances of populations S_1 and S_2 for the attribute attr are close to each other. This requires the parties to provide the variances s_1^2 and s_2^2. The test checks if:

$$s_1^2 / s_2^2 \overset{?}{\leq} T_{\alpha,m_1,m_2}$$

where T_{α,m_1,m_2} is determined by the significance level α, m_1, and m_2 and can be computed outside MPC.

Pearson's χ^2-test. This distribution test checks whether the population S (which can be a combination of multiple parties' input data) for the attribute attr follows some public distribution. Given the histogram $\overrightarrow{\mathsf{count}}$ over the attributes attr which has D buckets, the test checks if:

$$\textstyle\sum_{j=1}^{D}(\mathsf{count}[j] - mp_j)^2/(mp_j) \overset{?}{\leq} T_{\alpha,D},$$

where m is the number of entries and p_j is the probability mass for the j-th bucket of the public distribution $\vec{p} = (p_1, \ldots, p_D)$. The critical value $T_{\alpha,D}$ is determined by the significance level α and the number of buckets D.

5.4.3 Subsampling with malicious security

HOLMES supports distribution tests that are performed on a *random* subset of the dataset. Though this sacrifices some accuracy, it boosts the efficiency of individual tests and allows more tests to be performed with a given computational budget. The subsampling must be *maliciously secure*, so that a malicious party cannot retroactively reorder the data to pass the tests. Our approach is to decide this random subset after the data has been loaded to IZK and MPC. The random subset is chosen using a pseudorandom function, with a seed that comes from a coin toss protocol among the N parties [424].

5.4.4 Cost analysis

We now describe the cost of IZK and MPC for the multiparty case. The single-party case is similar with the only difference that IZK does the work of MPC.

IZK. We describe the computation cost of range, histogram, mean, variance, and trimmed mean, as follows.

- **Range:** $O(m\log(b-a))$ to check if $x_1, x_2, ..., x_m \in [a, b]$ for m entries.
- **Histogram:** $O(m(D + \max_{j=1}^{D}\log(b_j - a_j)))$ to put m entries to D buckets, where the j-th bucket contains values in $[a_j, b_j]$. The cost on $\log(b_j - a_j)$ is for the range check to prove that some value x_i belongs to $[a_j, b_j]$.
- **Mean and variance:** $O(m + \log\sum_{i=1}^{m} x_i)$ to compute the mean and/or variance for m entries $x_1, x_2, ..., x_m$. The cost on $\log\sum_{i=1}^{m} x_i$ is for the range check to prove that $m \cdot \overline{x} \approx \sum_{i=1}^{m} x_i$ and/or $s^2 \approx \frac{m}{m-1}(\overline{y} - \overline{x}^2)$, where the approximation relationship is checked by two range checks.
- **Trimmed mean:** $O(m\log\max(b-\theta, \theta-a) + \log\sum_{i=1}^{m} x_i)$ to compute the trimmed mean. The cost on $m\log\max(b-\theta, \theta-a)$ comes from the need to check if an entry x_i is in $[a, \theta]$ or $[\theta, b]$. The cost on $\log\sum_{i=1}^{m} x_i$ is from the computation of the mean of the entries in $[a, \theta]$.

MPC. In the multiparty setting, MPC computes over the statistics that have been verified by IZK, so the cost is often independent from the number of entries in the dataset m, and MPC is doing just a finishing touch, as follows.

- **z-, t-, F-tests:** $O(\log\max_{i=1}^{m} x_i)$ to compare the test statistics with the critical value. The cost $\log\max_{i=1}^{m} x_i$ is due to the overhead of comparison in MPC.
- **Pearson's χ^2-test:** $O(D + \log m)$ to compare the test statistics with the critical value. The cost D is because there are D buckets to count. The count $\log m$ is for comparison.

Note that in the multidimensional case, which we will describe in Section 5.5, the number of buckets is $\prod_{j=1}^{n} D_i$ where D_i is the number of nominal values in the i-th attribute, the cost becomes impractical for MPC and is not just a finishing touch. We will now show how to handle such multidimensional case with very high D using a different algorithm.

5.5 Multidimensional distribution testing

We now discuss a more interesting setting where we want to test the distribution over multiple dimensions. Particularly, we want to test if the distribution of the data is close to a public distribution (e.g., a balanced distribution where different groups are represented appropriately).

Baseline: multidimensional bucketing. Note that we can naturally extend the Pearson's χ^2-test in the one-dimensional case to multidimensional case, by creating multidimensional buckets. Specifically, the distribution test checks whether the population S for the attributes $\mathsf{attr}_1, \ldots, \mathsf{attr}_n$ follows some public distribution. Given the histogram $\overrightarrow{\mathsf{count}}$ over the attributes $(\mathsf{attr}_1, \cdots, \mathsf{attr}_n)$, the test checks if:

$$\sum_{j=1}^{D_1 \times D_2 \times \ldots \times D_n} (\mathsf{count}[j] - mp_j)^2/(mp_j) \overset{?}{\leq} T_{\alpha,D},$$

where m is the number of entries, D_i is the number of unique nominal values in the i-th attribute attr_i, $D = \prod_{i=1}^{n} D_i$ is the number of buckets, and p_j is the probability mass for the j-th bucket of the public distribution $\overrightarrow{p} = (p_1, \ldots, p_D)$. The critical value $T_{\alpha,D}$ is determined by the significance level α and the number of buckets D. We illustrate this test in Figure 5.5.

Cost analysis of the baseline. We now provide more details about how to perform multidimensional bucketing in IZK and MPC, which is to show that the baseline can soon become impractical

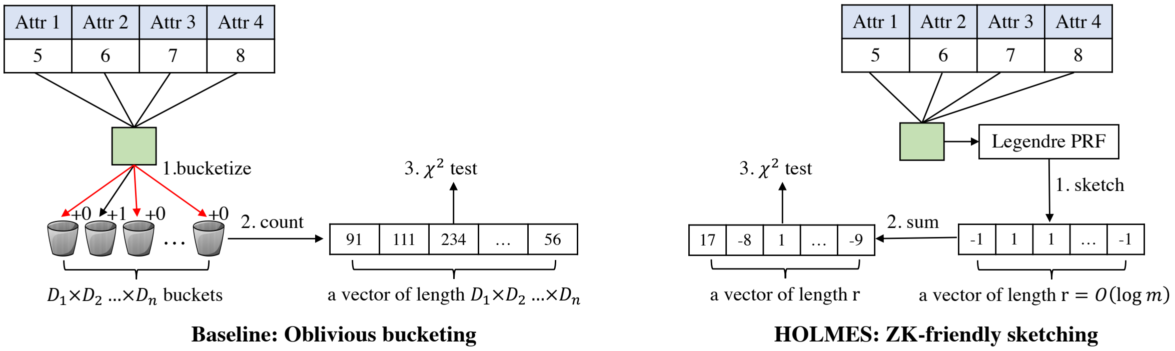


Figure 5.5: Two methods of multidimensional distribution testing.

when the number of buckets D becomes very high. This often happens when there are many dimensions.

- **IZK:** In the IZK, a multidimensional histogram needs to be computed. This histogram has $D = D_1 \times D_2 \times \ldots \times D_n$ buckets. To count how many entries belong to each bucket in an oblivious manner, linear scans are typically necessary, which have a cost of $O(D \log D)$ per entry, where the term $\log D$ is for comparing the index of the bucket.[3] Experiments show that this becomes expensive when D is large.
- **MPC:** The MPC performs the final computation for Pearson's χ^2-test, which involves the histogram $\overrightarrow{\text{count}}$ of length D. Therefore, the cost in MPC is $O(D)$.

The linear growth with respect to D is discouraging. In our experiments in Section 5.6.5, performing distribution testing over four attributes—age, jobs, marital status, and education—results in $D = 37,500$ and takes 10^5 seconds to compute.

Tool: unnormalized χ^2 test. Before we describe HOLMES's solution, we first describe another test for closeness of distributions, called unnormalized χ^2 test by Arias-Castro, Pelletier, and Saligrama [426]. The test has a more complicated critical value, but it removes the division by mp_j from the per-bucket computation, as shown below:

$$\sum_{j=1}^{D_1 \times D_2 \times \ldots \times D_n} (\mathsf{count}[j] - mp_j)^2 \overset{?}{\leq} T_{\alpha,m,p_1,p_2,\ldots,p_D},$$

where the critical value $T_{\alpha,m,p_1,p_2,\ldots,p_D}$ is computed from a variant of the generalized χ^2 distribution with parameters $(mp_1, \ldots, mp_D)$ [427–429]. We provide the statistical details in Section 5.10. Since the parameters are public, $T_{\alpha,m,p_1,p_2,\ldots,p_D}$ can be computed outside MPC.

[3] Another solution is ORAM [**ORAM:goldreich1987towards**, **ORAM:goldreich1996software**, **ORAM:RevisitedPinkas:2010**, **ORAM:Shi:2011**, **ORAM:Stefanov:TowardsPracticalORAM:2012**, **ORAM:gentry2013optimizing**, 186, 187] which has a sublinear complexity $O(\log D)$ per entry. Empirically speaking, ORAM starts to beat the linear scan when D is roughly larger than 2^{20} [102]. Note that initializing the ORAM and reading the counts in the end from ORAM are still needed, and would incur one-time costs of $O(D \log D)$.

HOLMES: ZK-friendly sketching. We find that there is another way to perform this distribution testing with a much lower computational complexity. We now describe its overall workflow, which is illustrated in Figure 5.5. Details will follow in the rest of this section.

- After the data is loaded to IZK and MPC and the consistency check passes, all the parties run a coin toss protocol to sample r different keys for Legendre PRF, denoted by $k_1, k_2, ..., k_r$.
- **Sketching:** The Legendre PRF is applied r times to each entry of the population S, each time with a different key. This produces, for each entry, a vector of length r that consists of elements from $\{-1, 1\}$. Here, $r = O(\log m)$ and does not grow with D (as long as $D << p$, where p is the prime modulus of the field used in IZK and MPC).
- **Summing:** All these length-r vectors are summed together, which is done both in IZK and MPC. IZK can verify the accumulation of these vectors from the prover and verify the single-party sum, where MPC can accumulate the single-party sums from multiple parties, to produce the overall sum, if the population S comes from multiple parties' data.
- χ^2**-test:** Let $\overrightarrow{\mathsf{sum}}$ denote the sum of all length-r vectors. The test can be rewritten as:

  $$\textstyle\sum_{j=1}^{r}(\mathsf{sum}[j] - q[j])^2 \overset{?}{\leq} r \cdot T_{\alpha,m,p_1,p_2,...,p_D},$$

  where vector $\overrightarrow{q}$ is constructed by multiplying a matrix $A \in \mathbb{F}^{r \times D}$ to the length-D vector $\overrightarrow{p} = (mp_1, mp_2, ..., mp_D)$, where $A_{i,j}$ is the result of Legendre PRF with key k_i over input j. This vector $\overrightarrow{q}$ can be computed publicly.

Cost analysis of the new approach. We will show why this sketching algorithm works in the rest of the section, and now we focus on its cost. For a dataset of m entries, the cost in IZK is $O(m \log m)$ for evaluating the Legendre PRF $r = O(\log m)$ times for each entry, and the cost in MPC is $O(r)$. There is some computation outside IZK and MPC, which is to compute $\overrightarrow{p}$, and it has a cost of $O(D \log m)$, including an unavoidable cost to at least parse the public distribution, which already is $O(D \log m)$. In practice, computing $\overrightarrow{p}$ outside IZK and MPC is not the main cost, and can be done beforehand. We will show in our experiments Section 5.6.4 that this new approach can be 10^4 faster in a real-world dataset.

5.5.1 Why sketching works

We now come back to the unnormalized χ^2 test and explain why the ZK-friendly sketching algorithm works. Observe that the left-hand side of the test in unnormalized χ^2 test is indeed the distance of $\overrightarrow{\mathsf{count}}$ and $\overrightarrow{p}$, both of length D. To compute this distance efficiency, it is known from statistics that one can pick a random matrix $A \in \mathbb{F}^{r \times D}$ whose entries are random in $\{-1, 1\}$ [408, 430]. And instead consider the distance between $\overrightarrow{\mathsf{sum}}$ and $\overrightarrow{q}$:

$$\overrightarrow{\mathsf{sum}} := A \cdot \overrightarrow{\mathsf{count}},\ \text{where } \overrightarrow{\mathsf{sum}} \in \mathbb{F}^r \text{ is a short vector}$$
$$\overrightarrow{q} := A \cdot \overrightarrow{p},\ \text{where } \overrightarrow{q} \in \mathbb{F}^r \text{ is a short vector}$$

By the Johnson-Lindenstrauss lemma [408], which we include in Section 5.8, we have:

$$d(\overrightarrow{\mathsf{count}}, \overrightarrow{p}) \approx d(\overrightarrow{\mathsf{sum}}, \overrightarrow{q})/r\ ,$$

for sufficiently large $r = O(\log m)$. Now, our plan is to let IZK verify the computation of $\overrightarrow{\mathsf{sum}}$. The test statistics can be computed in MPC (in the multiparty case) with $O(r)$ given $\overrightarrow{q}$, and $\overrightarrow{q}$ can be computed publicly. However, the cost is still high because verifying $\overrightarrow{\mathsf{count}}$ still takes $O(mD \log D)$.

Rewriting $\overrightarrow{\mathsf{sum}} := A \cdot \overrightarrow{\mathsf{count}}$. To solve this problem, we must avoid the computation of $\overrightarrow{\mathsf{count}}$. Note that $\overrightarrow{\mathsf{count}}$ stores the number of elements in each bucket. This means that we can rewrite $\overrightarrow{\mathsf{sum}} := A \cdot \overrightarrow{\mathsf{count}}$ as follows.

$$\overrightarrow{\mathsf{sum}} = A \cdot \overrightarrow{\mathsf{count}} = A \cdot \textstyle\sum_{i=1}^{m} \overrightarrow{\sigma_i} = \textstyle\sum_{i=1}^{m} A \cdot \overrightarrow{\sigma_i}$$

where $\overrightarrow{\sigma_i}$ is a one-hot vector $(0, 0, ..., 0, 1, 0, ..., 0)$ of length D where the j-th element is "1" if the i-th entry belongs to the j-th bucket, and all the remaining elements are "0". We use $I(i)$ to represent the index of this bucket j for the i-th entry. Note that if we represent the matrix A by its columns,

$$A = (\overrightarrow{a_1}^T \quad \overrightarrow{a_2}^T \quad \cdots \quad \overrightarrow{a_D}^T)$$
$$A \cdot \overrightarrow{\sigma_i} = \overrightarrow{a_{I(i)}}^T$$

where $A \cdot \overrightarrow{\sigma_i}$ is exactly the $I(i)$-th column of A. This allows us to rewrite $\overrightarrow{\mathsf{sum}} := \sum_{i=1}^{m} A \cdot \overrightarrow{\sigma_i}$ as follows:

$$\overrightarrow{\mathsf{sum}} := \textstyle\sum_{i=1}^{m} \overrightarrow{a_{I(i)}} \text{ where } \overrightarrow{a_{I(i)}} \in \mathbb{F}^r$$

Therefore, our problem reduces to, for each entry, finding the corresponding $\overrightarrow{a_{I(i)}}$. This is not efficient in IZK, because for zero-knowledge, IZK must perform computation in an oblivious manner. The lookup of $\overrightarrow{a_{I(i)}}$ from a *random matrix*, which is needed for every entry, is very expensive. If we use ORAM, it takes $O(mr \log D)$ but is concretely expensive. If we use linear scans, it takes $O(mrD)$, which is impractical.[4]

Replacing matrix A with PRF. We now show a very different yet efficient algorithm to perform this oblivious lookup. Note that A only needs to be a random matrix that is independent from the data. One way to generate this matrix is to use a pseudorandom function (PRF) and r different keys generated by a coin toss protocol by the N parties, and we define:

$$\overrightarrow{a_i} := (\text{PRF}_{K_1}(i), \text{PRF}_{K_2}(i), ..., \text{PRF}_{K_r}(i))$$

where the r different keys are K_1, K_2, ..., K_r. This allows us to rewrite $\overrightarrow{\mathsf{sum}} := \sum_{i=1}^{m} \overrightarrow{a_{I(i)}}$ in a way that requires no lookup.

$$\mathsf{sum}[j] := \textstyle\sum_{i=1}^{m} \text{PRF}_{K_j}(\ I(i)\ ) \quad j \in \{1, 2, ..., r\} \ .$$

[4] Readers who are familiar with zero-knowledge proofs may immediately think about polynomial commitment, which can make the lookup verification *holographic*, in that the verifier is very efficient. However, we want to remind the readers that holography does not mean that the prover is efficient. Here, we want to find a solution that the prover's work, even if it is outside IZK, is small. In other words, we need some sort of *double holography* where the prover also does not need to look at matrix A.

which only requires $O(mr)$ computation in IZK, as long as $D << p$ where p is the prime modulus of the field $\mathbb{F}$ used in IZK and MPC. The only remaining question is how to instantiate this PRF efficiently, which we discuss below.

Choice of r. The Johnson-Lindenstrauss lemma does not specify how to choose a concrete r. In practice, r is often chosen empirically. Venkatasubramanian and Wang [431] shows that one can conservatively choose $r = 2\ln m/\epsilon^2$ where ϵ is an error parameter. Our implementation follows this method.

5.5.2 Making the sketching ZK-friendly

A notable concern with using PRF in IZK (or more generally, ZK) is that many commonly used PRFs are very expensive, and therefore, people have been developing ZK-friendly hash functions such as Rescue [410] and Poseidon [409]. We identify that Legendre PRF [411–414] is very suitable for our ZK-friendly sketching.

Recall that in the sketching algorithm, we want to obtain r bits given an index $I(i) \in \{1, ..., D\}$. This is commonly achieved by running a hash function or a PRF and extracting bits from its output. Table 5.2 demonstrates the cost for four functions: SHA-256, Rescue, Poseidon, and Legendre, as well as their best-possible amortized cost by ignoring the ceiling function and dividing the per-entry cost by r.

We now explain where the cost per entry comes from. The number we present is for the case when $|p| = 62$ bits. Other moduli of different bit lengths follow a similar calculation.

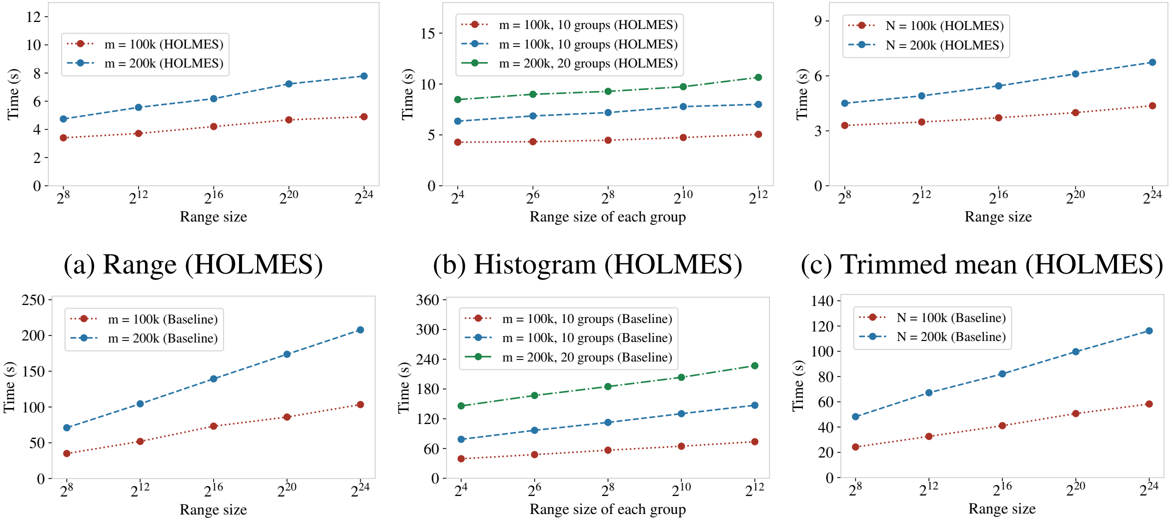
- SHA-256 is often evaluated as a binary circuit in MPC. The state-of-the-art circuit, by Steven Goldfeder [432], takes 22272 AND gates and outputs 256 bits in the end. To obtain r bits, we need to invoke SHA-256 for $\lceil r/256 \rceil$ times.
- We consider Rescue and Poseidon with 128-bit security, a state of length 3, and parameter $\alpha = 5$. Each invocation allows us to squeeze 2 random elements in $\mathbb{F}_p$, the cost of which is 384 and 200 for Rescue and Poseidon respectively. Here is the problem: to obtain unbiased bits, we need to discard the the highest 40 bits of each random element, so we can extract 21 bits from

	Cost per entry	**Amortized by r (best possible)**
SHA-256	$22272 \cdot \lceil r/256 \rceil$	87
Rescue	$384 \cdot \lceil r/42 \rceil + 186 \cdot \lceil r/21 \rceil$	18
Poseidon	$200 \cdot \lceil r/42 \rceil + 186 \cdot \lceil r/21 \rceil$	13.62
Legendre	$8 \cdot r$	8

Table 5.2: Cost per entry for different PRFs when used in our setting where $|p| = 62$ bits, counting the number of input gates and multiplication gates combined. We use hash functions as a PRF by including the key as the prefix of the input, known as domain separation.

	SHA-256	Rescue	Poseidon	Legendre
$r = 10$	22272	570	386	50
$r = 40$	22272	756	572	200

Table 5.3: Concrete costs for different r.



(a) Range (HOLMES) (b) Histogram (HOLMES) (c) Trimmed mean (HOLMES)

(d) Range (Baseline) (e) Histogram (Baseline) (f) Trimmed mean (Baseline)

Figure 5.6: Overhead of basic statistics for HOLMES and for the baseline.

each element. We also need to obtain the *unique* bit decomposition for a number in $\mathbb{F}_p$, which additionally requires a bit-by-bit comparison, and this leads to the cost of 124 per 21 bits. Finally, it also requires an additional input of 64 bits by the prover.

- The Legendre PRF can be seen as a random universal one-way function $f_k(x) = k_0 + \sum_{i=1}^{d} k_i x^i$ of degree d ($d = 3$ in our setting) and a one-bit extractor $b = \left(\frac{f_k(x)}{p}\right) \in \{-1, 1\}$, which is the Legendre symbol. To verify it, the prover inputs two elements and performs six multiplications. We provide the details of Legendre PRF in Section 5.11.

We show the concrete costs for $r = 10$ and $r = 40$ in Table 5.3. In our experiment with a real-world dataset of 41188 records, $r = 40$ is used for accuracy. We can see that Legendre PRF has a lower costs compared with others. To sum up, the use of Legendre PRF makes our sketching algorithm ZK-friendly. Our experiments show that this is up to 10^4 times faster than the baseline based on oblivious bucketing.

5.6 Implementation and evaluation

In this section, we present and discuss the evaluation results of HOLMES, which answer the following questions:

- What is the cost for HOLMES to verify the statistics? (Section 5.6.3)
- How does HOLMES's multidimensional distribution testing compare with the baseline implementation? (Section 5.6.4)
- What is the overhead of HOLMES on real-world datasets? What contributes to this overhead? (Section 5.6.5)

5.6.1 Setup

We ran our experiments on AWS c5.9xlarge instances, each with 36 cores and 72 GB memory. We limit each instance's bandwidth to 2 Gbps and add a round-trip latency of 20 ms. HOLMES and the baseline are implemented using state-of-the-art cryptographic libraries, as follows.

HOLMES. We use QuickSilver [394] for IZK. The version of QuickSilver we use has integrated the latest techniques in Silver [433]. We use SCALE-MAMBA [217] and MP-SPDZ [215, 216] for MPC, where the Low Gear protocol in MP-SPDZ is used for the offline phase of MPC due to its efficiency, and SCALE-MAMBA is used for the online phase. IZK and MPC work on the same prime field $\mathbb{F}_p$ where $p = 2^{62} - 2^{16} + 1$.

Baseline. The baseline runs the same checking algorithm in MPC, using SCALE-MAMBA and MP-SPDZ.

5.6.2 Artifact

We implement all the tools of HOLMES described in the paper, which has been open-sourced anonymously in GitHub. The implementation consists of three parts:

- **Compute engine:** The original QuickSilver is not compatible with many efficient MPC protocol because it works on a very special prime field.[5] We perform an extensive search for prime with low Hamming weight and is compatible with such MPC preprocessing protocols, and we settle down at $p = 2^{62} - 2^{16} + 1$. We contribute a fork of EMP-ZK, called EMP-ZK-HOLMES[6], which includes a highly tuned, specialized implementation for modular reduction and learning-parity-with-noise (LPN) map for this prime.
- **Test gadgets:** We implement and provide gadgets for range, histogram, mean, variance, trimmed mean, z-test, t-test, F-test, and χ^2 tests, including both oblivious bucketing and the ZK-friendly sketching. Integration tests, unit tests, and individual benchmarks are also included in the codebase.[7]

[5] QuickSilver is restricted to a Mersenne prime $p = 2^{61} - 1$. However, this prime is not compatible with MPC preprocessing protocols based on ring learning-with-error (LWE) and would force the MPC to choose preprocessing protocols based on oblivious transfer, which is slower.

[6] `https://github.com/holmes-anonymous-submission/emp-zk`

[7] `https://github.com/holmes-anonymous-submission/holmes-library`

- **Examples and benchmarks:** We assemble distribution tests for three real-world datasets (described in Section 5.6.5) using the gadgets and benchmark their performance. We also include a QuickSilver-to-SCALE-MAMBA source-to-source compiler and an online-only SCALE-MAMBA for ease of benchmark with the baseline.

5.6.3 Cost of statistics verification

We present the overhead for range, histogram, and trimmed mean of HOLMES and compare it with the baseline, as shown in Figure 5.6. All the evaluations below are done with $N = 2$ parties. As we discussed previously, this gap is larger when the number of parties increase.

Range. We start with the basic check that the value is in a range $[a, b]$. As shown in Section 5.5.2 and Section 5.5.2, the overhead grows almost linearly to the number of entries and logarithmically to the size of the range, as expected. HOLMES is about $12\times$ to $19\times$ more efficient than the baseline.

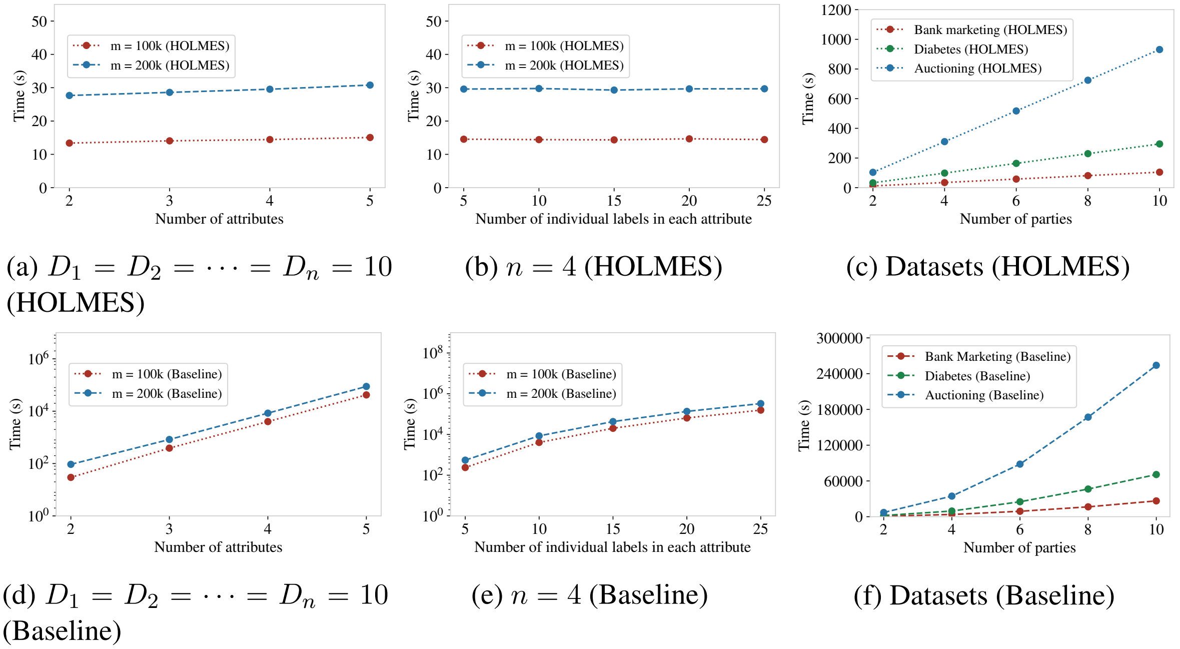
Histograms. We consider histogram over numeric attributes, which arranges data belonging to a range into a bucket (e.g., age $[20, 29]$). We assume each bucket has the same range of the bound. As shown in Section 5.5.2 and Section 5.5.2, the overhead grows linearly to the number of entries, linearly to the number of buckets, and logarithmically to the range size of each group, as expected from the cost analysis in Section 5.4.4. HOLMES is about $5\times$ to $11\times$ more efficient than the baseline.

Trimmed mean. We show the overhead of trimmed mean in Section 5.5.2 and Section 5.5.2. The overhead grows almost linearly to the number of entries and logarithmically to the range size, as expected. HOLMES is about $5\times$ to $10\times$ more efficient than the baseline.

5.6.4 Cost of multidimensional tests

We evaluate our multidimensional distribution testing for a variety of settings. We choose the parameter r with an error rate $\epsilon = 3/4$. As shown in Figure 5.7, HOLMES and the baseline have drastically different growth patterns. As shown in Section 5.6.4 and Section 5.6.4, the sketching approach is almost independent of the number of attributes and the number of different labels in each attribute, but just linear to the number of entries.

We present the overhead of the baseline, drawn in log scale, in Section 5.6.4 and Section 5.6.4. Since the baseline is extremely expensive, we were able to run the experiment only in a small scale; the figures show an extrapolation of our small scale experiments. The overhead grows exponentially to the number of buckets. Our results are consistent with the cost analysis in Section 5.5. In sum, HOLMES's sketching approach improves the efficiency of multidimensional tests for up to $10^4\times$ based in our experiments.



(a) $D_1 = D_2 = \cdots = D_n = 10$ (HOLMES)

(b) $n = 4$ (HOLMES)

(c) Datasets (HOLMES)

(d) $D_1 = D_2 = \cdots = D_n = 10$ (Baseline)

(e) $n = 4$ (Baseline)

(f) Datasets (Baseline)

Figure 5.7: Overhead of the multidimensional distribution testing and the three real-world datasets in HOLMES and in the baseline.

5.6.5 Evaluation on real-world datasets

We apply HOLMES's approach, which is a hybrid of IZK and MPC, and the baseline, which only uses MPC in the verification model, to three real-world datasets and study the overhead. We are interested in how the overhead of HOLMES and baseline changes when the number of parties N changes.

In our experiment, we vary the number of parties from 2 to 10 to see how HOLMES and the baseline scale with more parties. We assume that each party provides the same amount of the data, so when there are N parties, there are $N \cdot m$ data. For example, for the first dataset containing bank marketing data, we assume each party provides $m_{\mathsf{per}} = 41188$ entries of data. When there are 10 parties, the entire distribution testing would be over $m_{\mathsf{per}} \cdot N = 411880$ entries. We choose this approach because in secure collaborative learning with more parties, we expect to have access to more data. For the multidimensional distribution testing in Section 5.6.4 and Section 5.6.4, we use $r = 40$.

Datasets

We evaluate HOLMES and the baseline on three real-world datasets. For each dataset, we choose distribution tests that fit the common use case of such datasets.

Bank marketing. The dataset [417, 418] consists of telemarketing records for financial products.

Number of entries (41188×2)	82376
Number of attributes	21
Total time	12.15 s
Average time per entry	0.15 ms
Loading the data to IZK	0.27 s
Range tests for all attributes	5.63 s
Histogram and χ^2 test on *age*	0.86 s
Multidimensional χ^2 test on *age, job, marital status, education*	4.92 s
Mean, variance, and t test on *call duration*	0.21 s
Consistency check	< 0.01 s

Table 5.4: Breakdown of the cost for the bank marketing dataset (two-party).

Number of entries (101766×2)	203532
Number of attributes	49
Total time	33.43 s
Average time per entry	0.16 ms
Loading the data to IZK	1.59 s
Range tests for all attributes	23.27 s
Histogram and χ^2 test on *medical specialty*	7.59 s
Mean, variance, t test on *number of lab procedures*	0.23 s
Consistency check	< 0.01 s

Table 5.5: Breakdown of the cost for the diabetes dataset (two-party).

Number of entries (567291×2)	1134582
Number of attributes	15
Total time	103.49 s
Average time per entry	0.09 ms
Loading the data to IZK	2.59 s
Range tests for all attributes	92.48 s
Trimmed mean and z test on *total displays*	7.45 s
Consistency check	< 0.01 s

Table 5.6: Breakdown of the cost for the auctioning dataset (two-party).

It includes client profile and call records. A set of banks may want to jointly train a classifier to predict the success of the campaign. Before the training, though, they want to ensure that the dataset has a balanced number of customers from different backgrounds. Therefore, banks can consider the following distribution tests:

- Pearson's χ^2-test over age grouped into the buckets 10–19, 20–29, $\cdots$, 90–99 to ensure that the dataset distribution is similar to the national census age distribution,

- Pearson's χ^2-test over age, job, educational level, and marital status to check if the dataset is balanced across customers with different backgrounds,
- t-test over the telemarketing call duration to check whether their telemarketing records are similar enough to train a model together, and
- range checks for all the attributes.

Diabetes. The dataset [419, 420] consists of admission records for patients with diabetes. It includes of patient profile, treatment, and admission history. A use case is for hospitals to compare the success rate of different treatments. Therefore, hospitals can consider the following checks:

- Pearson's χ^2-test for the medical specialty of initial admissions,
- t-test on the number of lab procedures across hospitals, and
- range checks for all the attributes.

Auctioning. The dataset [421] consists of advertisement biding history. It includes the advertisement location, its expected number of displays, and revenue. Bidders may want to train a model that predict the number of displays. So, bidders can consider the following two tests:

- z-test on the number of displays that are lower than a specific threshold, which checks if the bidding history between different parties is similar, excluding outliers, and
- range checks for all the attributes.

Results

We present our results in Section 5.6.4 and Section 5.6.4. We put them in separate graphs because the overhead is very different. For the three datasets, HOLMES's approach outperforms the baseline by $62\times$ to $272\times$. The gap is larger as the number of parties increases. Recall the complexity analysis in Section 5.1, we know that HOLMES's overhead grows linearly to N, while the baseline overhead grows quadratically to N. The results shown in Section 5.6.4 and Section 5.6.4 confirm this growth patterns.

Breakdown. To understand what contributes to this overhead, we present the breakdown of the overhead in Table 5.4, Table 5.5, and Table 5.6. As shown in the tables, the average time per entry varies between datasets because we perform different tests on them. In all three experiments, we see that the range tests contributes to a large portion of the overhead. In contrast, multidimensional testing is quite efficient. The consistency check between IZK and MPC also has a small overhead.

5.7 Related work

We summarize the related works and explain their connection to HOLMES.

Secure multiparty computation frameworks. A rich body of works propose MPC protocols [82, 207, 434] for malicious adversaries and dishonest majority, with SPDZ [188, 206, 270] and authenticated garbling [435–438] being the state-of-the-art. HOLMES uses SPDZ because it is more suitable for arithmetic computation that is used for secure collaborative learning.

Zero-knowledge proofs. Zero-knowledge proofs [439] enable a party to prove a statement without leaking any information. Recently, constructing practical ZK has gained much attention, especially since succinct non-interactive proofs [440–444] have been used in blockchains. In HOLMES, we require small proving time, so succinctness and non-interactivity are not important. Hence, we use QuickSilver [394] due to the lower prover overhead. As new protocols for interactive zero-knowledge proofs based on silent OT [394–400] are releasing, HOLMES may also change to these newer protocols, such as line-point zero-knowledge (LPZK) [397].

Statistics and range checks. A line of works [445, 446] use MPC techniques for statistical tests. In contrast to our protocol, they mostly focus on the two-party case and consider different threat models. Also, range checks [73, 190, 441] are frequently used to limit the effect of misreported values in secure computation. HOLMES uses range checks as a building block for performing efficiently statistical test. As an example, Prio [73] (or Prio+ [190]) is a system that aggregates statistics over multiple clients who wish to preserve the confidentiality of their individual data but relies on the existence of non-colluding semi-honest servers, whereas HOLMES properly functions even with a dishonest majority.

Secure collaborative computation systems. Multiple works build systems for data analytics and machine learning against malicious adversaries [32, 71, 249–256, 348, 353, 361, 364, 367, 392]. HOLMES contributes to secure collaborative computation systems by providing an efficient method for distribution testing.

Reducing bias in machine learning models. HOLMES is a useful tool for identifying bias in datasets or machine learning models without compromising their privacy. Nevertheless, HOLMES is just a protocol, and it does not specify what distribution tests should be done to mitigate such bias. Robust statistics [447–449] focus on statistics that are resilient to biased input distributions. A fascinating future direction is to augment HOLMES with robust statistics that not only detect bias, but are able to make the model resilient to biased data.

5.8 Statistics lemmas

We include the DeMillo-Lipton-Schwartz–Zippel lemma from [191–193] and Johnson-Lindenstrauss lemma from [408], and we rephrase them for our setting (in a field) as follows.

Lemma 4 (DeMillo-Lipton-Schwartz–Zippel lemma)**.** Let $f(x)$ be a non-zero polynomial of degree n in a prime field $\mathbb{F}$. Pick $\beta \xleftarrow{\$} \mathbb{F}$. Then, we have $\mathbf{Pr}\left[f(\beta) = 0\right] \leq n/|\mathbb{F}|$.

Lemma 5 (Johnson-Lindenstrauss lemma (informal))**.** Let $\overrightarrow{x} \in \mathbb{F}^D$ and let A be a random $r \times D$ matrix that satisfies certain uniformity and normality requirements, then we have:

$$\sum_{j=1}^{D} x_j^2 \approx \sum_{j=1}^{r} ((A\overrightarrow{x})_j)^2 ,$$

where $r = O(\log m)$ and $m = \sum_{j=1}^{n} x_j$.

5.9 Security proof sketches

Theorem 6. *Under standard cryptographic assumptions and static corruptions, and under the random oracle and the ideal cipher models, the protocol of HOLMES securely realizes the ideal functionality $\mathcal{F}$.*

Proof sketch. We now discuss how to prove the security in the $(\mathcal{F}_{\mathsf{IZK}}, \mathcal{F}_{\mathsf{SFE}})$-hybrid world, where an interactive zero-knowledge proof ideal functionality $\mathcal{F}_{\mathsf{IZK}}$ models IZK, and a secure function evaluation ideal functionality $\mathcal{F}_{\mathsf{SFE}}$ models MPC. We also need to invoke a random oracle for subsampling, and/or an ideal cipher for our multidimensional distribution testing to model Legendre PRF. For ease of presentation, we focus on the part involving $(\mathcal{F}_{\mathsf{IZK}}, \mathcal{F}_{\mathsf{SFE}})$ in this proof sketch as follows.

We assume that the adversary corrupts at least one party, otherwise the adversary does not even know the list of statistical tests. For simplicity, we assume malicious parties do not abort in the middle of the protocol. The simulator in the ideal world can simulate the transcripts of HOLMES's protocol, by invoking the corresponding simulators $\mathsf{Sim}_{\mathsf{IZK}}$ and $\mathsf{Sim}_{\mathsf{SFE}}$, on the corresponding computation as well as the consistency check. Note that, during the consistency check, the simulator can easily compute the results at a random point $z = \beta$ sampled by a simulated coin toss. Therefore, the simulator is able to simulate the view of the hybrid world. We show that HOLMES's protocol securely realizes the ideal functionality $\mathcal{F}$ by a hybrid argument. □

5.10 Unnormalized χ^2- test

First, we define the generalized χ^2 distribution, denoted by

$$\tilde{\chi}^2(mp_1, mp_2, \ldots, mp_D) \ .$$

Let $\mathcal{N}(\mu, \sigma^2)$ be the normal distribution with mean μ and variance σ^2, then a sample from $\tilde{\chi}^2(p_1, \ldots, p_D)$ is computed as follows: for $j \in \{1, \ldots, D\}$ sample $Z_j \leftarrow\!\$\, \mathcal{N}(0, p_j(1-p_j))$ and $\mathbb{E}[Z_i \cdot Z_j] = mp_ip_j$, and output $\sum_{j=1}^{n} Z_j^2$. We can compute the critical value based on the public parameters $(p_1, \ldots, p_D)$.

Now, we show that the test passes if and only if the dataset follows the public distribution. The unnormalized χ^2-test says that if the data is close to the public distribution, then

$$T = \textstyle\sum_{j=1}^{n} (\mathsf{count}[j] - mp_j)^2 \xrightarrow{m \to \infty} \tilde{\chi}^2(p_1, \ldots, p_D),$$

otherwise the test statistic T is larger than expected.

When the dataset follows the public distribution, from the central limit theorem $Z_j = \mathsf{count}[j] - mp_j \to \mathcal{N}(0, p_j(1-p_j))$, and as in the χ^2-test case,

$$\begin{aligned}\mathbb{E}[Z_i \cdot Z_j] &= \mathbb{E}[\mathsf{count}[i] \cdot \mathsf{count}[j]] - m^2 p_i p_j \\ &= m(m-1)p_ip_j - m^2p_ip_j = mp_ip_j.\end{aligned}$$

So, the test statistic is close to the expected value. When the dataset does not follow the public distribution, then there is some bucket for which $\mathsf{count}[j] \to mp'_j \neq mp_j$. So, $\mathsf{count}[j] - mp_j \to m(p'_j - p_j) = \infty$, and the test statistic will be larger than expected.

5.11 Legendre PRF

In HOLMES, the multidimensional distribution testing requires the sampling of a random matrix A as described in Section 5.5. We provide details regarding the Legendre PRF used to sample this matrix.

Definition. Let p be an odd prime and $d \geq 2$ be an integer. The degree-d Legendre PRF [411–414] is a family of functions $L_K : \mathbb{Z}_p^* \to \{-1, 1\}$ where $K = (k_0, k_1, ..., k_d) \leftarrow\$ \left(\mathbb{Z}_p^*\right)^d$, defined as $L_K(x) = \left(\frac{f_K(x)}{p}\right)$, where $f_K(x) = k_0 + \sum_{i=1}^{d} k_i x^i$ is an irreducible degree-d polynomial and does not have a nontrivial stabilizer, and $\left(\frac{x}{p}\right) \in \{-1, 0, 1\}$ is the Legendre symbol. For $x \in \mathbb{Z}_p^*$, $L_K(x) \neq 0$ because $f_K(x)$ is irreducible and has degree at least 2.

Verification. Proving the evaluation of the Legendre PRF in IZK is efficient, since to show that $\left(\frac{x}{p}\right) = y \in \{-1, 1\}$, it suffices to provide $a = \sqrt{x} \bmod p$ (if $y = 1$) or $a = \sqrt{b \cdot x} \bmod p$ (if $y = -1$) where b is a quadratic nonresidue. Thus, the prover in IZK proves the following:

- $(y + 1)(y - 1) = 0 \bmod p$, which proves that $y \in \{-1, 1\}$
- $2a^2 = (1 - y) \cdot b \cdot x + (1 + y) \cdot x \bmod p$, which proves that a is the modular square root of $x \bmod p$ (if $y = 1$) or $b \cdot x$ (if $y = -1$)

Sampling the Legendre PRF key. The N parties use the same random matrix, and therefore the same set of PRF keys. Before sampling this random matrix, all parties must have already loaded their input in IZK and MPC. This is to prevent a malicious party from adaptively adjusting their input according to the random matrix.

The parties first use a random coin toss protocol to agree on a freshly generated pseudorandom seed. Then, using this pseudorandom seed as source of randomness, they run the Legendre PRF key generation algorithm for each row of matrix A. The protocol for the Legendre PRF key generation needs to sample a random key $K = (k_0, k_1, ..., k_d)$ such that $f_K(x)$ is irreducible in $\mathbb{Z}_p^*$ and has no nontrivial stabilizer. This requirement has arisen recently [450, 451] as a safeguard against known attacks on Legendre PRF.

The Legendre PRF key generation is performed as follows. First, the parties sample a random degree-d polynomial and check its reducibility using Rabin's irreducibility test [452]. The expected time to find an irreducible polynomial with a probability of about $1/d$ [453, 454]. Even though the testing for nontrivial stabilizer is known to be very inefficient, for $d \geq 3$ a random polynomial over $\mathbb{Z}_p^*$ has nontrivial stabilizers only with a probability at most $9/p$ [451]. Hence, following the recommendations in [451], we avoid nontrivial stabilizers by having $\gcd(p^2 - 1, d) = 1$. For our choice of parameter $d = 3$, this condition is satisfied for any prime larger than or equal to 3.

Pseudorandomness of Legendre PRF. The protocol relies on the pseudorandomness of the Legendre sequence generated by the PRF. Though there is no known reduction to standard cryptographic assumptions, it has been conjectured, like other symmetric key primitives, to have such a property. The pseudorandomness of Legendre PRF has been studied in many different aspects, including linear complexity, collision, avalanche effect, and so on [455–457]. There is a recent trend to study Legendre PRF as a MPC-friendly PRF function [412, 413, 450, 451, 458, 459].

Note that the Johnson-Lindenstrauss lemma does not require a high level of pseudorandomness—hash functions with bounded independence already suffice [460, 461]. We only need to ensure that the functions are chosen independently from the input, which is guaranteed by loading the data in IZK and MPC before sampling the PRF keys.

Recently, there have been several works on key-recovery attacks on Legendre PRF [413, 450, 451], which are not relevant to our settings because our protocol never hides the Legendre PRF keys. Our protocol only uses the Legendre PRF to generate a pseudorandom mapping. Still, in HOLMES, we conservatively choose our parameters to be sufficient to resist key-recovery attacks, so that we have some security margin against future attacks on pseudorandomness. Even with state-of-the-arts attack techniques [462], there are no known PRF distinguishing attacks on the Legendre PRF that are not based on key-recovery attacks [414]. Ethereum foundation currently has an active bounty program [458] on this open problem.

Chapter 6

Reducing Participation Costs via Incremental Verification for Ledger Systems

Ledger systems are applications run on peer-to-peer networks that provide strong integrity guarantees. However, these systems often have high participation costs. For a server to join this network, the bandwidth and computation costs grow linearly with the number of state transitions processed; for a client to interact with a ledger system, it must either maintain the entire ledger system state like a server or trust a server to correctly provide such information. In practice, these substantial costs centralize trust in the hands of the relatively few parties with the resources to maintain the entire ledger system state.

The notion of *incrementally verifiable computation*, introduced by Valiant (TCC '08), has the potential to significantly reduce such participation costs. While prior works have studied incremental verification for basic payment systems, the study of incremental verification for a general class of ledger systems remains in its infancy.

In this paper we initiate a systematic study of incremental verification for ledger systems, including its foundations, implementation, and empirical evaluation. We formulate a cryptographic primitive providing the functionality and security for this setting and then demonstrate how it captures applications with privacy and user-defined computations. We build a system that enables incremental verification for applications such as privacy-preserving payments with universal (application-independent) setup. Finally, we show that incremental verification can reduce participation costs by orders of magnitude for a bare-bones version of Bitcoin.

6.1 Introduction

Ledger systems are applications running across many servers in peer-to-peer networks that offer strong integrity guarantees. Cryptocurrencies, ranging from basic payments to rich smart contracts, are notable examples. The strong integrity guarantees, however, come with high participation costs. In this paper we study how to reduce participation costs in ledger systems.

Participation costs. Servers ("full nodes") in a ledger system maintain the entire application state by performing a new state transition with each new transaction (or batch of transactions in a block), according to some consensus protocol. New servers that join the network must download every transaction and perform each state transition executed since the start of the system. The bandwidth and computation costs increase linearly in the number of transactions and soon become substantial. For example, the Bitcoin ledger is over 300GB; downloading and executing each transaction to reach the latest state can take days, depending on the machine.

Moreover, clients wishing to interact with the application must either keep the entire application state like a server or ask a server to answer queries about the current state (or even past transactions). The first option requires clients to perform computations not relevant to them (e.g., process all payments in the system) and also excludes clients running on weak devices (e.g., smartphones). The second option requires clients to trust that servers answer queries correctly.[1] Both options are undesirable.

Consequently, while in theory ledger systems enable peer-to-peer applications, in practice high participation costs centralize the trust in the hands of a few parties with the resources to maintain the entire application state.

Avoiding re-execution via cryptographic proofs. Several works (reviewed in Section 6.8) have studied systems that leverage cryptographic proofs to avoid re-executing transactions when checking state transitions. Informally, cryptographic proofs enable anyone to produce a short string attesting to the correctness of a computation, which can be verified exponentially faster than the proved computation itself. Using this tool, for each state transition, one can generate a proof of the transition's correctness by referring to two short commitments that summarize the application state before and after the transition. Now, validating the latest state only requires downloading all state commitments and transition proofs (much less data than all transactions) and checking all transition proofs (much less work than re-executing all transactions).

Transition proofs reduce participation costs for both servers and clients, but servers and clients still have to process *every* transition proof. The costs to catch up with the latest state still grow linearly with the number of state transitions that have occurred since the last "synchronization" (or since the start of the system for a new participant). In particular, this is expensive for clients who have spent long periods of time offline.[2]

This idea naturally extends to considering an untrusted operator that produces transition proofs for *batches* of transactions gathered by the operator. This is a popular "layer-2 scaling solution" that has been used in practice with concrete efficiency benefits (see Section 6.8). But the basic idea, as well as the asymptotic complexity, remains essentially the same because batches cannot be too large.

Can one reduce the participation costs further?

Incremental verification. Valiant [465] introduced *incrementally verifiable computation* (IVC) to capture an everlasting computation whose every intermediate state is accompanied by an easy-to-

[1] Simplified Payment Verification (SPV) [463], FlyClient [464], and other light client schemes offer the client some security guarantees without storing the entire application state, but the client still has to trust that the current state is a result of applying transactions correctly.

[2] Additionally using a light client protocol can reduce synchronization costs, but this is at the cost of qualitatively weaker security guarantees, because the client would have to trust that all transition proofs have been verified.

verify proof attesting to its correctness relative to the *initial* state (not the previous state). This capability is achievable via recursive composition of cryptographic proofs, i.e., by producing proofs that (informally) certify both a state transition and the correctness of the prior proof. The exponential speedup of verification relative to execution ensures that the cost of producing each proof from the previous proof does *not* depend on the number of past state transitions.

Incremental verification can dramatically reduce participation costs in ledger systems. Now, validating the latest state requires downloading only the current state (or a short commitment to it) and a *short proof that attests to its correctness relative to the initial state*.[3]

Towards fulfilling the potential. The blockchain community has recognized the potential of incremental verification and has studied it for payment systems built on Nakamoto consensus [466] and on proof-of-stake [467, 468]; the latter has been deployed as a cryptocurrency [469].

However, incremental verification for ledger systems remains in its infancy. First, applications studied so far include simple user-to-user money transfers but not richer applications (e.g., with privacy or smart contracts). Second, advances in cryptographic proofs [470–475] imply that incremental verification can be based on proof systems that are better suited for deployment than those used in [466, 468], namely, proof systems with a *simple and universal* setup. Both directions are suggested as future work in [466, 468] and are being explored by practitioners [476].

Even more fundamentally, as we elaborate in Section 6.1.3, while definitions and constructions of IVC have been studied in detail [297, 477, 478], prior work has only informally discussed the specific needs beyond IVC that arise in ledger systems, and prior work also did not empirically evaluate the benefits and drawbacks of incremental verification with respect to participation costs.

This paper. This paper aims to initiate a systematic study of incremental verification for ledger systems and of its effectiveness in reducing participation costs. We now describe our theoretical contributions (Section 6.1.1) and systems contributions (Section 6.1.2).

6.1.1 Our theoretical contributions

We introduce a cryptographic primitive that captures incremental verification for ledger systems and express several applications within the formalism of this primitive. We elaborate on these two items below.

(i) Incrementally verifiable ledger systems. Valiant's notion of IVC is useful but insufficient for the setting of incremental verification for ledger systems. First, IVC refers to automata computations (arbitrary transitions of a small application state), while applications on ledger systems involve transition functions that, with each transaction, make *few* accesses to a *large* application state. Second, IVC is envisioned for one powerful entity performing state transitions for a long time and then passing its responsibility to another powerful entity (Valiant calls this a "multi-generational computation" [465]); in ledger systems, however, many parties performing state transitions may go offline for periods of time and then, when back online, need to efficiently "catch up" from when

[3] Moreover, incremental verification can be viewed as orthogonal to the "layer-2 scaling solutions" based on batch proofs, and could be used in a hybrid architecture that inherits benefits from both batch proofs and incremental verification.

they left to the latest state. Third, ledger systems include clients who do not wish to store the entire application state; rather, they wish to learn select information by querying servers that store the entire state, with integrity guarantees.

Moreover, definitions proposed by prior work on incremental verification for ledger systems [466, 468] have several limitations, which we discuss in Section 6.3.

To fill these gaps, we introduce a cryptographic primitive for transforming a ledger system specified in a certain formalism into a corresponding *incrementally verifiable ledger system* (IVLS); we call this an *IVLS compiler* and discuss it in Section 6.3. Then in Section 6.4 we explain how the specific interfaces and security properties of our definition let us build peer-to-peer systems that, via incremental verification, achieve low participation costs. An IVLS compiler can be obtained, in a straightforward way, from IVC and collision-resistant hash functions.

(ii) Incrementally verifiable applications. We validate our modeling by showing how to make several applications incrementally verifiable. By carefully designing the applications' states, transactions, and transition functions, we obtain IVLS for five applications: (i) UTXO-based payments ("bare-bones" Bitcoin); (ii) account-based payments, the ledger application studied in prior works on incremental verification [466, 468]; (iii) privacy-preserving payments, á la Zerocash [479]; (iv) privacy-preserving decentralized computation, á la Zexe [480]; and (v) key transparency [481], a popular approach to public-key infrastructure.[4] Our work shows how applications that involve privacy and rich computations are compatible with incremental verification (see Section 6.5 and Section 6.9). This provides solid theoretical foundations that can be used to study the security of new architectures based on recursive proofs that are being explored for Zcash [482]. Subsequent work on key transparency also falls within our IVLS model [483].

6.1.2 Our systems contributions

We build a system prototype in Rust that realizes our IVLS primitive, contributing the following: (1) incremental verification with a universal (application-independent) setup, based on pairings; (2) an incrementally verifiable analogue of Zerocash; and (3) an empirical evaluation that validates the reduction in participation costs via measurements for a ten-year "bare-bones" Bitcoin.

Before elaborating on each of these contributions, we first describe our system at high level. In our Rust library, the programmer specifies the desired application via a transition function represented as a constraint system with read/write memory gates (which our implementation realizes for the application). Our implementation then produces an incrementally verifiable version of that application, creating functionality for producing proofs with each state transition from the prior state and proof, for validating proofs, and so on.

(1) Recursing a universal SNARK based on pairings. Incremental verification is obtained via a recursive use of cryptographic proofs called *succinct non-interactive arguments of knowledge* (SNARKs) [465, 477, 478]. The predominant approach to achieve good efficiency in this setting is based on SNARKs that support "preprocessing" [297, 473]. Informally, this means that the

[4] The key transparency architecture has a central server and multiple clients, meaning that we only consider participation costs for clients, as the single server always remains online.

SNARK verifier can check the satisfiability of a given circuit in time that is exponentially fast by using a short verifying key that was produced in an offline (preprocessing) phase. Prior work on incremental verification [466, 468] uses SNARKs whose preprocessing is part of the SNARK's setup: the circuit to be proved/verified must be fixed once and for all when the public parameters for the SNARK are sampled via a cryptographic ceremony [484–486]. This *circuit-specific setup* is ill-suited for many applications where the circuits to be proved/verified are determined later on by users once the ledger system is already running (e.g., as in Zexe [480]), or where circuits need to be updated.

We give the first demonstration of efficient recursive proofs based on a pairing-based preprocessing SNARK with *universal setup*. Here "universal setup" denotes the desirable property that the SNARK's public parameters do *not* depend on circuits to be proved/verified (but only on an upper bound on their size). As we discuss shortly, this is helpful, or even necessary, for many applications.

Our contribution is to implement and evaluate a (rank-1) constraint system for the verifier of Marlin [470], a state-of-the-art preprocessing SNARK with universal setup and a simple setup ceremony. Our constraint system is less than ten times larger than a constraint system for the state of the art with circuit-specific setup [487], which suffices for building incrementally verifiable ledger systems. We discuss our implementation of the Marlin verifier's constraint system in Section 6.6 and its evaluation in Section 6.7.1.

Other recent works study recursion of preprocessing SNARKs with a universal setup based on other cryptographic tools (cyclic groups [472] or hash functions [473]), but their proof sizes are markedly larger.

(2) Incremental verification for private payments. We provide the first realization of private payments (á la Zerocash [479]) with incremental verification, and evaluate its concrete efficiency. Prior works on incremental verification only considered basic public payments, while we show that incremental verification for more complex applications is not only possible but also practical. Our system prototype makes realizing private payments particularly simple: we only need to program the compliance predicate outlined in Section 6.5 and Section 6.9. We describe our implementation in Section 6.6 and evaluation in Section 6.7.3.

(3) Incremental verification reduces participation costs. We provide an evaluation that quantifies how incremental verification reduces participation costs. Prior work on incremental verification [466, 468] provided only microbenchmarks or network data about the Mina blockchain, without studying how recursive proofs reduce network and computation costs for participants.

We evaluate the effect of IVLS on the participation costs of a ten-year simplified version of Bitcoin (e.g., no scripts): recursive proofs reduce synchronization costs by *orders of magnitude* when compared with not using proofs (that is, naive synchronization) or using per-block transition proofs. Since our code facilitates swapping in different SNARKs over different curves, we evaluate multiple configurations: (a) SNARKs with circuit-specific setup and with universal setup; and (b) different MNT cycles (low-security and high-security levels).

6.1.3 Related work

We summarize prior work on incremental verification for ledger systems; its focus is complementary to ours. We discuss other, less relevant prior work in Section 6.8.

Bonneau et al. [468] (expanding on a prior whitepaper [467]) design an incrementally verifiable payment system based on a proof-of-stake consensus protocol. They show how to modify the Ouroboros Genesis protocol [488] so that its chain selection rule can be realized via a small-space algorithm, thereby obtaining a consensus protocol amenable to incremental state updates.

Kattis and Bonneau [466] design an incrementally verifiable payment system whose consensus protocol requires miners to solve a cryptographic puzzle that updates the prior state's proof to the next state's proof; they call this paradigm a *Proof of Necessary Work*. They show how to modify the Nakamoto consensus protocol to incorporate the proof-computation process to ensure that solving puzzles is "amortization resistant" across solution attempts (a desirable fairness property for mining).

These works focus on basic payments (and no privacy guarantees) and on tackling challenges that arise in making consensus protocols compatible with incremental updates. In contrast, our focus is to formulate and realize a cryptographic primitive with concrete capabilities and security properties and show how it can express a range of applications (including with privacy goals). We believe that this will facilitate further work in incremental verification, which can make black-box use of our primitive. We discuss other limitations of the definitions in [466, 468] in Section 6.3. Further, our implementation achieves incremental verifiability based on recent advances in SNARKs with a simple and universal setup, which are better suited to real-world deployment and not studied previously. We also empirically validate how incremental verification reduces participation costs.

Both [468] and [466] suggest for future work: (1) to explore richer (e.g. privacy-preserving) applications; and (2) SNARKs with universal setup. We address both.

6.2 Participation costs in ledger systems

In this section we elucidate what we mean by "participation costs" in a ledger system and discuss how they are qualitatively affected by no proofs, transition proofs, and recursive proofs. This discussion will motivate the cryptographic primitive in Section 6.3 and its usage in Section 6.4.

A ledger system involves heavyweight *servers* responsible for maintaining the application state and lightweight *clients* that perform queries to the current state (or to consult past transactions). Within this paper, participation costs consist of the following two types of costs.

- **Synchronization costs:** Both servers and clients may go offline and later "catch up" to the current state of the system. (When servers or clients join, they start from the initial state.) Servers and clients verify the current state's integrity by checking that it is the correct result of a sequence of transactions.

- **Query costs:** Clients only store a short state commitment and want to ensure that servers answer their queries correctly relative to the corresponding state. Clients may also make queries to prior application states, as the application may "forget" information that the client still cares about (e.g., a transaction that is already spent in a UTXO-based payment system). If the client only holds a commitment to the current state, it needs assurance that the prior state used to answer that query is in the history of the current application state.

Below we discuss participation costs in ledger systems with: no proofs, transition proofs, and recursive proofs (which underlie incremental verification). Figure 6.1 summarizes how these different options affect the computations required to synchronize; our experiments in Section 6.7.4 confirm this qualitative behavior in practice.

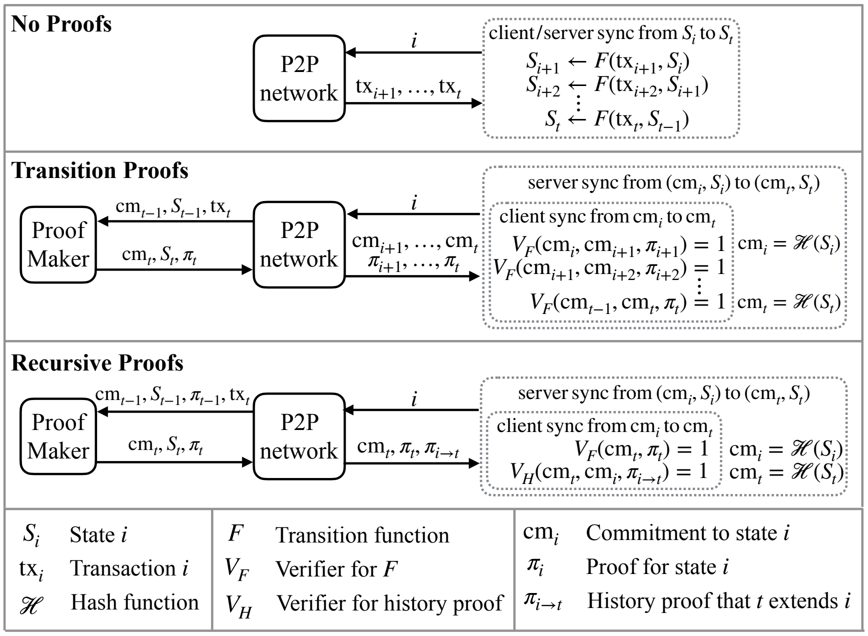

No Proofs
P2P network
i
$\text{tx}_{i+1}, \ldots, \text{tx}_t$
client/server sync from S_i to S_t
$S_{i+1} \leftarrow F(\text{tx}_{i+1}, S_i)$
$S_{i+2} \leftarrow F(\text{tx}_{i+2}, S_{i+1})$
$S_t \leftarrow F(\text{tx}_t, S_{t-1})$
Transition Proofs
Proof Maker
$\text{cm}_{t-1}, S_{t-1}, \text{tx}_t$
cm_t, S_t, π_t
P2P network
i
$\text{cm}_{i+1}, \ldots, \text{cm}_t$
$\pi_{i+1}, \ldots, \pi_t$
server sync from (cm_i, S_i) to (cm_t, S_t)
client sync from cm_i to cm_t
$V_F(\text{cm}_i, \text{cm}_{i+1}, \pi_{i+1}) = 1$
$V_F(\text{cm}_{i+1}, \text{cm}_{i+2}, \pi_{i+2}) = 1$
$V_F(\text{cm}_{t-1}, \text{cm}_t, \pi_t) = 1$
$\text{cm}_i = \mathcal{H}(S_i)$
$\text{cm}_t = \mathcal{H}(S_t)$
Recursive Proofs
Proof Maker
$\text{cm}_{t-1}, S_{t-1}, \pi_{t-1}, \text{tx}_t$
cm_t, S_t, π_t
P2P network
i
$\text{cm}_t, \pi_t, \pi_{i\to t}$
server sync from (cm_i, S_i) to (cm_t, S_t)
client sync from cm_i to cm_t
$V_F(\text{cm}_t, \pi_t) = 1$
$V_H(\text{cm}_t, \text{cm}_i, \pi_{i\to t}) = 1$
$\text{cm}_i = \mathcal{H}(S_i)$
$\text{cm}_t = \mathcal{H}(S_t)$
S_i State i
tx_i Transaction i
$\mathcal{H}$ Hash function
F Transition function
V_F Verifier for F
V_H Verifier for history proof
cm_i Commitment to state i
π_i Proof for state i
$\pi_{i\to t}$ History proof that t extends i

Figure 6.1: Synchronization costs in ledger systems with different proof types (see Section 6.4 for discussion of the proof maker). Proving that one state extends another ($\pi_{i\to t}$) can be made lightweight enough that it can be done by peer-to-peer networks without the proof maker. The server synchronization includes the client synchronization.

No proofs. Without proofs, servers or clients who want to be confident in an application state have no choice but to derive the application state themselves by downloading and executing each transaction since they were last online, leading to high synchronization costs. This essentially removes the distinction between clients and servers, as clients must hold a copy of the current application state to check its validity. The query cost is thus the cost for the client to execute its query over this local copy.

Transition proofs. Instead of executing every state transition, servers or clients can download, for each state transition, a proof and a commitment to the next state. Servers also download the current application state and check that the final commitment commits to the current state. As clients no longer have the entire application state, query costs are different: clients now only use a commitment to the state to verify query responses. Transition proofs are a significant improvement over no proofs (Section 6.7.4), and batching can reduce costs even further [489, 490]. However, synchronization costs remain linear in the number of state transitions and remain substantial for servers and clients that were offline for a long time.

Recursive proofs. Recursive proofs further reduce synchronization costs for servers and clients by removing costs linear in the number of state transitions. Servers and clients only have to verify one proof that the current state is correct and possibly another lightweight proof that the current state extends the prior state held by them. The prover can recursively compose prior proofs to reduce proving cost.[5] Query costs with recursive proofs match those of transition proofs; clients must check that their queries were correctly executed using a commitment to the state.

This informal paradigm is the starting point of our work. In Section 6.3 we formulate a cryptographic primitive that captures these capabilities, and in Section 6.4 we explain how to use the primitive in a system.

6.3 Definition of IVLS

We capture the capabilities of recursive proofs with a cryptographic primitive called an *incrementally verifiable ledger system* (IVLS). An *IVLS compiler* transforms any ledger system into a corresponding IVLS.

We first discuss limitations of prior definitions related to incremental verification for ledgers and then present our set of definitions.

Limitations of prior definitions. Bonneau et al. [468] define the notion of a *succinct blockchain* to capture incremental verification of a blockchain application over a compatible consensus algorithm.[6] While [468] takes a first step in formulating definitions about IVC for ledgers, we revisit these definitions for two reasons.

First, the definitions in [468] do not differentiate between participants that maintain the entire state (full nodes) and participants that maintain only a short commitment to it (light clients). Light clients need to query full nodes for information about the state (as they only have a summary), and may also go offline for a period of time and need to synchronize their old summary with a

[5] We could also imagine generating these proofs non-recursively and proving the correctness of the current state from the initial state after every state transition. This is prohibitively expensive for the prover because the proving cost grows linearly with the number of total transactions.

[6] We use the term *incrementally verifiable ledger system* because: (1) it highlights the main feature (incremental verification) while being consistent with a closely related primitive (incrementally verifiable computation); and (2) "succinct blockchain" is, in our opinion, a misnomer (the blockchain and the application state are not succinct themselves). Also, we prefer the term *ledger system* over *state machine* because the latter equally apply to Valiant's IVC (e.g., it does not suggest a setting with clients and servers).

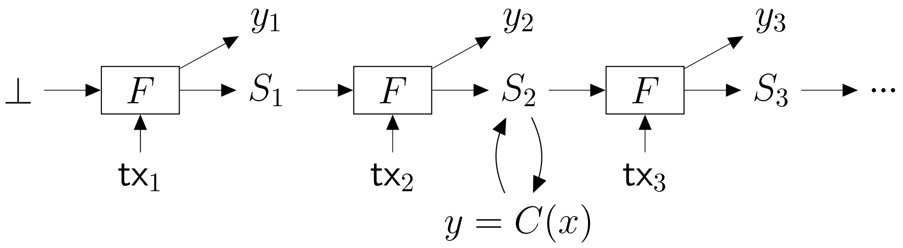
summary of the current state. Our definitions distinguish between these, ensuring that clients can interact with the system in a meaningful way with only a summary of the state.

Second, the definitions in [468] separate the application and the consensus protocol, which complicates interfaces and introduces properties that only relate to consensus (e.g., chain growth). More importantly, this is unnecessary: any consensus protocol compatible with incremental verification can be "folded" without loss of generality into the application itself. Designing consensus protocols compatible with incremental verification is, of course, important but seems better left to the details of the application. For example, our definitions can capture incremental verification not only for traditional blockchain applications (e.g., simple payments and privacy-preserving payments) but also for applications like key transparency that maintain state via a different centralized consensus (see Section 6.5). And, indeed, a subsequent work on verifiable key transparency [483] specifically points out that their work falls under the general framework of IVLS.

Kattis and Bonneau [466] also describe a distributed payment system with incremental verification. They discuss the benefits of incremental verification at a high level, but do not provide interfaces and associated security properties that the system needs to fulfill.

Our definitions. We present our definitions for IVLS; later in Section 6.4 we explain how these abstract definitions apply to real systems and how our properties provide meaningful guarantees to participants, and in Section 6.5 we exercise our definitions to show how to capture several applications of interest, including privacy-preserving payments, privacy-preserving computation, and key transparency (a direction left as future work by [466, 468]).[7]

A *ledger system* is a pair $\mathsf{LS} = (F, C)$ where F is a read-write program called the *transition function* and C is a read-only program called the *client function*.



The transition function F specifies how transactions modify the state. That is, given as input a transaction tx and query access to the current state S, F produces an output y and a state update Δ (denoted by $(y, \Delta) := F^S(\mathsf{tx})$); the new state S' is obtained by applying the update Δ to the old state S (denoted by $S' := S + \Delta$).

The client function C specifies the supported types of client queries over a state: given an input query x and query access to a state S, C produces a query answer y (denoted by $y := C^S(x)$).

For example, in a simple UTXO-based payment system, the application state contains the pool of all unspent transactions, an incoming transaction consumes the outputs of prior transactions to generate new outputs, and the state transition function accepts a new transaction if it consumes

[7] We follow the standard game-based approach, and leave to future work a treatment in the universal composability framework [491].

unspent outputs. The client function searches for unspent transactions that can be spent by a public key. (See Section 6.5 for more examples.)

We wish to transform any ledger system into a ledger system that has the same functionality and is incrementally verifiable. For this, we define an *IVLS compiler*: a tuple $(\mathsf{Setup}, \mathsf{MakeSF}, \mathsf{MakeC}, \mathsf{History})$ that we require to fulfill certain syntax and properties. First, Setup is used to sample public parameters pp; this is a one-time setup that can be used to transform any number of ledger systems. Next, we separately discuss each of $\mathsf{MakeSF}, \mathsf{MakeC}, \mathsf{History}$ in the next three subsections.

Remark 7 (extensions)**.** Our definition of a ledger system assumes for simplicity that the initial state is empty and that the transition function applies one transaction at a time. All discussions in this paper extend, in a straightforward way, to the case of non-empty initial states and to the case of transition functions that apply blocks of transactions. Moreover, we do not consider an algorithm for validating transactions before they are processed by the transition function since the transition function can check validity itself (and if not, return an error in its output and an empty state update). In a real system, validating transactions separately is likely to be more convenient.

Remark 8 (deterministic vs. probabilistic)**.** In the definitions of this section we assume that MakeSF and MakeC are deterministic algorithms. This is the case in our construction (in Section 6.11) and also simplifies many definitions because an adversary can compute $(\mathsf{vS}, \mathsf{vF})$ and vC from F and C, respectively, by itself. All of our definitions naturally extend to the case where MakeSF and MakeC can toss coins: the adversary needs to be given $(\mathsf{vS}, \mathsf{vF})$ and vC as an input at the right place in the definitions.

6.3.1 Properties of MakeSF

Given public parameters pp and a transition function F, MakeSF outputs $(\mathsf{vS}, \mathsf{vF})$ where vF is an incrementally verifiable version of F and vS are functions that manages vF's state. We describe each below.

vS**.** The tuple $\mathsf{vS} = (\mathsf{Info}, \mathsf{VerifyCM}, \mathsf{VerifyAll})$ consists of efficient deterministic algorithms working as follows:

- $\mathsf{vS}.\mathsf{Info}^{S,A}() \to (t, \mathsf{cm}, \pi_{\mathsf{F}})$. Given oracle access to a state S and auxiliary state A, Info outputs the state's current index t, commitment cm, and proof π_{F}.
- $\mathsf{vS}.\mathsf{VerifyCM}(S, \mathsf{cm}) \to b$. On input a state S and commitment cm, $\mathsf{VerifyCM}$ determines if cm is a valid commitment to S. We use the convention $\mathsf{vS}.\mathsf{VerifyCM}(\perp, \perp) = 1$ (i.e., the empty commitment is a valid commitment for the empty state).
- $\mathsf{vS}.\mathsf{VerifyAll}(S, A) \to b$. On input a state S and auxiliary state A, $\mathsf{VerifyAll}$ checks if the states are valid. We use the convention $\mathsf{vS}.\mathsf{VerifyAll}(\perp, \perp) = 1$.

For now, the only property that we explicitly require is that it is infeasible to find distinct states with the same valid commitment (Definition 9). Definitions later in this section imply other properties of vS.

Definition 9 (binding of vS)**.** For every polynomial-size adversary $\mathcal{A}$ and sufficiently large security parameter λ,

$$\Pr\left[\begin{array}{c} \mathsf{vS.VerifyCM}(S_1, \mathsf{cm}) = 1 \\ \mathsf{vS.VerifyCM}(S_2, \mathsf{cm}) = 1 \\ \downarrow \\ S_1 = S_2 \end{array}\middle|\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (F, S_1, S_2, \mathsf{cm}) \leftarrow \mathcal{A}(\mathsf{pp}) \\ (\mathsf{vS}, \mathsf{vF}) \leftarrow \mathsf{MakeSF}(\mathsf{pp}, F) \end{array}\right] \geq 1 - \mathrm{negl}(\lambda) \ .$$

vF**.** The tuple $\mathsf{vF} = (\mathsf{Run}, \mathsf{Verify})$ consists of efficient algorithms working as follows:

- $\mathsf{vF.Run}^{S,A}(\mathsf{tx}) \to (y, \Delta)$. Given oracle access to a state S and auxiliary state A, and given as input a transaction tx, Run outputs a result y and a state update $\Delta = (\Delta_S, \Delta_A)$ to modify S and A. The result y and state update Δ_S equal the output of the original transition function F (on input tx and state S).

- $\mathsf{vF.Verify}(t, \mathsf{cm}, \pi_\mathsf{F}) \to b$. On input a state's index t, commitment cm, and proof π_F, Verify decides if the commitment is consistent with a state resulting from applying the transition function F on t transactions. We will assume $\mathsf{vF.Verify}(0, \perp, \perp) = 1$.

We require two properties. *Completeness* (Definition 10) states that using vF.Run to apply a transaction to a valid state yields a new valid state that is consistent with the original transition function F and a new auxiliary state that contains information passing all the relevant checks. *Knowledge soundness* (Definition 11) states that if an efficient adversary outputs a state commitment and proof that are valid according to vF.Verify, then an efficient extractor can output a sequence of transactions that, via transition function F, lead to a state with the claimed commitment.

Definition 10 (completeness of vF)**.** For every polynomial-size adversary $\mathcal{A}$ and security parameter λ,

$$\Pr\left[\begin{array}{c} \mathsf{vS.VerifyAll}(S, A) = 1 \\ \downarrow \\ \mathsf{vF.Verify}(t, \mathsf{cm}, \pi_\mathsf{F}) = 1 \\ (y, \Delta_S) = F^S(\mathsf{tx}) \\ t' = t + 1 \\ \mathsf{vS.VerifyAll}(S', A') = 1 \end{array}\middle|\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (F, S, A, \mathsf{tx}) \leftarrow \mathcal{A}(\mathsf{pp}) \\ (\mathsf{vS}, \mathsf{vF}) \leftarrow \mathsf{MakeSF}(\mathsf{pp}, F) \\ (t, \mathsf{cm}, \pi_\mathsf{F}) \leftarrow \mathsf{vS.Info}^{S,A}() \\ (y, (\Delta_S, \Delta_A)) \leftarrow \mathsf{vF.Run}^{S,A}(\mathsf{tx}) \\ S' := S + \Delta_S,\ A' := A + \Delta_A \\ (t', \mathsf{cm}', \pi'_\mathsf{F}) \leftarrow \mathsf{vS.Info}^{S',A'}() \end{array}\right] = 1 \ .$$

Definition 11 (knowledge soundness of vF)**.** For every polynomial-size adversary $\mathcal{A}$ there exists a polynomial-size extractor $\mathcal{E}$ such that for every sufficiently large security parameter λ,

$$\Pr\left[\begin{array}{c} \mathsf{vF.Verify}(t, \mathsf{cm}, \pi_\mathsf{F}) = 1 \\ \downarrow \\ S = F(\mathsf{tx}_1, \ldots, \mathsf{tx}_t) \\ \mathsf{vS.VerifyCM}(S, \mathsf{cm}) = 1 \end{array}\middle|\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (F, t, \mathsf{cm}, \pi_\mathsf{F}) \leftarrow \mathcal{A}(\mathsf{pp}) \\ (\mathsf{tx}_1, \ldots, \mathsf{tx}_t) \leftarrow \mathcal{E}(\mathsf{pp}) \\ (\mathsf{vS}, \mathsf{vF}) \leftarrow \mathsf{MakeSF}(\mathsf{pp}, F) \end{array}\right] \geq 1 - \mathrm{negl}(\lambda) \ .$$

6.3.2 Properties of MakeC

Given public parameters pp and a client function C, MakeC outputs a tuple $\mathsf{vC} = (\mathsf{Run}, \mathsf{Verify})$, the verifiable version of C, that enables proving/verifying that, given a commitment to the current state, the client function was executed correctly. In more detail, these work as follows:

- $\mathsf{vC.Run}^{S,A}(x) \to (y, \pi_{\mathsf{C}})$. Given oracle access to a state S and auxiliary state A and input x, Run produces an output y and a proof π_{C} attesting that $y = C^S(x)$.

- $\mathsf{vC.Verify}(\mathsf{cm}, x, y, \pi_{\mathsf{C}}) \to b$. On input a state's commitment cm, input x, claimed output y, and proof π_{C}, Verify determines if cm is consistent with a state S such that $y = C^S(x)$.

We require two properties. *Completeness* (Definition 12) states that outputs of $\mathsf{vC.Run}$ are accepted by $\mathsf{vC.Verify}$ and are consistent with executions of the given client function C. *Soundness* (Definition 13) states that if an efficient adversary outputs (a valid state and auxiliary state and) a proof that is accepted by $\mathsf{vC.Verify}$ for a given input and claimed output, then the claimed output equals C's output.

Definition 12 (completeness of vC)**.** For every polynomial-size adversary $\mathcal{A}$ and security parameter λ,

$$\Pr\left[\begin{array}{c|r} & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{vS.VerifyAll}(S, A) = 1 & (F, C, S, A, x) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \downarrow & (\mathsf{vS}, \mathsf{vF}) \leftarrow \mathsf{MakeSF}(\mathsf{pp}, F) \\ y = C^S(x) & \mathsf{vC} \leftarrow \mathsf{MakeC}(\mathsf{pp}, C) \\ \mathsf{vC.Verify}(\mathsf{cm}, x, y, \pi_{\mathsf{C}}) = 1 & (y, \pi_{\mathsf{C}}) \leftarrow \mathsf{vC.Run}^{S,A}(x) \\ & (\cdot, \mathsf{cm}, \cdot) \leftarrow \mathsf{vS.Info}^{S,A}() \end{array}\right] = 1 \ .$$

Definition 13 (soundness of vC)**.** For every polynomial-size adversary $\mathcal{A}$ and sufficiently large security parameter λ,

$$\Pr\left[\begin{array}{c|r} \mathsf{vS.VerifyAll}(S, A) = 1 & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{vC.Verify}(\mathsf{cm}, x, y, \pi_{\mathsf{C}}) = 1 & (F, C, S, A, x, y, \pi_{\mathsf{C}}) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \downarrow & (\mathsf{vS}, \mathsf{vF}) \leftarrow \mathsf{MakeSF}(\mathsf{pp}, F) \\ y = C^S(x) & \mathsf{vC} \leftarrow \mathsf{MakeC}(\mathsf{pp}, C) \\ & (\cdot, \mathsf{cm}, \cdot) \leftarrow \mathsf{vS.Info}^{S,A}() \end{array}\right] \geq 1 - \mathrm{negl}(\lambda) \ .$$

6.3.3 Properties of $\mathsf{History}$

$\mathsf{History} = (\mathsf{Prove}, \mathsf{Verify})$ enables proving/verifying that a prior state commitment is on the same timeline as another (later) state commitment. $\mathsf{History.Prove}$ produces a proof that the current (committed) state can be reached from an earlier (committed) state. $\mathsf{History.Verify}$ checks the proof. In more detail, the algorithms work as follows:

- $\mathsf{History.Prove}^{S,A}(\mathsf{pp}, t) \to \pi_{\mathsf{H}}$. Given oracle access to a state S and auxiliary state A, and given as input public parameters pp and a previous state's index t, Prove outputs a proof π_{H} attesting to the relationship between the t-th state commitment in the history and the current state commitment. (If the current state has an index less than or equal to t, Prove outputs $\perp$.)

- $\mathsf{History.Verify}(\mathsf{pp}, \mathsf{cm}, t, \mathsf{cm}_t, \pi_{\mathsf{H}}) \to b$. On input public parameters pp, the current state's commitment cm, previous state's index t and commitment cm_t, and a proof π_{H}, Verify determines if cm_t was the valid t-th state commitment in the history leading to cm.

We require two properties. *Completeness* (Definition 14) states that valid proofs can be generated for any past prior state commitment, relative to the state's index. *Binding* (Definition 15) states that it is infeasible to find, for the same state commitment cm, two distinct commitments that are valid for the same prior index t.

Definition 14 (completeness of $\mathsf{History}$)**.** For every polynomial-size adversary $\mathcal{A}$ and security parameter λ,

$$\Pr\left[\begin{array}{c|r} & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{vS.VerifyAll}(S, A) = 1 & (F, S, A, (\mathsf{tx}_1, \dots, \mathsf{tx}_n)) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \downarrow & (\mathsf{vS}, \mathsf{vF}) \leftarrow \mathsf{MakeSF}(\mathsf{pp}, F) \\ t' = t + n & (t, \mathsf{cm}, \cdot) \leftarrow \mathsf{vS.Info}^{S,A}() \\ \mathsf{History.Verify}(\mathsf{pp}, \mathsf{cm}', t, \mathsf{cm}, \pi_{\mathsf{H}}) = 1 & (S', A') \xleftarrow{\mathsf{vF.Run}(\mathsf{tx}_1, \dots, \mathsf{tx}_n)} (S, A) \\ & (t', \mathsf{cm}', \cdot) \leftarrow \mathsf{vS.Info}^{S',A'}() \\ & \pi_{\mathsf{H}} \leftarrow \mathsf{History.Prove}^{S',A'}(\mathsf{pp}, t) \end{array}\right] = 1 \ .$$

Above we use a shorthand for going from state (S, A) to state (S', A') via the transactions $(\mathsf{tx}_1, \dots, \mathsf{tx}_n)$.

Definition 15 (binding of $\mathsf{History}$)**.** For every polynomial-size adversary $\mathcal{A}$ and sufficiently large security parameter λ,

$$\Pr\left[\begin{array}{c|r} \mathsf{History.Verify}(\mathsf{pp}, \mathsf{cm}, t, \mathsf{cm}_t, \pi_{\mathsf{H}}) = 1 & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{History.Verify}(\mathsf{pp}, \mathsf{cm}, t, \mathsf{cm}'_t, \pi'_{\mathsf{H}}) = 1 & \left(\mathsf{cm}, t, \begin{array}{c} \mathsf{cm}_t, \pi_{\mathsf{H}} \\ \mathsf{cm}'_t, \pi'_{\mathsf{H}} \end{array}\right) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \downarrow & \\ \mathsf{cm}_t = \mathsf{cm}'_t & \end{array}\right] \geq 1 - \operatorname{negl}(\lambda) \ .$$

6.4 System architecture

We described how an IVLS compiler transforms a given ledger system $\mathsf{LS} = (F, C)$ into a new ledger system $\mathsf{IVLS} = (\mathsf{vF}, \mathsf{vC})$ that (a) has the same functionality and similar efficiency and (b) is incrementally verifiable in a precise sense. Next we describe how IVLS gives rise to a peer-to-peer architecture with (much) smaller participation costs. In Section 6.10.1 we explain how consensus protocols integrate with this system architecture, and in Section 6.10.2 we discuss how privacy fits into this system architecture. Recall from Section 6.2 that synchronization costs are incurred by servers and clients who want to verify that the current state is derived from a past state correctly, and query costs are incurred by clients who want to verify query responses.

The new system. Informally, servers use the new transition function vF instead of the original F, and clients use the new client function vC instead of the original C. We now discuss the different operations.

State updates. Besides the application state S, each server maintains an auxiliary state A that stores cryptographic information. A server processes a new transaction tx by computing $(y, \Delta) := \mathsf{vF.Run}^{S,A}(\mathsf{tx})$ and applying the state update Δ to the augmented state (S, A). (For comparison, in the original application, each server would compute $(y, \Delta_S) = F^S(\mathsf{tx})$ and apply the state update Δ_S to S.) Any server can run $\mathsf{vS.Info}^{S,A}$ to obtain the number t of transactions that have been applied since the start of the system, a commitment cm to the current application state, and a proof π_{F} of correctness.

State validity. A server joining the system can simply download and verify the state and auxiliary information (S, A) by checking the state proof π_{F} and checking the consistency between S and A using $\mathsf{vS.VerifyAll}(S, A)$. The new server does not need to itself apply the transition function $\mathsf{vF.Run}$ for each prior transaction.

Client queries. vC enables a server to convince a client that it answered the client's query correctly. The client sends a query x to a server, the server computes and returns the answer and proof $(y, \pi_{\mathsf{C}}) \leftarrow \mathsf{vC.Run}^{S,A}(x)$, and the client checks the answer by running $\mathsf{vC.Verify}(\mathsf{cm}, x, y, \pi_{\mathsf{C}})$, where cm is the commitment to S.

Relation between states. A server or client may be offline for a period of time and need a way to establish that an old state commitment $\mathsf{cm}^{\mathsf{old}}$ and a new state commitment $\mathsf{cm}^{\mathsf{new}}$ are not just individually valid (as established via $\mathsf{vF.Verify}$) but also belong to the same "timeline". Moreover, a client that has a current state commitment and wants to make a query about an old state needs to be able to check that the current state was derived from the old state. Both tasks can be accomplished via functionality offered by $\mathsf{History}$, which enables skipping along this "timeline" of states using only commitments. The server can prove that the state committed to by $\mathsf{cm}^{\mathsf{new}}$ is reachable from a state committed to by $\mathsf{cm}^{\mathsf{old}}$ by computing a history proof $\pi_{\mathsf{H}} := \mathsf{History.Prove}^{S,A}(\mathsf{pp}, t^{\mathsf{old}})$ where t^{old} is the old state's index. The client checks this proof by running $\mathsf{History.Verify}(\mathsf{pp}, \mathsf{cm}^{\mathsf{new}}, t^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}}, \pi_{\mathsf{H}})$.

Motivation for security properties. We discuss how the security properties presented in Section 6.3 translate to the informal security desiderata we have discussed for synchronizing state and executing queries.

Synchronization. vF's knowledge soundness (Definition 11) states that every state commitment accepted by $\mathsf{vF.Verify}$ implies a sequence of transactions that, via the transition function F, lead to the committed state. The binding property of $\mathsf{History}$ (Definition 15) states that one cannot find prior state commitments that contradict each other but are both accepted by $\mathsf{History.Verify}$ for the current state commitment. Together these imply that the state committed to by prior state commitments is valid.

Query execution. vC's soundness (Definition 13) ensures that every query result over a valid state accepted by $\mathsf{vC.Verify}$ implies that the result y is the output of running the client function C over the state with the input x.

Reducing participation costs. The new system reduces synchronization costs and query costs.

Synchronization now only requires servers and clients to verify the latest state proof π_{F} produced by $\mathsf{vF.Run}$ along with, if they have a past commitment, a history proof π_{H} produced by $\mathsf{History.Prove}$. To verify the consistency between the state and the auxiliary state, the server additionally runs $\mathsf{vS.VerifyAll}$. These are efficient checks, and our evaluation of synchronization costs shows their practical benefits (see Section 6.7.4).

Query costs are simply the time for the client to verify the query proof π_{C} produced by $\mathsf{vC.Run}$, which is much faster than executing the query itself on the state.

Integration with consensus protocols. For consensus-based ledger systems, the application's transition function F should be tasked with maintaining consensus information; in particular, the consensus protocol should be compatible with incremental verification. (Otherwise, the overall system would not be incrementally verifiable.) Prior works [466, 468] have designed consensus protocols for incremental verification, which can be used here.

Who produces the proofs?. Introducing cryptographic proofs into a real-world system raises the question of who is responsible for producing these proofs. We briefly summarize two approaches described in prior work, which can be used in IVLS. Mina [468] introduces a new party that makes proofs (a "snarker") and is monetarily incentivized to produce proofs: block producers ask snarkers to produce proofs for generated blocks, and the two parties agree on a fee in what is essentially a lowest-price auction. Another approach is to directly embed producing proofs into the cryptographic PoW puzzle, known as Proof of Necessary Work [466], which requires the proof-computing process to satisfy *amortization resistance* (a property plausibly satisfied by known pairing-based SNARKs on NP statements, incorporating nonces). We discuss and evaluate proving costs in Section 6.7.5.

6.5 Applications

We describe how the formalism of ledger systems (see Section 6.3) can capture several applications of interest.[8] By applying an IVLS compiler to these applications, one can make the application incrementally verifiable, and hence significantly reduce participation costs for it. Overall, we show that IVLS supports not only basic payment systems (the only ones studied in prior works on incremental verifiability) but also systems with strong privacy or rich user-defined applications (e.g., smart contracts).

We outline how to "program" ledger systems to express each application, with a focus on transition functions; for completeness, we provide formal details (and a discussion of client functions)

[8] As we discuss in Section 6.3, the application must take consensus into account, where the consensus must be incrementally verifiable. Since a consensus protocol can be viewed as an algorithm that can be folded into the application, by suitably augmenting the application state and transition function, we do not discuss the details of consensus in this section. We discuss integration with consensus protocols in more detail in Section 6.10.1.

in Section 6.9. We adopt two design principles for efficiency: (a) each transaction results in a small number of reads/writes to the application state; (b) the application state contains the minimum information necessary for ensuring the application's integrity. (Users are responsible for storing any information specific to them.)

Basic payments. As a warmup, we first discuss basic payments (user-to-user transfers without privacy guarantees), in the account-based and UTXO-based models.

In the account-based model (studied in [466, 468]), the application state is a map from public keys to balances. The transition function takes as input a signed message, under the sender's public key, specifying the amount to be paid and the receiver's public key. To prevent replaying a prior transaction, the application state maintains a counter for each public key, which increases with each payment; the signed message includes this counter. The application state's size is linear in the number of public keys.

In the UTXO-based model (think bare-bones Bitcoin with no scripts), the state maps public keys to identifier-value pairs each representing a coin. The transition function takes as input a sender public key; a list of existing coin identifiers all owned by the sender; and information for receivers in the form of their public keys, new coin identifiers, and their values (whose total equals that of the sender's coins). All of this is signed under the sender's public key. The identifiers of the spent coins are removed from the application state.

Privacy-preserving payments. We discuss how to express payments with user privacy, as in Zerocash [479]: user-to-user payments that reveal no information about the sender, receiver, or transferred amount. While superficially such a system looks very different from other payment systems, modeling it as a ledger system is not difficult, as explained below. Recall that each Zerocash transaction contains the serial numbers of old (spent) coins and the commitments to new (created) coins along with a zero-knowledge proof attesting that the old coins were created at some point in the past and now have been spent by someone who knows their private keys, and that the new coins were committed correctly and preserve the monetary value of the old coins.

The application state includes a list of all serial numbers and a list of all coin commitments. (More precisely, the serial numbers are stored in a search tree and the coin commitments in a Merkle tree.) The transition function validates a transaction by checking its zero-knowledge proof and checking that its serial numbers do not already appear in the list of all serial numbers. If so, the transition function adds the transaction's serial numbers and coin commitments to the application state. Crucially, the transition function here only makes a few accesses to the application state.

Clients must identify transactions where they are recipients. Naively, this requires a linear scan over all transactions. This linear cost can be avoided via *viewing keys*, which allow the server to identify transactions relevant to the client without the server being able to spend the client's funds. Viewing keys protect the clients' funds but not privacy. To achieve privacy from the server, light clients can leverage prior work using secure hardware [492–494] and/or private information retrieval (PIR) [495], which can be directly applied to our setting.

We discuss the implementation and evaluation of incrementally verifiable privacy-preserving payments in Section 6.6 and Section 6.7.3 respectively.

Privacy-preserving computation. Zexe [480] extends Zerocash to support privacy-preserving general computation, as captured via a computation model that involves data units called *records*, which contain scripts for how they can be created or consumed. Analogously, we extend the previous design to privacy-preserving computation, by setting the application state to be a list of all serial numbers and a list of all record commitments. Each transaction contains the serial numbers of old (consumed) records and the commitments to new (created) records along with a zero-knowledge proof attesting that scripts contained in all the records were correctly executed. The transition function validates a transaction by checking its proof and by checking that its serial numbers do not already appear in the list of all serial numbers; if so, it adds the serial numbers and record commitments in the transaction to the application state.

Key transparency. We conclude with an application beyond cryptocurrencies: key transparency [481, 496], a public directory mapping usernames to public keys. Unlike previous applications, in key transparency, a central server maintains the application state, and other parties verify that it does so correctly. Users can publish their own public keys to a directory maintained by the central server, query other users' public keys, and check that the directory maintains this mapping correctly.

The application state is pairs of usernames and public keys. The transition function processes two types of transactions: in the first type, a user can register a key by sending a new username and public key; in the second type, a user can update an existing public key with a signature of the new public key and the username under the old public key. While server participation costs are not a concern (application state is maintained by a single central server), incremental verifiability does reduce client participation costs. In the original design, a user must regularly query the central server to check that it continuously maintains the mapping between their username and public key. In our incrementally verifiable design, users only need to check a single proof that the *entire* application state has been maintained correctly.

6.6 Construction and implementation

We implement the IVLS compiler in Rust. The main components are summarized below and are illustrated in Figure 6.2. Several components are of independent interest, as they simplify the use of recursive SNARKs in many settings. For example, we provide a generic implementation of proof-carrying data and a constraint system encoding the correct execution of the verifier of a state-of-the-art pairing-based SNARK with *universal* setup. We used and contributed to the constraint-writing framework of the `arkworks` [497] library (formerly `libzexe`) and its algebra libraries for finite fields and elliptic curves.

Our implementation is open-sourced under the Apache v2 license or the MIT license and is available online. [9] Our code base has been extended in subsequent work to support new types of

[9] We contributed our code to the `arkworks` library.

– IVLS: `https://github.com/arkworks-rs/ivls`
– PCD: `https://github.com/arkworks-rs/pcd`
– Non-native field arithmetic: `https://github.com/arkworks-rs/nonnative`
– Constraints of Marlin: `https://github.com/arkworks-rs/marlin`
– Constraints of Marlin's polynomial commitments: `https://github.com/arkworks-rs/poly-commit`

recursion [472, 474, 475].

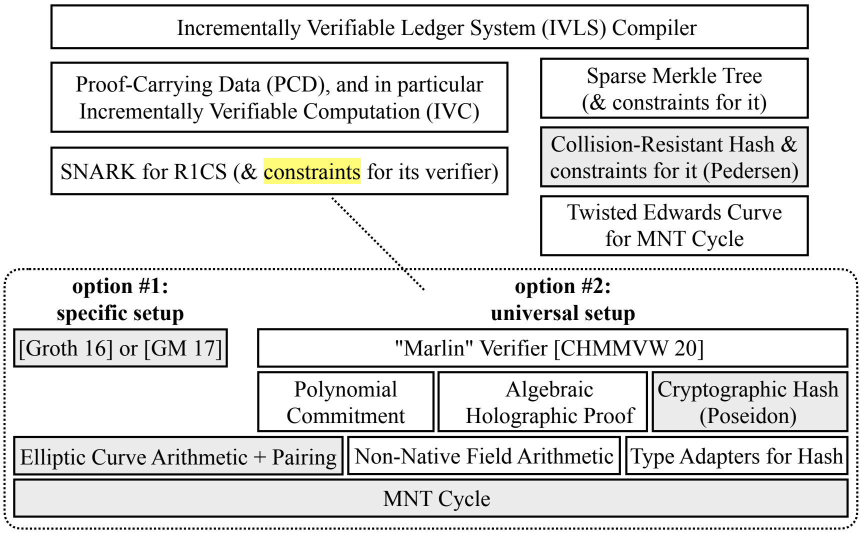

Incrementally Verifiable Ledger System (IVLS) Compiler
Proof-Carrying Data (PCD), and in particular Incrementally Verifiable Computation (IVC)
Sparse Merkle Tree (& constraints for it)
Collision-Resistant Hash & constraints for it (Pedersen)
SNARK for R1CS (& constraints for its verifier)
Twisted Edwards Curve for MNT Cycle
option #1: specific setup
option #2: universal setup
[Groth 16] or [GM 17]
"Marlin" Verifier [CHMMVW 20]
Polynomial Commitment
Algebraic Holographic Proof
Cryptographic Hash (Poseidon)
Elliptic Curve Arithmetic + Pairing
Non-Native Field Arithmetic
Type Adapters for Hash
MNT Cycle

Figure 6.2: Diagram illustrating the relation between different components of our system. The gray boxes denote components that exist in prior libraries, while the white boxes denote components contributed in this work.

IVLS. The top-level interface is a collection of traits that closely follow the interface of an IVLS compiler in Section 6.3. Its construction combines IVC and a Merkle tree via ideas that require some care but are primarily standard; for reference, we provide the construction in Section 6.11 and its security proof in Section 6.12. The user specifies the transition function for the ledger system by providing code for the native execution and also a (rank-1) constraint system for it. The latter representation, known as R1CS, is a standard representation for NP statements that can be viewed as a generalization of arithmetic circuits. We discuss the IVC scheme below. As for the Merkle tree, like prior works in the SNARK literature [480, 498], we base it on a Pedersen hash function over a suitable elliptic curve (a twisted Edwards curve), whose base field matches the field over which the IVC scheme's constraint system is defined, or on the Poseidon hash function [409]. This is for efficiency, as traditional hash functions such as SHA-256 are expensive in constraint systems.

PCD. We provide a generic trait for a PCD scheme, which enables the user to specify transitions whose incremental verification is desired, by giving an R1CS constraint system that checks their validity. We provide a generic implementation of this trait from any pairing-based SNARK for R1CS that comes equipped with a constraint system for its own verifier, by using the technique of MNT cycles in [297].[10] Our implementation works with both MNT cycles in `arkworks` (the

[10] Informally, there are two pairing-friendly curves with matching parameters, and two pairing-based SNARKs instantiated over these two curves. One SNARK verifies the other SNARK, and vice versa.

lower-security 298-bit cycle and the higher-security 753-bit cycle). In particular, we can base the PCD scheme on pairing-based SNARKs for R1CS with a *circuit-specific* setup, already part of arkworks (such as [487]), or with a *universal* setup that we contribute in this work ([470], as described below). Though not part of this work, we anticipate it to be rather straightforward to also implement the PCD scheme trait via hash-based (post-quantum) SNARKs for R1CS such as [473]. Lastly, to support the Pedersen-based Merkle tree, we contribute twisted Edwards curves suitable for the MNT cycles in arkworks that underlie PCD.

The IVC scheme that we use for IVLS is a special case of the PCD scheme above. We built PCD because PCD can used to distribute proving work via proof trees (e.g., see "parallel scan states" in [468]), and PCD is useful in security applications beyond IVLS [499–502].

Recursing a universal SNARK. We provide the first implementation of a constraint system for a pairing-based SNARK for R1CS with *universal* setup [470]. Recall that this means that the trusted generation of system parameters for the SNARK does not depend on the R1CS instance whose satisfiability is proved (but only on some upper bound to it). This enables realizing PCD/IVC based on the SNARK in [470] (and implemented in marlin [503]), so that the trusted generation of the system parameters for the PCD/IVC scheme does not depend on the user's choice of automaton. Our constraint system is a useful addition to existing constraint systems for other types of SNARKs, such as pairing-based SNARKs with circuit-specific setup (starting with [297]) and post-quantum SNARKs [473].

The universal SNARK in [470] is constructed via a general paradigm combining three ingredients: a *polynomial commitment scheme* (PC scheme), an *algebraic holographic proof* (AHP), and a cryptographic hash function for the Fiat–Shamir heuristic [218]. The SNARK verifier is assembled from these ingredients, and our SNARK verifier constraint system reflects this structure, which should facilitate the implementation of similar universal SNARKs in the future.

- *PC scheme.* We write a constraint system for the checks in the polynomial commitment scheme of Kate et al. [504], which ensures that a sender has correctly opened a committed polynomial at a desired point. Our constraint system supports degree enforcement and batching as described in [470] and is implemented in poly-commit [505]. Our constraint system supports degree enforcement and batching as described in [470]. This is the part of the verifier that checks a pairing-product equation on a pairing-friendly curve.

  Our implementation includes optimizations to reduce the number of constraints. For example, to batch multiple polynomial commitments and pairing checks into one, we replace the linear combination $r, r^2, r^3, \dots$ used in [470] with the linear combination $r_1, r_2, r_3, \dots$ as it is cheaper to derive multiple challenges from the Poseidon sponge (for the Fiat–Shamir transformation) instead of computing one challenge's powers via non-native arithmetic.
- *AHP.* We write a constraint system for the AHP verifier described in [470]. This involves checking polynomial equations, using values in the proof and values derived from the Poseidon sponge. These operations are over the field exposed by the PC scheme above. As the latter is different from the field of the constraint system (due to properties of pairing-friendly curves), we use constraints for non-native field arithmetic (see below).

  In [470], the verifier evaluates a few vanishing polynomials, which is expensive due to non-native field arithmetic. To reduce the cost, we modify Marlin to have the prover convince the verifier of the correct evaluation of the vanishing polynomials. This change also reduces the number of verifier constraints from $O(\log N)$ to $O(1)$, where N is the number of constraints of the circuit being verified in the constraint system.

- *Hashing.* For the cryptographic hash function we use an algebraic sponge (implemented in [473] based on the Poseidon hash function). We set the field of the algebraic sponge to equal the field of the constraint system. For our setting we wrote "adapters" to absorb into and squeeze out of the algebraic sponge different types of inputs that arise in our SNARK verifier (commitments from the PC scheme and non-native field elements).

Our modular design facilitates obtaining constraint systems for other SNARKs built via the same paradigm in [470]. For example, by modifying the equations checked by the AHP, it would be relatively straightforward to obtain a constraint system for the SNARK for arithmetic circuits in [471].

Non-native field arithmetic. The constraint system of the verifier requires checking arithmetic over a field $\mathbb{F}_q$ that is different from the field $\mathbb{F}_r$ of the constraint system. We provide a generic implementation in `arkworks` that hides the differences between native and non-native from the developers: one can program a constraint system with non-native field arithmetic as easily as if it were native. Our implementation is based on [489, 506, 507], but we optimize it for lower constraint weight (see below), which requires different parameter selection and multiplication.

Optimizing constraint weight. While the main efficiency metric of a constraint system is the number of constraints, a secondary efficiency metric is its *weight*, i.e., the number of non-zero entries in the coefficient matrices. While for SNARKs with circuit-specific setup (such as [487]) weight does not matter much, for all known SNARKs with universal setup (including [470]) efficiency also depends on weight. To address this additional consideration, we extended the constraint-writing framework `arkworks` [497] to enable weight-reducing optimizations. For example, we implement an automatic procedure that builds a dependency graph over all the linear combinations of constraint system variables and rewrites the constraint system to avoid re-using the same the linear combinations too many times (which greatly penalizes weight while saving few constraints). More generally, throughout all of our constraint writing, we balance the two (sometimes competing) goals of reducing number of constraints and reducing weight.

Privacy-preserving payments. Using our IVLS interface, implementing privacy-preserving payments is straightforward. We simply assembled the compliance predicate described in Section 6.5, and our library produced an incrementally verifiable version for this application. To optimize application memory for performance, we chose a Merkle tree layout that is suitable for coin commitments and serial numbers. We evaluate our implementation in Section 6.7.3.

6.7 Evaluation

We measure the size of our constraint system for the verifier of Marlin [470], the costs of IVC with universal setup vs. circuit-specific setup, and how incremental verification reduces participation costs. We establish that incremental verification based on preprocessing SNARKs with a

	Curve bit security	**Proving time (μs/constraint)**		**Verification time (ms)**		**Proof size (byte)**	
		Groth16	**Marlin**	**Groth16**	**Marlin**	**Groth16**	**Marlin**
BLS12-381	~ 128	46	485	4.69	8.31	192	1024
MNT4-298	~ 80	44	456	4.97	7.64	152	950
MNT6-298	~ 80	46	479	9.41	8.67	190	950
MNT4-753	~ 128	395	4426	51.25	72.18	380	2375
MNT6-753	~ 128	266	5540	92.98	81.26	475	2375

Table 6.2: Proving time per constraint, verification time, and proof size across preprocessing SNARKs and curves.

universal setup incurs modest overheads compared with the case of circuit-specific setup. Moreover, incremental verification significantly reduces participation costs compared with systems with no proofs or with transition proofs. While servers and clients greatly benefit from incremental verification, proof makers pay a high cost to produce proofs; we discuss these costs as well as techniques for reducing them in Section 6.7.5.

Our measurements are taken on a machine with an Intel Xeon 6136 CPU with a base frequency of 3.00 GHz and 252 GB of memory using a single thread. All reported proving times can be significantly reduced by using multiple threads (a capability that is already part of the codebase).

Below, we frequently refer to two state-of-the-art proof systems: (a) Groth16 [487], a preprocessing SNARK with a *circuit-specific setup*; and (b) Marlin [470], a preprocessing SNARK with a *universal setup* (the SNARK whose verifier we expressed as a constraint system).

6.7.1 Constraints for SNARK verification

We measure the size of our constraint system for the verifier of Marlin [470]. We discuss the MNT-298 cycle with the SNARK verifier built on the MNT6-298 curve, which verifies a proof over the MNT4-298 curve. (Costs when the curves are swapped are similar.)

The Marlin verifier that checks an R1CS instance with K-element public input has roughly $328825 + 4794K$ constraints, compared with $43186 + 7754K$ in the case of Groth16. This larger cost is to be expected because the (desirable) property of having a universal setup is harder to achieve and often leads to more expensive verifiers.

The size of our constraint system remains modest and is within a factor of ten of the size for a circuit-specific setup. Our constraint system for the Marlin verifier establishes the feasibility of recursive proofs via pairing-based SNARKs with universal setup. We show a breakdown of the constraint system of the Marlin verifier in Table 6.1. The polynomial commitment (PC) check, particularly group exponentiation, is responsible for much of the cost. In the future, this cost could be reduced via incomplete group arithmetic [508].

Marlin Verifier Component	**Constraints**	**Weight**
Prepare verification key	$61,506$	$294,603$
AHP	$62,807$	$339,043$
- Non-native arithmetic	$33,143$	$216,106$
PC check	$186,493$	$994,080$
- Group exponentiations	$161,862$	$821,769$
- Pairing	$9,376$	$65,804$
Fiat–Shamir	$29,664$	$122,937$
Other	$2,036$	$96,078$
Total	$\mathbf{332,828}$	$\mathbf{1,723,804}$

Table 6.1: Cost of the Marlin verifier and its main sub-components, including both the number and weight of constraints with $K = 10$ elements for public input.

6.7.2 Incremental verification

We compare the overhead of recursive proofs with that of transition proofs for the case of pairing-based SNARKs. The overhead originates from two sources: (1) the additional constraints used to verify the prior proof (beyond the constraints to prove the desired statement); and (2) the use of an MNT cycle to realize recursion, instead of using more efficient pairing-friendly curves, e.g., BLS12-381. Our measurements (Table 6.2) show that the recursion overhead is modest with Groth16 and Marlin.

(1) Additional constraints. Suppose that one wishes to recursively prove the correct execution of a transition function whose constraint system has M constraints. (I.e., the *compliance predicate* in IVC has size M.) Following the MNT-cycle paradigm of [297], we need to: (a) prove, over the MNT4 curve, the satisfiability of a constraint system of size $M + |V_{\text{MNT6}}|$, where V_{MNT6} denotes (a constraint system for) the MNT6 verifier; and (b) prove, over the MNT6 curve, the satisfiability of a constraint system of size $|V_{\text{MNT4}}|$, where V_{MNT4} denotes (a constraint system for) the MNT4 verifier. Proving times over the two curves are essentially the same, so each recursion amounts to proving $M+|V_{\text{MNT6}}|+|V_{\text{MNT4}}|$ constraints (rather than M without recursion). For MNT-298: (1) in Groth16, $|V_{\text{MNT4}}| + |V_{\text{MNT6}}|$ is 1.3×10^5 constraints; (2) in Marlin, $|V_{\text{MNT4}}| + |V_{\text{MNT6}}|$ is 5.9×10^5 constraints.

In other words, the number of constraints to prove for recursion is M plus a term that grows much slower than M—as M grows, the number of additional constraints is a smaller and smaller fraction of the number of proved constraints. For example, if M is two million constraints, recursion requires proving less than 2.6 million constraints.

(2) MNT cycles vs. BLS. We measured the main costs of a preprocessing SNARK (proving time, verification time, and proof size) on the BLS12-381 curve, the MNT-298 cycle, and the MNT-753 cycle, for both Groth16 and Marlin, shown in Table 6.2. MNT-298 has similar efficiency to

BLS12-381 but only 80-bit security; MNT-753 has greater security at an increased cost.

- *Proving time.* The proving times in both proof systems are quasilinear in the number of constraints. But, for a large range of parameters, proving times approximately grow linearly, as prior works show, and so we report proving time *per constraint*. Compared with Groth16, Marlin is $10\times$ slower. Compared with BLS12-381, MNT-298 has a similar proving time, but MNT-753 is $10\times$ slower.
- *Verification time.* Pairing-based preprocessing SNARKs typically have short verification times. We measured the verification times for a constraint system with $N = 2^{16}$ constraints and $K = 10$ public inputs. All measurements are less than 100 ms. Compared with Groth16, Marlin is $2\times$ slower. Compared with BLS12-381, MNT-298 is $2\times$ slower or less, and MNT-753 is $10\times$ to $20\times$ slower.
- *Proof size.* All proof sizes are less than three kilobytes. Compared with Groth16's, Marlin's proof is about $5\times$ larger. Proof sizes over BLS12-381 and MNT-298 are similar, while proof sizes over MNT-753 are $2.5\times$ larger.

6.7.3 Privacy-preserving payments

We measure the cost of incremental verification for privacy-preserving payments as described in Section 6.5 in Table 6.3. The compliance predicate requires checking four Merkle proofs (two for the set of coin commitments and two for the set of serial numbers) and verifying a zero-knowledge proof attesting the validity of the transaction. (Our compliance predicate does not incorporate a consensus protocol, which would be necessary in practice.) Therefore, the cost of incremental verification largely depends on the choice of hash function and the choice of proof system (Table 6.3). As expected, a universal SNARK incurs some overhead compared with a circuit-specific one. The choice of hash function also makes a difference: the constraint-optimized Poseidon hash function, compared with Pedersen, is $5\times$ faster in Groth16, and $3.5\times$ faster in Marlin.

Hash function	**Proof system**	
	Groth16	**Marlin**
Pedersen	$1,230,478$	$1,377,884$
Poseidon	$218,088$	$365,494$

Table 6.3: Number of constraints for incrementally verifiable privacy-preserving payments using different hash functions and different proof systems on the MNT-298 cycle.

	Network		**Sync time**	
	Server	Client	Server	Client
No proofs	172.3 GB	172.3 GB	16.9 hrs	16.9 hrs
Transition proofs				
Groth16 (BLS12-381)	2.3 GB	53.4 MB	1.6 hrs	19 min
Marlin (BLS12-381)	2.4 GB	217.4 MB	1.9 hrs	35 min
Recursive proofs				
Groth16 (MNT-298)	2.2 GB	0.2 KB	1.3 hrs	4.8 ms
Marlin (MNT-298)	2.2 GB	1.6 KB	1.3 hrs	16.3 ms
Groth16 (MNT-753)	2.2 GB	0.4 KB	1.3 hrs	50.8 ms
Marlin (MNT-753)	2.2 GB	3.9 KB	1.3 hrs	159.0 ms

Table 6.4: Estimated synchronization costs. Sync time does not include the time to download data. We use the time to verify two ECDSA signatures over the secp256k1 curve as a lower bound for the time to verify a transaction. We assume that transition proofs and recursive proofs use a Pedersen hash function to summarize the current state, and transition proofs are generated for each block.

6.7.4 Synchronization costs

We now consider the synchronization costs, which dominate the participation costs for servers and clients. To demonstrate the benefits of IVLS (recursive proofs), we consider a simplified version of Bitcoin with no scripts. We base the parameters in Table 6.6 on statistics from Bitcoin over the last ten years.

In Table 6.4, we show how no proofs, transition proofs, and recursive proofs affect synchronization costs. As expected, the system with no proofs imposes the highest synchronization costs, as it requires both servers and clients to download and re-execute every transaction. Transition proofs reduce the overhead of both clients and servers by orders of magnitude, and recursive proofs decrease the overhead for clients by orders of magnitude again, reducing the sync time to milliseconds and network cost to kilobytes for the clients. The server sync time and network cost are not as affected by the switch from transition proofs to recursive proofs because for both the network cost is dominated by the state size and the sync time is dominated by the time to hash the state. We show recursive proofs for both the MNT-298 curves ($\sim$80-bit security) and the MNT-753 curves ($\sim$128-bit security), as the BLS12-381 curve offers $\sim$128-bit security.[12]

[11] Transition proofs could be generated less frequently to reduce network cost and verifier computation; this would result in longer periods where proofs have not been generated for the system state.

[12] While these curves nominally target the 80-bit or 128-bit security levels, all pairing-based SNARKs lose a few bits of security relative to the underlying curve due to the fact that public parameters contains certain powers of generators

Total blocks:	25K	50K	100K	250K
Bitcoin SPV	1.9 MB	3.8 MB	7.6 MB	19 MB
FlyClient	109 KB	135 KB	163 KB	204 KB
Plumo				
Groth16	6.1 KB	7.4 KB	10 KB	18 KB
Marlin	9.7 KB	15 KB	25 KB	54 KB
Recursive proofs				
Groth16 (MNT-298)	0.2 KB			
Marlin (MNT-298)	1.6 KB			
Groth16 (MNT-753)	0.4 KB			
Marlin (MNT-753)	3.9 KB			

Table 6.5: Network cost for different light client schemes, for a ledger system modeled after a 10-year "bare-bones" Bitcoin (no scripts). Plumo uses the CP6 curve from Zexe [480].[11]

Transaction size:	370 B	Transactions per block:	$2,000$
Total transactions:	500M	Transaction verification time:	122 μs
State size:	2.2 GB	Hash time for 1 MB:	2.059 s

Table 6.6: Setup for the 10-year "bare-bones" Bitcoin.

In terms of network cost, the most expensive recursive proofs (Marlin over MNT-753) are still 46M× lighter than no proofs and 57K× lighter than transition proofs. In terms of synchronization time, the foregoing recursive proofs are 382K× faster than no proofs and 3K× faster than transition proofs.

Comparison with other light client systems. Recursive proofs (from IVLS) make it possible to construct a client that is *extremely light*, both in network cost and in synchronization time. In Table 6.6, we show the differences in network cost and synchronization time between recursive proofs and the following: (1) Bitcoin's simplified payment verification (SPV), (2) FlyClient's proof of consensus [464], and (3) Plumo's proof of consensus [511].

Compared with these prior schemes for light clients, recursive proofs (via IVLS) reduce client network cost and synchronization time, at the cost of relying on sufficiently powerful proof makers, as we discuss in Section 6.7.5. Moreover, recursive proofs provide a stronger security guarantee by verifying the correctness of the application transitions as well as consensus transitions (as described in Section 6.10.1).

[509]. Similar security losses also happen for SNARKs based on generic groups due to the security reduction [510], and are typically ignored in practice.

Bitcoin's simple payment verification (SPV), estimated with the parameters in Table 6.5, would require downloading all the block headers of the blockchain—about 20 MB—which is at least $5,000\times$ larger than recursive proofs. Synchronization time is small since it mainly involves checking the proof of work for every block. However, SPV provides only limited security and functionality: (1) Though SPV enables the client to check the consensus (proof of work), the client does not check the correctness of the transitions. (2) With SPV, the client can check if a transaction is present, but the client cannot check if the transaction is unspent; light clients of the same account need to synchronize with one another.

FlyClient enables a client to efficiently check the consensus in proof-of-work protocols; rather than downloading every block header as in SPV, FlyClient downloads a *logarithmic* number of headers. Estimated with a flat proof-of-work difficulty and parameters in Table 6.6, a client needs to download proofs of 204 KB, using the simulator in [512]. (We report a smaller proof size than the FlyClient paper because we are measuring over our bare-bones Bitcoin rather than Ethereum, which has larger block headers.) While both network cost and synchronization time are much better than SPV, the network cost is still $50\times$ larger than that of recursive proofs. Moreover, FlyClient does not check the correctness of the transitions and cannot check if a transaction is unspent (as in SPV).

Plumo provides efficient checking for changes in the consensus committee. If a proof is generated every six months (as suggested in [511]), the client needs to download at least 18 KB with Groth16 or 54 KB with Marlin. In Table 6.5, we show that the network cost grows linearly with the number of blocks, assuming a constant block rate. Synchronization time remains small due to efficient verification. Similar to SPV and FlyClient, Plumo does not check the transitions. Moreover, Plumo does not fully verify the consensus, such as whether committee members are indeed winners of the proof of stake. Plumo is also inefficient when a proof for recent transactions is unavailable (a common case, as proofs are generated infrequently); in this case, the client needs to download and verify these transactions. Making proof generation more frequent, or for a larger period of time, would increase the cost for either proving or verification.

6.7.5 Limitation: producing proofs is costly

Recursive proofs (or even transition proofs) are generally advantageous for servers and clients compared with no proofs. The main limitation of this approach, which needs to be balanced against the advantages, is the cost incurred by proof makers to produce proofs. While asymptotically producing proofs is not much more expensive than executing transactions, concrete costs make proving *orders of magnitude* slower than execution. For example, generating a proof for the correct execution of 20 two-input two-output transactions in our privacy-preserving payments system takes ~ 33 s using 8 threads with Groth16 over the MNT-298 cycle with the Poseidon hash. This latency limits a system's throughput.

That said, proving times for proof makers can be significantly reduced via existing techniques such as proof trees (or "parallel scan state" [468]), parallelism across many machines [513], or specialized proving hardware [514]. These techniques can increase the system's transaction throughput: Using proof trees, 100 machines (with the same setup above) can together produce proofs for one million privacy-preserving transactions in nine hours. (The Mina cryptocurrency [469] uses proof trees for this reason.) We leave a detailed study to future work (and real-world deployments), and here only mention that proof trees should be straightforward to build given our PCD module. We conclude by noting that recursive proofs are not much harder to produce than transition proofs, as proving the state transition, not recursive verification, dominates the costs. This suggests that moving from transition proofs to recursive proofs is generally a good choice.

6.8 Other related work

Less related to our goals, several works have studied how to reduce participation costs via methods other than incremental verification, for either servers or clients.

Servers. Transition proofs, which avoid having every server re-execute every transaction, have been studied by researchers, open-source developers, and industry. Transition proofs for *batches* of transactions have been studied as a "layer-2 scaling solution" on Ethereum for concrete applications like payments or self-custodial trading, such as StarkDEX [515], StarkPay [516], and "rollups" [517, 518]. Ozdemir et al. [489] study more efficient transition proofs for large batches of transactions using RSA accumulators for local updates to the application's state. Lee et al. [490] study liveness for applications that use transition proofs and show efficiency gains for modest-size batches of ERC-20 transactions compared with naive re-execution. Gabizon et al. [511] propose Plumo, which uses transition proofs to prove correct evolution of consensus over large periods of time (several months), leading to fewer proofs required for client synchronization; focusing on the consensus rather than application updates allows proving many transitions at once. All of these approaches based on transition proofs incur participation costs that grow linearly in the number of state transitions; incremental verification (our focus) avoids such costs. Leung et al. [519] design bootstrapping techniques for the Algorand proof-of-stake protocol [520] that provide guarantees about the validity of past transactions, without relying on a long-standing committee to store and certify state in order to protect against adaptive corruption. They do this by minimizing the amount of state that needs to be tracked, sharding this state across servers, and generating checkpoints to prevent new servers joining the system from having to check all transitions from the initial state.

Clients. A rich line of work has explored techniques for reducing client participation costs, starting with Nakamoto's Simplified Payment Verification (SPV) [463]. A recurring theme is to use lightweight approaches while settling for weaker, yet meaningful, security. For example, SPV enables a server to convince a client that a transaction belongs to some past block, though the client cannot check if the current state is the result of applying transactions correctly. Analogous protocols, known as "light clients", have been developed for other cryptocurrencies, including ones with privacy guarantees [492, 521, 522]. Some light clients hide the queries themselves using trusted hardware [492–494] or private information retrieval [495].

Some works have further improved the efficiency of light clients. Bünz et al. [464] propose FlyClient, which builds on previous work on "non-interactive proofs of proof-of-work" (NIPoPoWs) [523, 524], allowing light clients to validate the cumulative work put into a chain by looking at only a logarithmic number of block headers.

Combining transition proofs with a light client protocol can reduce synchronization costs at the server (via transition proofs) and client (via the light client protocol). Although such a solution may have comparable or even better concrete efficiency compared to IVLS, it does not provide the same security guarantees: the client does not verify transition proofs and so must trust that the server state was reached by applying transactions correctly.

6.9 Formalizing applications

We describe how to express several applications in the formalism of ledger systems (Section 6.3), so that, by using our transformation, one can obtain incrementally verifiable versions of these applications. In Section 6.9.1 we discuss account-based payments, in Section 6.9.2 UTXO-based payments, in Section 6.9.3 privacy-preserving payments, in Section 6.9.4 privacy-preserving computation, and in Section 6.9.5 key transparency. For each application, we discuss the main efficiency features of the incrementally verifiable version of the application.

In this section we let $\mathsf{SIG} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme. Public parameters required by SIG (e.g., the description of a cyclic group) can be viewed as hardcoded in a ledger system's programs.

6.9.1 Account-based payments

We describe a ledger system that captures the functionality of a simple account-based currency.

• **State.** The state S of the ledger system is a search tree that contains all accounts. Each account is a tuple $(\mathsf{pk}, \mathsf{bal}, \mathsf{ctr})$ where pk is a signature public key of SIG, bal is its corresponding balance, and ctr is a counter of the number of transactions that this user initiates.

• **Transition function.** The transition function F processes three types of transactions tx:

– A *create transaction* of the form $(\texttt{create}, \mathsf{pk})$. If the state S already contains a tuple of the form $(\mathsf{pk}, \mathsf{bal}, \mathsf{ctr})$, the transition function F outputs the result $y := \texttt{error}$ and empty state update $\Delta_S := \bot$. Otherwise, F outputs the result $y := \texttt{ok}$ and the state update Δ_S that inserts the tuple $(\mathsf{pk}, 0, 0)$ into S.

– A *deposit transaction* of the form $(\texttt{deposit}, \mathsf{pk}, \mathsf{amt})$. If the state S does not contain a tuple of the form $(\mathsf{pk}, \mathsf{bal}, \mathsf{ctr})$, the transition function F outputs the result $y := \texttt{error}$ and empty state update $\Delta_S := \bot$. Otherwise, F outputs the result $y := \texttt{ok}$ and the state update Δ_S that replaces $(\mathsf{pk}, \mathsf{bal}, \mathsf{ctr})$ with $(\mathsf{pk}, \mathsf{bal} + \mathsf{amt}, \mathsf{ctr})$.

– A *transfer transaction* of the form $(\texttt{transfer}, \mathsf{pk}_{\mathsf{from}}, \mathsf{pk}_{\mathsf{to}}, \mathsf{amt}, \mathsf{sig})$. The transition function F checks that the state S contains tuples of the form $(\mathsf{pk}_{\mathsf{from}}, \mathsf{bal}_{\mathsf{from}}, \mathsf{ctr}_{\mathsf{from}})$ and $(\mathsf{pk}_{\mathsf{to}}, \mathsf{bal}_{\mathsf{to}}, \mathsf{ctr}_{\mathsf{to}})$, that $\mathsf{amt} \leq \mathsf{bal}_{\mathsf{from}}$, and that $\mathsf{SIG}.\mathsf{Verify}(\mathsf{pk}_{\mathsf{from}}, (\mathsf{pk}_{\mathsf{from}}, \mathsf{pk}_{\mathsf{to}}, \mathsf{amt}, \mathsf{ctr}_{\mathsf{from}}), \mathsf{sig}) = 1$. If any of these checks fails, F outputs the result $y := \texttt{error}$ and empty state update $\Delta_S := \bot$. Otherwise, F outputs the result $y := \texttt{ok}$ and the state update Δ_S that replaces $(\mathsf{pk}_{\mathsf{from}}, \mathsf{bal}_{\mathsf{from}}, \mathsf{ctr}_{\mathsf{from}})$, $(\mathsf{pk}_{\mathsf{to}},$

$\mathsf{bal}_{\mathsf{to}}, \mathsf{ctr}_{\mathsf{to}})$ with $(\mathsf{pk}_{\mathsf{from}}, \mathsf{bal}_{\mathsf{from}} - \mathsf{amt}, \mathsf{ctr}_{\mathsf{from}} + 1), (\mathsf{pk}_{\mathsf{to}}, \mathsf{bal}_{\mathsf{to}} + \mathsf{amt}, \mathsf{ctr}_{\mathsf{to}})$. The counter is used to prevent replaying a previous transaction.

• **Client function.** The client function C processes two types of queries:

– A *balance query* of the form $(\mathtt{balance}, \mathsf{pk})$ that returns the output $y := \mathsf{bal}$ if the state S contains a tuple of the form $(\mathsf{pk}, \mathsf{bal}, \mathsf{ctr})$. Otherwise, C returns the output $y := \mathtt{error}$.

– A *counter query* of the form $(\mathtt{ctr}, \mathsf{pk})$ that returns the output $y := \mathsf{ctr}$ if the state S contains a tuple of the form $(\mathsf{pk}, \mathsf{bal}, \mathsf{ctr})$. Otherwise, C returns the output $y := \mathtt{error}$.

6.9.2 UTXO-based payments

We describe a ledger system that captures the functionality of a simple UTXO-based currency, modeling a bare-bones version of Bitcoin.

• **State.** The state S is a search tree containing key-value pairs where each key is a signature public key pk of SIG and the corresponding value is a search tree of pairs $(\mathsf{cid}, \mathsf{amt})$ where cid is a coin identifier and amt is the corresponding amount of this coin; these pairs represent the coins owned by the public key pk.

• **Transition function.** The transition function F processes transactions of the form:

$$\mathsf{tx} = \left([\mathsf{cid}_i^{\mathsf{in}}]_1^m, [\mathsf{cid}_j^{\mathsf{out}}]_1^n, [\mathsf{amt}_j^{\mathsf{out}}]_1^n, [\mathsf{pk}_j^{\mathsf{out}}]_1^n, \mathsf{pk}^{\mathsf{in}}, \mathsf{sig}\right)$$

where $[\mathsf{cid}_i^{\mathsf{in}}]_1^m$ are coin identifiers of the inputs to tx and $[\mathsf{cid}_j^{\mathsf{out}}]_1^n$ are coin identifiers of the outputs of tx. For each $j \in [n]$, the value $\mathsf{amt}_j^{\mathsf{out}}$ is the amount being sent to $\mathsf{pk}_j^{\mathsf{out}}$. The value sig is a signature over the transaction with respect to the public key $\mathsf{pk}^{\mathsf{in}}$.

The transition function F looks up $\mathsf{pk}^{\mathsf{in}}$ in S in order to find tuples of the form $\{(\mathsf{cid}_i^{\mathsf{in}}, \mathsf{amt}_i^{\mathsf{in}})\}_{i \in [m]}$ belonging to $\mathsf{pk}^{\mathsf{in}}$. Then it checks that $\sum_{i=1}^m \mathsf{amt}_i^{\mathsf{in}} = \sum_{j=1}^n \mathsf{amt}_j^{\mathsf{out}}$ and that S does not contain any tuple of the form $(\mathsf{cid}_j^{\mathsf{out}}, \cdot)$ for $\mathsf{pk}_j^{\mathsf{out}}$ for any $j \in [n]$. Moreover, F verifies that the signature for this transaction is valid by checking

$$\mathsf{SIG.Verify}\left(\mathsf{pk}^{\mathsf{in}}, ([\mathsf{cid}_i^{\mathsf{in}}]_1^m, [\mathsf{cid}_j^{\mathsf{out}}]_1^n, [\mathsf{amt}_j^{\mathsf{out}}]_1^n, [\mathsf{pk}_j^{\mathsf{out}}]_1^n), \mathsf{sig}\right) \overset{?}{=} 1 \ .$$

If any lookup or check fails, F outputs the result $y := \mathtt{error}$ and empty state update $\Delta_S := \bot$. Otherwise, F outputs the result $y := \mathtt{ok}$ and the state update Δ_S that: (a) for every $i \in [m]$ removes $(\mathsf{cid}_i^{\mathsf{in}}, \mathsf{amt}_i^{\mathsf{in}})$ from the search tree of $\mathsf{pk}^{\mathsf{in}}$; and (b) for every $j \in [n]$ adds $(\mathsf{cid}_j^{\mathsf{out}}, \mathsf{amt}_j^{\mathsf{out}})$ to the search tree of $\mathsf{pk}_j^{\mathsf{out}}$.

• **Client function.** The client function C processes two types of queries:

– A *balance query* of the form $(\mathtt{balance}, \mathsf{pk})$ that returns the total assets held by the public key pk. Namely, C outputs $y := \sum_{i=1}^k \mathsf{amt}_i$ for the k pairs $(\cdot, \mathsf{amt}_i)$ associated with pk in S.

– A *coin query* of the form $(\mathtt{coin}, \mathsf{pk})$ that returns all the coin identifiers and amounts owned by the public key pk. Namely, C outputs y containing all pairs of the form $(\mathsf{cid}, \mathsf{amt})$ for pk in S.

6.9.3 Privacy-preserving payments

We describe a ledger system that captures the functionality of a simple privacy-preserving currency, modeling the basic functionality of Zerocash [479]. We assume basic familiarity with Zerocash and only discuss the aspects relevant for the formalism of ledger systems. We denote by $\mathsf{VerifyZKP}$ the algorithm that validates the zero-knowledge proof contained in a Zerocash transaction (after suitably parsing the transaction).

• **State.** The state S contains (a) a Merkle tree of coin commitments, (b) a Merkle tree of serial numbers, and (c) a list of encrypted coins of all the transactions so far.

• **Transition function.** The transition function F processes two types of transactions tx:

– A *mint transaction* of the form $\mathsf{tx} = (\mathtt{mint}, \mathsf{cm}, v, \pi_{\text{MINT}})$, which creates a coin of value v with commitment cm. If $\mathsf{VerifyZKP}(\mathsf{tx}) = 1$, F outputs the result $y := \mathtt{ok}$ and the state update Δ_S that adds cm to the Merkle tree of commitments. Otherwise, F outputs the result $y := \mathtt{error}$ and empty state update $\Delta_S := \perp$.

– A *pour transaction* of the form $\mathsf{tx} = (\mathtt{pour}, \mathsf{sn}_1^{\mathsf{in}}, \mathsf{sn}_2^{\mathsf{in}}, \mathsf{cm}_1^{\mathsf{out}}, \mathsf{cm}_2^{\mathsf{out}}, \mathbf{e}_1, \mathbf{e}_2, \pi_{\text{POUR}})$. The values $\mathsf{sn}_1^{\mathsf{in}}, \mathsf{sn}_2^{\mathsf{in}}$ are the serial numbers of the old (spent) coins, while the values $\mathsf{cm}_1^{\mathsf{out}}, \mathsf{cm}_2^{\mathsf{out}}$ are the commitments of the new (created) coins. The values $\mathbf{e}_1, \mathbf{e}_2$ are encryptions of the new coins under public keys of the (unknown) receivers. The value π_{POUR} is a proof used to attest to the validity of this transaction.

  The transition function F checks that $\mathsf{VerifyZKP}(\mathsf{tx}) = 1$ and that the serial numbers $\mathsf{sn}_1^{\mathsf{in}}$ and $\mathsf{sn}_2^{\mathsf{in}}$ do not appear in the list of serial numbers. If both checks pass, F outputs the result $y := \mathtt{ok}$ and the state update Δ_S that adds $\mathsf{cm}_1^{\mathsf{out}}$ and $\mathsf{cm}_2^{\mathsf{out}}$ to the Merkle tree of commitments, adds $\mathsf{sn}_1^{\mathsf{in}}$ and $\mathsf{sn}_2^{\mathsf{in}}$ to the Merkle tree of serial numbers, and appends encrypted coins $\mathbf{e}_1$ and $\mathbf{e}_2$ to the list of encrypted coins. Otherwise, F outputs the result $y := \mathtt{error}$ and empty state update $\Delta_S := \perp$.

• **Client function.** The client function C enables a user's client to find payments sent to the user. The client function C takes as input a viewing key $\mathsf{sk}_{\mathsf{view}}$ and checks which of the encrypted coins in the state S belong to this user. This check can be done by scanning through all encrypted coins and trying to decrypt each using $\mathsf{sk}_{\mathsf{view}}$. We can, by careful designs, further constrain $\mathsf{sk}_{\mathsf{view}}$ to only capable of testing whether the encrypted coins belong to the user but incapable of seeing the underlying information.

6.9.4 Privacy-preserving computation

We can also express privacy-preserving computation in a ledger system, which models the basic functionality in Zexe [480]. This is similar to the case of privacy-preserving payment, so we will focus on the differences as follows. We assume basic familiarity with Zexe and only discuss the aspects relevant for the formalism of ledger system. In Zexe, computation is expressed in terms of operations over *records*. A computation step consumes some previous records (takes old data as input) and creates some new records (outputs new data).

• **State.** The state S comprises a Merkle tree of all record commitments appearing in all transactions and a Merkle tree of serial numbers so far.

• **Transition function.** The transition function processes a transaction tx of the form $([\mathsf{sn}_i^{\mathsf{in}}]_1^N,$ $[\mathsf{cm}_i^{\mathsf{out}}]_1^M, \pi)$, which consumes records associated with serial numbers in $[\mathsf{sn}_i^{\mathsf{in}}]_1^N$ and generates records associated with commitments in $[\mathsf{cm}_i^{\mathsf{out}}]_1^M$. The value π is a cryptographic proof that the computation is executed correctly. The transition function F checks that $\mathsf{VerifyZKP}(\mathsf{tx}) = 1$ and that all the serial numbers $[\mathsf{sn}_i^{\mathsf{in}}]_1^N$ do not appear in the list of serial numbers in S. If the checks pass, F outputs the result $y := \texttt{ok}$ and the state update Δ that adds $[\mathsf{cm}_i^{\mathsf{out}}]_1^M$ to the Merkle tree of commitments and adds $[\mathsf{sn}_i^{\mathsf{in}}]_1^N$ to the search tree of serial numbers. Otherwise, F outputs $y :=$ $\texttt{error}$ and empty state update $\Delta := \perp$.

• **Client function.** The client function C checks if a record has been committed and if a record has been consumed in the state S. The client function C processes two types of queries:

– A *commitment query* of the form $(\texttt{commitment}, \mathsf{cm})$ that returns 1 if cm is in the Merkle tree of commitments in S and 0 otherwise.

– A *consumption query* of the form $(\texttt{consumption}, \mathsf{sn})$ that returns 1 if sn is in the search tree of revealed serial numbers in S and 0 otherwise.

6.9.5 Key transparency

An application of ledger systems outside of cryptocurrencies is key transparency [481, 496], a public directory that maps usernames to public keys. Users can publish their own public keys on this directory, query other users' public keys, and monitor that the directory behaves correctly. We consider two variants of key transparency, which differ in how they handle the case when a user wants to change their public key in the directory (e.g., due to secret key loss or due to routine key update).

Variant 1: key update with authorization. A user, to update their own public key, must provide a signature under their previous public key or under a trusted authority's public key (to account for secret key loss).

• **State.** The state S consists of a Merkle tree that maps usernames to public keys. The state S also stores the trusted authority's public key, denoted by $\mathsf{pk}_{♔}$.

• **Transition function.** The transition function F processes two types of transactions tx:

– A *registration transaction* of the form $(\texttt{new}, u, \mathsf{pk}_u)$. If the tree in S already has a node for the username u, F outputs the result $y := \texttt{error}$ and empty state update $\Delta_S := \perp$. Otherwise, F outputs the result $y := \texttt{ok}$ and the state update Δ_S that adds (u, pk_u) into the tree.

– An *update transaction* of the form $(\mathsf{update}, u, \mathsf{pk}_u^{\mathsf{new}}, \mathsf{sig})$. If the tree in S does not contain a node with username u, F outputs the result $y := \texttt{error}$ and empty state update $\Delta_S := \perp$. Otherwise, letting $(u, \mathsf{pk}_u^{\mathsf{old}})$ be the node for for u, F checks that $\mathsf{Verify}(\mathsf{pk}_u^{\mathsf{old}}, (u, \mathsf{pk}_u^{\mathsf{new}}), \mathsf{sig}) = 1$ or $\mathsf{Verify}(\mathsf{pk}_{♔}, (u, \mathsf{pk}_u^{\mathsf{old}}, \mathsf{pk}_u^{\mathsf{new}}), \mathsf{sig}) = 1$. If so, F outputs the result $y := \texttt{ok}$ and the state update

Δ_S that changes the node to $(u, \mathsf{pk}_u^{\mathsf{new}})$; else if the signature is invalid, F outputs $y := \texttt{error}$ and empty state update $\Delta_S = \perp$.

6.10 Further considerations

We discuss further considerations for applications that are beyond the scope of this paper, but may be useful for real-world deployments.

6.10.1 Integration with consensus protocols

While in describing applications (in Section 6.5) we do not mention a consensus protocol, in the real world one may need to take consensus into account because it *must* be incrementally verifiable if the system as a whole is to be incrementally verifiable. We view a consensus protocol as an algorithm that can be folded into the application, by suitably augmenting the application state and transition function, so that its incremental evolution will then be proved together with the application. Consensus protocols need not be compatible with incremental verification "off-the-shelf", and prior work has studied how to make specific consensus protocols incrementally verifiable [466, 468]; a systematic treatment of this remains an exciting research direction.

Here, we give a brief overview of how this can be achieved. Consider peer-to-peer systems with multiple servers running an application. The servers must somehow agree on the next transaction (or block of transactions) to process. This is typically achieved by way of a consensus protocol, which leverages paradigms such as proof of work (PoW), proof of stake (PoS), or others.

We summarize how to integrate consensus protocols with IVLS. To make ledger systems incrementally verifiable, one must augment the application's transition function F to add an algorithm that maintains consensus information. Otherwise, even if the application is incrementally verifiable, the consensus would not be, and the overall system would not be incrementally verifiable. (Participation costs would remain high due to the need to "catch up" on consensus information from the initial state of the system.) However, not all consensus protocols are amenable to incremental verification: some protocols require servers to store large amounts of consensus information (e.g., going back arbitrarily far in history).

Different consensus paradigms have been studied (see recent surveys [525, 526]), and their suitability for incremental verification varies. Prior work has focused on adapting specific consensus protocols for incremental verification [466, 468], so we explain how they fit within IVLS.

Nakamoto consensus. The canonical consensus protocol based on proof of work is the Nakamoto consensus protocol [463], which instructs servers to follow the "longest chain rule", meaning that the correct chain is the one with the highest amount of cumulative work. In more detail, each transaction contains a proof of work with a specific difficulty, and the cumulative work of a chain is the sum of the difficulties across all transactions in the chain. To maintain this information, the IVLS application state is augmented with a field that stores the cumulative work; when processing a transaction, the transition function adds the difficulty of the transaction's proof of work to the cumulative work stored in the application state. Therefore, the cumulative work is incrementally updated, and anyone can correctly choose between different application states by selecting the one with the highest cumulative work.

Ouroboros. As noted in [468], many proof-of-stake protocols are not incrementally updatable "off the shelf" because their chain selection rules require reasoning about information arbitrarily in the past to protect against long-range fork attacks. Bonneau et al. [468] construct a proof-of-stake protocol called Ouroboros Samasika (based on Ouroboros Genesis [488]) whose chain selection rule is *incrementally updatable*. Because chain selection is incrementally updatable, the IVLS transition function can be augmented to support it efficiently.

Consensus with incremental updates. Given the value of incremental verifiability, we believe that it would be valuable to see more research into consensus protocols that are suitable for incremental verification. An informative first step would be to conduct a systematic study of which known protocols based on proof of work (e.g., Bitcoin-NG [527], GHOST [528], Spectre [529], and others) can be implemented via incremental updates to a small state; a similar study for proof-of-stake protocols would be valuable. Moreover, future research in consensus protocols may want to consider incremental updates to be one of the "standard" desiderata.

6.10.2 Privacy considerations

Some ledger systems provide privacy guarantees (e.g., [479, 480] as discussed in Section 6.5). Applying an IVLS compiler to such ledger systems does *not* affect those privacy guarantees.

However, the queries that a client makes to the (public) application state may involve private information, and so a client may wish to have some privacy guarantees against servers that answer its queries. We consider this problem beyond the scope of this paper (our focus is on reducing participation costs). In particular, the notion of an IVLS does not make any attempts to ensure that clients in $(\mathsf{vF}, \mathsf{vC})$ have any more privacy guarantees against servers than in the original ledger system $\mathsf{LS} = (F, C)$.

It remains an interesting direction to investigate how to achieve private queries in practice, possibly while also reducing participation costs (as achieved by incremental verifiability). Here we only briefly recall two generic techniques that in principle could be used to protect the client's query privacy against servers.

Secure multi-party computation. One approach is for the client and server to engage in a two-party protocol to securely compute $y \leftarrow C^S(x)$. The client would provide the query x as input to the protocol, and receive the answer y as output; the server would provide the application state S as input to the protocol, and receive no output. The two-party protocol can leverage private information retrieval (PIR) [530] to hide client function C's access to the state. Alternatively, for efficiency reasons, the client could instead choose to engage in a *multi*-party protocol [81, 82, 207, 208], enlisting the help of multiple servers that are assumed to not collude (up to some threshold). In either case, privacy would require hiding access patterns to the state, which may be expensive. (Alternatively one could aim to trade more efficiency for less privacy by deliberately leaking specific information about x.)

Secure hardware. Another approach is for the client to communicate with a piece of secure hardware (e.g., using hardware enclaves [531, 532]) on the server's machine, which will execute the client's query within certain secure boundaries. The privacy of the client would then depend upon the threat model and security guarantees of the secure hardware used.

6.11 Construction of an IVLS compiler

We construct an incrementally verifiable ledger compiler $(\mathsf{Setup}, \mathsf{MakeSF}, \mathsf{MakeC}, \mathsf{History})$. In Section 6.11.1 we describe the building blocks that we use. In Section 6.11.2 we describe the auxiliary state. In Sections 6.11.3 to 6.11.6 we describe each of the components. We discuss the security proof in Section 6.12, where we also specify in more detail the security properties that we assume hold for the building blocks.

6.11.1 Building blocks

Non-interactive argument of knowledge. A non-interactive argument of knowledge (ARK) is a tuple of polynomial-time algorithms $\mathsf{ARK} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$ with the following syntax:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}_{\mathsf{ARK}}$: on input a security parameter 1^λ, Setup samples ARK public parameters $\mathsf{pp}_{\mathsf{ARK}}$.
- $\mathsf{KeyGen}(\mathsf{pp}_{\mathsf{ARK}}, R) \to (\mathsf{pk}_R, \mathsf{vk}_R)$: on input public parameters $\mathsf{pp}_{\mathsf{ARK}}$ and a specification of an NP relation R, KeyGen deterministically derives a proving key pk_R and a verification key vk_R for the NP relation R.
- $\mathsf{Prove}(\mathsf{pk}_R, \mathbb{x}, \mathbb{w}) \to \pi$: on input a proving key pk_R, instance $\mathbb{x}$, and witness $\mathbb{w}$, Prove outputs a proof π.
- $\mathsf{Verify}(\mathsf{vk}_R, \mathbb{x}, \pi) \to b$: on input a verification key vk_R, instance $\mathbb{x}$, and proof π, Verify determines if π is a convincing proof (of knowledge) for the statement "there exists $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in R$".

The completeness and knowledge soundness properties that we will assume for ARK are stated in Section 6.12.3.

Incrementally verifiable computation. Incrementally verifiable computation (IVC) [465] is a cryptographic primitive that augments multi-step automata computations with proofs attesting to the correctness of the computation so far. There is an efficient procedure to move from a state of the automaton and its proof to the new state of the automaton and a corresponding new proof. The formalism for IVC that we use is drawn from the literature on proof-carrying data [297, 477, 478], which extends IVC to more general settings of incrementally verifiable distributed computations. In particular, the automata computations that we consider are non-deterministic: each transition takes an old state *and an auxiliary input* to a new state.

An IVC scheme is a tuple of polynomial-time algorithms $\mathsf{IVC} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$ with the following syntax.

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}_{\mathsf{IVC}}$: on input a security parameter 1^λ, Setup samples IVC public parameters $\mathsf{pp}_{\mathsf{IVC}}$.
- $\mathsf{KeyGen}(\mathsf{pp}_{\mathsf{IVC}}, \Pi) \to (\mathsf{pk}_\Pi, \mathsf{vk}_\Pi)$: on input IVC public parameters $\mathsf{pp}_{\mathsf{IVC}}$ and a specification of an IVC compliance predicate Π, KeyGen outputs a proving key pk_Π and a verification key vk_Π for the predicate Π. A compliance predicate Π determines which transitions are valid: it receives input a new message z^{new} (a new "state"), local data w (an "auxiliary input"), and old message z^{old} (an old "state"), and outputs a bit indicating whether the transition from z^{old} to z^{new} is valid.
- $\mathsf{Prove}(\mathsf{pk}_\Pi, z^{\mathsf{new}}, w, z^{\mathsf{old}}, \pi^{\mathsf{old}}) \to \pi^{\mathsf{new}}$: on input a proving key pk_Π, new message z^{new}, local data w, and previous message z^{old} with proof π^{old}, Prove outputs a proof π^{new} attesting that this message transition is in compliance with the predicate Π. In the base case, one takes $z^{\mathsf{old}} = \pi^{\mathsf{old}} = \perp$.
- $\mathsf{Verify}(\mathsf{vk}_\Pi, z, \pi) \to b$: on input a verification key vk_Π, message z, and proof π, Verify determines if π is a valid proof attesting that z is the output of a Π-compliant transcript of computation.

The completeness and knowledge soundness properties that we will assume for IVC are stated in Section 6.12.2.[13]

Merkle tree. A Merkle tree (MT) is an authenticated data structure with efficient lookups and updates. We view it as a tuple of algorithms $\mathsf{MT} = (\mathsf{Setup}, \mathsf{New}, \mathsf{Root}, \mathsf{Validate}, \mathsf{Lookup}, \mathsf{VerifyLookup}, \mathsf{Modify}, \mathsf{VerifyModify})$ with the syntax specified below.

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}_{\mathsf{MT}}$: on input a security parameter 1^λ, Setup samples MT public parameters $\mathsf{pp}_{\mathsf{MT}}$.
- $\mathsf{New}(\mathsf{pp}_{\mathsf{MT}}) \to T$: on input MT public parameters $\mathsf{pp}_{\mathsf{MT}}$, New outputs an empty tree T.
- $\mathsf{Root}^T(\mathsf{pp}_{\mathsf{MT}}) \to \mathsf{rh}$: on input MT public parameters $\mathsf{pp}_{\mathsf{MT}}$, with oracle access to the tree T, Root outputs the root hash value of the tree rh.
- $\mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T) \to b$: on input MT public parameters $\mathsf{pp}_{\mathsf{MT}}$ and the tree T, $\mathsf{Validate}$ determines if the Merkle tree is constructed correctly (e.g., the hashes of internal nodes correspond to their children).
- $\mathsf{Lookup}^T(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{addr}_i]_1^n) \to \pi_{\mathsf{lookup}}$: on input MT public parameters $\mathsf{pp}_{\mathsf{MT}}$ and locations $[\mathsf{addr}_i]_1^n$, with oracle access to the tree T, Lookup outputs a proof π_{lookup} certifying the values at those locations.

[13] These definitions can be straightforwardly extended to oracle-based models in those cases where security must hold in the presence of application-specific oracles (e.g., a signing oracle for a signature scheme), by following the analogous case for argument systems in [533].

- $\mathsf{VerifyLookup}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \pi_{\mathsf{lookup}}) \rightarrow b$: on input MT public parameters $\mathsf{pp}_{\mathsf{MT}}$, the tree's root hash rh, the queried locations $[\mathsf{addr}_i]_1^n$, the data at those locations $[\mathsf{data}_i]_1^n$, and the proof π_{lookup}, $\mathsf{VerifyLookup}$ determines if the lookup result is valid.

- $\mathsf{Modify}^T(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n) \rightarrow (\mathsf{rh}, \Delta_T, \pi_{\mathsf{modify}})$: on input MT public parameters $\mathsf{pp}_{\mathsf{MT}}$, the locations to be modified $[\mathsf{addr}_i]_1^n$, and the new data $[\mathsf{data}_i]_1^n$, with oracle access to the tree T, Modify outputs the new root hash rh, the update to the tree Δ_T, and a proof π_{modify}.

- $\mathsf{VerifyModify}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}^{\mathsf{old}}, \mathsf{rh}^{\mathsf{new}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \pi_{\mathsf{modify}}) \rightarrow b$: on input MT public parameters $\mathsf{pp}_{\mathsf{MT}}$, the old tree's root hash $\mathsf{rh}^{\mathsf{old}}$, the new tree's root hash $\mathsf{rh}^{\mathsf{new}}$, the modified locations $[\mathsf{addr}_i]_1^n$, the new data $[\mathsf{data}_i]_1^n$, and the proof π_{modify}, $\mathsf{VerifyModify}$ determines if the modification is valid.

The completeness and security properties that we will assume for MT are stated in Section 6.12.5.

6.11.2 Auxiliary state

In addition to the state S of the original transition function F, the incrementally verifiable transition function vF will maintain an auxiliary state A of the following form

$$A = (t, \mathsf{cm}, \pi_{\mathsf{F}}, T_{\mathsf{state}}, T_{\mathsf{history}}) \ ,$$

where: t records the current time (the number of transactions applied so far); T_{state} is a Merkle tree over S; T_{history} is a Merkle tree over all state commitments so far, ordered chronologically; cm is the current state commitment, and is defined as $\mathsf{cm} := (\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$ where $\mathsf{rh}_{\mathsf{state}}$ is the root hash of T_{state} and $\mathsf{rh}_{\mathsf{history}}$ is the root hash of T_{history}; π_{F} is a proof attesting that "cm commits to a state obtained with t transactions".

6.11.3 Construction of Setup

$\mathsf{Setup}(1^\lambda) \rightarrow \mathsf{pp}$. Run the setup algorithms of the building blocks: $\mathsf{pp}_{\mathsf{MT}} \leftarrow \mathsf{MT.Setup}(1^\lambda)$; $\mathsf{pp}_{\mathsf{IVC}} \leftarrow \mathsf{IVC.Setup}(1^\lambda)$; $\mathsf{pp}_{\mathsf{ARK}} \leftarrow \mathsf{ARK.Setup}(1^\lambda)$. Output the public parameters $\mathsf{pp} := (\mathsf{pp}_{\mathsf{MT}}, \mathsf{pp}_{\mathsf{IVC}}, \mathsf{pp}_{\mathsf{ARK}})$.

6.11.4 Construction of MakeSF

$\mathsf{MakeSF}(\mathsf{pp}, F) \rightarrow (\mathsf{vS}, \mathsf{vF})$. Parse pp as $(\mathsf{pp}_{\mathsf{MT}}, \mathsf{pp}_{\mathsf{IVC}}, \mathsf{pp}_{\mathsf{ARK}})$. Create the IVC predicate Π as defined below, which hardcodes $(\mathsf{pp}_{\mathsf{MT}}, F)$.

$\Pi(z^{\mathsf{new}}, w, z^{\mathsf{old}})$:

- Parse the new message z^{new} as $(t^{\mathsf{new}}, \mathsf{cm}^{\mathsf{new}})$. Parse the commitment $\mathsf{cm}^{\mathsf{new}}$ as $(\mathsf{rh}^{\mathsf{new}}_{\mathsf{state}}, \mathsf{rh}^{\mathsf{new}}_{\mathsf{history}})$.
- Parse the local data w as $(\mathsf{trans}, [\mathsf{raddr}_i]_1^N, [\mathsf{rdata}_i]_1^N, \pi_{\mathsf{read}}, \mathsf{tx}, [\mathsf{waddr}_i]_1^M, [\mathsf{wdata}_i]_1^M, \pi_{\mathsf{write}}, \pi_{\mathsf{insert}})$.
- Check that trans is a valid execution transcript of the transition function F on input tx, read queries $[\mathsf{raddr}_i]_1^N$, corresponding read answers $[\mathsf{rdata}_i]_1^N$, and output state update $\Delta_S = ([\mathsf{waddr}_i]_1^M, [\mathsf{wdata}_i]_1^M)$.
- For the base case ($z^{\mathsf{old}} = \bot$), check the following:
  - (index is initialized correctly) $t^{\mathsf{new}} = 1$;
  - (reads return no data because the initial state is empty) $[\mathsf{rdata}_i]_1^N = \bot$;
  - (writes are performed) $\mathsf{MT.VerifyModify}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}^{\mathsf{empty}}_{\mathsf{state}}, \mathsf{rh}^{\mathsf{new}}_{\mathsf{state}}, [\mathsf{waddr}_i]_1^M, [\mathsf{wdata}_i]_1^M, \pi_{\mathsf{write}}) = 1$, where $\mathsf{rh}^{\mathsf{empty}}_{\mathsf{state}}$ is the root hash of an empty Merkle tree (a tree created by $\mathsf{MT.New}(\mathsf{pp}_{\mathsf{MT}})$).
- Otherwise ($z^{\mathsf{old}} \neq \bot$), parsing the old message z^{old} as $(t^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}})$ and parsing $\mathsf{cm}^{\mathsf{old}}$ as $(\mathsf{rh}^{\mathsf{old}}_{\mathsf{state}}, \mathsf{rh}^{\mathsf{old}}_{\mathsf{history}})$, check the following:
  - (index grows correctly) $t^{\mathsf{new}} = t^{\mathsf{old}} + 1$.
  - (reads are correct) $\mathsf{MT.VerifyLookup}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}^{\mathsf{old}}_{\mathsf{state}}, [\mathsf{raddr}_i]_1^N, [\mathsf{rdata}_i]_1^N, \pi_{\mathsf{read}}) = 1$.
  - (writes are performed) $\mathsf{MT.VerifyModify}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}^{\mathsf{old}}_{\mathsf{state}}, \mathsf{rh}^{\mathsf{new}}_{\mathsf{state}}, [\mathsf{waddr}_i]_1^M, [\mathsf{wdata}_i]_1^M, \pi_{\mathsf{write}}) = 1$.
  - (commitment is added to history) $\mathsf{MT.VerifyModify}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}^{\mathsf{old}}_{\mathsf{history}}, \mathsf{rh}^{\mathsf{new}}_{\mathsf{history}}, t^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}}, \pi_{\mathsf{insert}}) = 1$.

Derive the IVC key pair $(\mathsf{pk}_\Pi, \mathsf{vk}_\Pi) \leftarrow \mathsf{IVC.KeyGen}(\mathsf{pp}_{\mathsf{IVC}}, \Pi)$. Construct and output the tuple $\mathsf{vS} = (\mathsf{Info}, \mathsf{VerifyCM}, \mathsf{VerifyAll})$ as defined below. Note that VerifyCM hardcodes $\mathsf{pp}_{\mathsf{MT}}$ and VerifyAll hardcodes $(\mathsf{pp}_{\mathsf{MT}}, \mathsf{vk}_\Pi)$.

- $\mathsf{vS.Info}^{S,A}() \to (t, \mathsf{cm}, \pi_{\mathsf{F}})$. Retrieve $(t, \mathsf{cm}, \pi_{\mathsf{F}})$ from the auxiliary state A, and output it.
- $\mathsf{vS.VerifyCM}(S, \mathsf{cm}) \to b$. If $S = \bot$ and $\mathsf{cm} = \bot$, output 1. Otherwise, parse cm as $(\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$, and check that $\mathsf{rh}_{\mathsf{state}}$ is the root of a Merkle tree built on the state S.
- $\mathsf{vS.VerifyAll}(S, A) \to b$. For the base case ($A = \bot$), check that $S = \bot$. Otherwise ($A \neq \bot$), parse A as $(t, \mathsf{cm}, \pi_{\mathsf{F}}, T_{\mathsf{state}}, T_{\mathsf{history}})$ and cm as $(\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$. Check that:
  - $\mathsf{vS.VerifyCM}(S, \mathsf{cm}) = 1$;
  - $\mathsf{IVC.Verify}(\mathsf{vk}_\Pi, (t, \mathsf{cm}), \pi_{\mathsf{F}}) = 1$;
  - the state tree T_{state} is well-formed ($\mathsf{MT.Validate}(\mathsf{pp}_{\mathsf{MT}}, T_{\mathsf{state}}) = 1$);
  - the state tree T_{state} has root $\mathsf{rh}_{\mathsf{state}}$ ($\mathsf{rh}_{\mathsf{state}} = \mathsf{MT.Root}^{T_{\mathsf{state}}}(\mathsf{pp}_{\mathsf{MT}})$);
  - the history tree T_{history} is well-formed ($\mathsf{MT.Validate}(\mathsf{pp}_{\mathsf{MT}}, T_{\mathsf{history}}) = 1$);
  - the history tree T_{history} has root $\mathsf{rh}_{\mathsf{history}}$ ($\mathsf{rh}_{\mathsf{history}} = \mathsf{MT.Root}^{T_{\mathsf{history}}}(\mathsf{pp}_{\mathsf{MT}})$).

Construct and output the tuple $\mathsf{vF} = (\mathsf{Run}, \mathsf{Verify})$ as defined below. Note that $\mathsf{vF.Run}$ hardcodes $(\mathsf{pk}_\Pi, \mathsf{pp}_{\mathsf{MT}}, F)$, and $\mathsf{vF.Verify}$ hardcodes vk_Π.

- $\mathsf{vF.Run}^{S,A}(\mathsf{tx}) \to (y, \Delta)$.
  - Simulate $(y, \Delta_S) = F^S(\mathsf{tx})$, obtaining read queries $[\mathsf{raddr}_i]_1^N$ to S and the corresponding read answers $[\mathsf{rdata}_i]_1^N$; let trans denote the execution transcript of F. Parse Δ_S as $([\mathsf{waddr}_i]_1^M, [\mathsf{wdata}_i]_1^M)$.
  - Initialize the update to the auxiliary state $\Delta_A := \perp$.
  - For the base case ($A = \perp$):
    - set $t^{\mathsf{old}} := 0$, $\mathsf{cm}^{\mathsf{old}} := \perp$, $z^{\mathsf{old}} := \perp$, $\pi_{\mathsf{F}}^{\mathsf{old}} := \perp$, $\pi_{\mathsf{read}} := \perp$, and $\pi_{\mathsf{insert}} := \perp$;
    - add to Δ_A the creation of an empty state tree $T_{\mathsf{state}} := \mathsf{MT.New}(\mathsf{pp}_{\mathsf{MT}})$;
    - add to Δ_A the creation of an empty history tree $T_{\mathsf{history}} := \mathsf{MT.New}(\mathsf{pp}_{\mathsf{MT}})$, and set $\mathsf{rh}_{\mathsf{history}}^{\mathsf{new}} := \mathsf{MT.Root}^{T_{\mathsf{history}}}(\mathsf{pp}_{\mathsf{MT}})$.
  - Otherwise ($A \neq \perp$):
    - retrieve $(t^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}}, \pi_{\mathsf{F}}^{\mathsf{old}})$ from the auxiliary state A;
    - compute $\pi_{\mathsf{read}} := \mathsf{MT.Lookup}^{T_{\mathsf{state}}}(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{raddr}_i]_1^N)$;
    - compute $(\mathsf{rh}_{\mathsf{history}}^{\mathsf{new}}, \Delta_T^{\mathsf{history}}, \pi_{\mathsf{insert}}) := \mathsf{MT.Modify}^{T_{\mathsf{history}}}(\mathsf{pp}_{\mathsf{MT}}, t^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}})$, and add $\Delta_T^{\mathsf{history}}$ to Δ_A;
    - set $z^{\mathsf{old}} := (t^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}})$.
  - Compute $(\mathsf{rh}_{\mathsf{state}}^{\mathsf{new}}, \Delta_T^{\mathsf{state}}, \pi_{\mathsf{write}}) := \mathsf{MT.Modify}^{T_{\mathsf{state}}}(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{waddr}_i]_1^M, [\mathsf{wdata}_i]_1^M)$, and add $\Delta_T^{\mathsf{state}}$ to Δ_A.
  - Set the new state index $t^{\mathsf{new}} := t^{\mathsf{old}} + 1$ and new state commitment $\mathsf{cm}^{\mathsf{new}} := (\mathsf{rh}_{\mathsf{state}}^{\mathsf{new}}, \mathsf{rh}_{\mathsf{history}}^{\mathsf{new}})$.
  - Set the new message $z^{\mathsf{new}} := (t^{\mathsf{new}}, \mathsf{cm}^{\mathsf{new}})$.
  - Set the local data $w := (\mathsf{trans}, [\mathsf{raddr}_i]_1^N, [\mathsf{rdata}_i]_1^N, \pi_{\mathsf{read}}, \mathsf{tx}, [\mathsf{waddr}_i]_1^M, [\mathsf{wdata}_i]_1^M, \pi_{\mathsf{write}}, \pi_{\mathsf{insert}})$.
  - Compute the IVC proof $\pi_{\mathsf{F}}^{\mathsf{new}} := \mathsf{IVC.Prove}(\mathsf{pk}_\Pi, z^{\mathsf{new}}, w, z^{\mathsf{old}}, \pi_{\mathsf{F}}^{\mathsf{old}})$.
  - Add to Δ_A the change of the metadata to $(t^{\mathsf{new}}, \mathsf{cm}^{\mathsf{new}}, \pi_{\mathsf{F}}^{\mathsf{new}})$.
  - Output (y, Δ) where $\Delta := (\Delta_S, \Delta_A)$ is the state update to (S, A).

- $\mathsf{vF.Verify}(t, \mathsf{cm}, \pi_{\mathsf{F}}) \to b$. If $t = 0, \mathsf{cm} = \perp, \pi_{\mathsf{F}} = \perp$, output 1. Else output $\mathsf{IVC.Verify}(\mathsf{vk}_\Pi, (t, \mathsf{cm}), \pi_{\mathsf{F}})$.

6.11.5 Construction of MakeC

$\mathsf{MakeC}(\mathsf{pp}, C) \to \mathsf{vC}$. Parse pp as $(\mathsf{pp}_{\mathsf{MT}}, \mathsf{pp}_{\mathsf{IVC}}, \mathsf{pp}_{\mathsf{ARK}})$. Create the NP relation R as defined below.

> The NP relation R, which depends on $(\mathsf{pp}_{\mathsf{MT}}, C)$, considers instances of the form $\mathbb{x} = (\mathsf{cm}, x, y)$ and witnesses of the form $\mathbb{w} = (\mathsf{trans}, [\mathsf{raddr}_i]_1^N, [\mathsf{rdata}_i]_1^N, \pi_{\mathsf{read}})$. A pair $(\mathbb{x}, \mathbb{w})$ is in the relation R if the following conditions hold: (a) trans is a valid execution transcript of C with input x, read queries $[\mathsf{raddr}_i]_1^N$, read answers $[\mathsf{rdata}_i]_1^N$, and output y; (b) if $\mathsf{cm} = \perp$, check that $\mathsf{rdata} = \perp$; (c) if $\mathsf{cm} \neq \perp$, parse cm as $(\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$ and check that $\mathsf{MT.VerifyLookup}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}_{\mathsf{state}}, [\mathsf{raddr}_i]_1^N, [\mathsf{rdata}_i]_1^N, \pi_{\mathsf{read}}) = 1$.

Derive the ARK key pair $(\mathsf{pk}_R, \mathsf{vk}_R) \leftarrow \mathsf{ARK.KeyGen}(\mathsf{pp}_{\mathsf{ARK}}, R)$. Construct and output the tuple $\mathsf{vC} = (\mathsf{Run}, \mathsf{Verify})$ as defined below. Note that vC.Run hardcodes $(\mathsf{pp}_{\mathsf{MT}}, C, \mathsf{pk}_R)$ and vC.Verify hardcodes vk_R.

- $\mathsf{vC.Run}^{S,A}(x) \rightarrow (y, \pi_{\mathsf{C}})$.
  - Simulate the client function as $y = C^S(x)$, obtaining the list of read queries $[\mathsf{raddr}_i]_1^N$ to the state S and the corresponding read answers $[\mathsf{rdata}_i]_1^N$. Also, denote by trans the execution transcript of C.
  - Retrieve the information $(t, \mathsf{cm}, \pi_{\mathsf{F}})$ from the auxiliary state A, and parse cm as $(\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$.
  - If $t = 0$, set $\pi_{\mathsf{read}} := \bot$.
    Otherwise, compute $\pi_{\mathsf{read}} := \mathsf{MT.Lookup}^{T_{\mathsf{state}}}(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{raddr}_i]_1^N)$ where T_{state} is the state tree in A.
  - Set the NP instance $\mathbb{x} := (\mathsf{cm}, x, y)$ and the NP witness $\mathbb{w} := (\mathsf{trans}, [\mathsf{raddr}_i]_1^N, [\mathsf{rdata}_i]_1^N, \pi_{\mathsf{read}})$.
  - Output (y, π_{C}) where the ARK proof is $\pi_{\mathsf{C}} := \mathsf{ARK.Prove}(\mathsf{pk}_R, \mathbb{x}, \mathbb{w})$.
- $\mathsf{vC.Verify}(\mathsf{cm}, x, y, \pi_{\mathsf{C}}) := \mathsf{ARK.Verify}(\mathsf{vk}_R, (\mathsf{cm}, x, y), \pi_{\mathsf{C}})$.

6.11.6 Construction of History

$\mathsf{History.Prove}^{S,A}(\mathsf{pp}, t) \rightarrow \pi_{\mathsf{H}}$. If $t = 0$, output $\pi_{\mathsf{H}} := \bot$. Otherwise, parse pp as $(\mathsf{pp}_{\mathsf{MT}}, \mathsf{pp}_{\mathsf{IVC}}, \mathsf{pp}_{\mathsf{ARK}})$ and retrieve the state commitment $\mathsf{cm} = (\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$ from the auxiliary state A. Compute the proof for location t as $\pi_{\mathsf{H}} := \mathsf{MT.Lookup}^{T_{\mathsf{history}}}(\mathsf{pp}_{\mathsf{MT}}, t)$ where T_{history} is the history tree in A. Output π_{H}.

$\mathsf{History.Verify}(\mathsf{pp}, \mathsf{cm}, t, \mathsf{cm}_t, \pi_{\mathsf{H}}) \rightarrow b$. If $t = 0, \mathsf{cm}_t = \bot, \pi_{\mathsf{H}} = \bot$, return 1. Otherwise, parse pp as $(\mathsf{pp}_{\mathsf{MT}}, \mathsf{pp}_{\mathsf{IVC}}, \mathsf{pp}_{\mathsf{ARK}})$, parse cm as $(\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$, and check that $\mathsf{MT.VerifyLookup}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}_{\mathsf{history}}, t, \mathsf{cm}_t, \pi_{\mathsf{H}}) = 1$.

6.12 Security

Theorem 16. *Our construction of an IVLS scheme in Section 6.11 satisfies the correctness and security properties described in Section 6.3 (assuming that the building blocks* IVC, ARK, *and* MT *that we use in our construction meet standard notions of correctness and security).*

This section is dedicated to proving the above theorem, and is organized to mirror the desired correctness and security properties in Section 6.3. For convenience we include, in the relevant places, formal statements of the properties of IVC and ARK that we use. We separately provide in Section 6.12.5 the (tedious but precise) list of properties of MT that we use.

6.12.1 Security of vS

Lemma 17. The construction of $\mathsf{VerifyCM}$ in Section 6.11.4 satisfies the binding property (Definition 9).

Proof. Suppose that $\mathcal{A}$ is an adversary that, given as input $\mathsf{pp} = (\mathsf{pp}_{\mathsf{MT}}, \mathsf{pp}_{\mathsf{IVC}}, \mathsf{pp}_{\mathsf{ARK}}) \in \mathsf{Setup}(1^\lambda)$, outputs a transition function F, state commitment cm, and two states S_1 and S_2 such that $\mathsf{vS}.\mathsf{VerifyCM}(S_1, \mathsf{cm}) = 1$ and $\mathsf{vS}.\mathsf{VerifyCM}(S_2, \mathsf{cm}) = 1$ where vS is produced by $\mathsf{MakeSF}(\mathsf{pp}, F)$. We argue that $S_1 = S_2$ except with negligible probability. We assume that $\mathsf{cm} \neq \bot$; otherwise we are done.

We use $\mathcal{A}$ to construct an adversary $\mathcal{A}'$ that breaks the binding property of root hashes (Definition 33). On input $\mathsf{pp}_{\mathsf{MT}} \in \mathsf{MT}.\mathsf{Setup}(1^\lambda)$, $\mathcal{A}'$ works as follows:

– Sample $\mathsf{pp}_{\mathsf{IVC}} \leftarrow \mathsf{IVC}.\mathsf{Setup}(1^\lambda)$ and $\mathsf{pp}_{\mathsf{ARK}} \leftarrow \mathsf{ARK}.\mathsf{Setup}(1^\lambda)$.
– Assemble $\mathsf{pp} := (\mathsf{pp}_{\mathsf{MT}}, \mathsf{pp}_{\mathsf{IVC}}, \mathsf{pp}_{\mathsf{ARK}})$.
– Compute $(F, S_1, S_2, \mathsf{cm}) := \mathcal{A}(\mathsf{pp})$.
– Construct Merkle trees T_1 from S_1 and T_2 from S_2.
– Compute $\mathsf{rh} \leftarrow \mathsf{MT}.\mathsf{Root}^{T_1}(\mathsf{pp}_{\mathsf{MT}})$.
– Output (T_1, T_2, rh).

Note that if the two states S_1 and S_2 are distinct, then so are the corresponding (valid) Merkle trees T_1 and T_2. Recall that cm is a pair of root hashes $(\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$. Note that $\mathsf{vS}.\mathsf{VerifyCM}(S_1, \mathsf{cm}) = 1$ implies that the Merkle tree on S_1 has root hash $\mathsf{rh}_{\mathsf{state}}$, and similarly the Merkle tree on S_2 has the same root hash $\mathsf{rh}_{\mathsf{state}}$. This means that whenever $\mathcal{A}$ outputs distinct states with the same commitment, $\mathcal{A}'$ outputs distinct Merkle trees with the same root hash, which can happen with at most negligible probability. □

6.12.2 Security of vF

The construction of vF appears in Section 6.11.4, and its two main subroutines are MT (a Merkle tree) and IVC (incrementally verifiable computation). We state the properties of IVC that we use (omitting the tuple prefix IVC), and then prove the security properties of vF. We remark that the formalization of IVC that we use follows as a straightforward special case of definitions of proof-carrying data [477, 478].

- An *IVC predicate* Π is an algorithm that takes as input the new step's output z^{new}, the new step's auxiliary input w, and the previous step's output z^{old} ($\bot$ if the new step is the first step) as input, and outputs 1 or 0.

- An *IVC transcript* is a tuple $\mathsf{T} = ([w_i]_1^n, [z_i]_1^n)$ where w_i is the auxiliary input to the i-th step and z_i is the output after the i-th step. The *output* of T, denoted by $\mathsf{out}(\mathsf{T})$, is z_n. We say that $\mathsf{T} = ([w_i]_1^n, [z_i]_1^n)$ is Π-*compliant*, denoted by $\Pi(\mathsf{T}) = 1$, if $\Pi(z_i, w_i, z_{i-1}) = 1$ for every $i \in [n]$ with $z_0 := \bot$.

Definition 18 (completeness of IVC)**.** For every polynomial-size adversary $\mathcal{A}$ and security parameter λ,

$$\Pr\left[\begin{array}{c|r} (z^{\mathsf{old}} = \bot \ \vee \ \mathsf{Verify}(\mathsf{vk}_\Pi, z^{\mathsf{old}}, \pi^{\mathsf{old}}) = 1) & \mathsf{pp}_{\mathsf{IVC}} \leftarrow \mathsf{Setup}(1^\lambda) \\ \Pi(z^{\mathsf{new}}, w, z^{\mathsf{old}}) = 1 & (\Pi, z^{\mathsf{new}}, w, z^{\mathsf{old}}, \pi^{\mathsf{old}}) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{IVC}}) \\ \downarrow & (\mathsf{pk}_\Pi, \mathsf{vk}_\Pi) \leftarrow \mathsf{KeyGen}(\mathsf{pp}_{\mathsf{IVC}}, \Pi) \\ \mathsf{Verify}(\mathsf{vk}_\Pi, z^{\mathsf{new}}, \pi^{\mathsf{new}}) = 1 & \pi^{\mathsf{new}} \leftarrow \mathsf{Prove}(\mathsf{pk}_\Pi, z^{\mathsf{new}}, w, z^{\mathsf{old}}, \pi^{\mathsf{old}}) \end{array}\right] = 1 \ .$$

Definition 19 (knowledge soundness of IVC)**.** For every polynomial-size adversary $\mathcal{A}$, there exists a polynomial-size extractor $\mathcal{E}$ such that for every sufficiently large security parameter λ and every auxiliary input $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$,

$$\Pr\left[\begin{array}{c|r} \mathsf{Verify}(\mathsf{vk}_\Pi, z, \pi) = 1 & \mathsf{pp}_{\mathsf{IVC}} \leftarrow \mathsf{Setup}(1^\lambda) \\ \downarrow & (\Pi, z, \pi) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{IVC}}, \mathsf{aux}) \\ \mathsf{out}(\mathsf{T}) = z & \mathsf{T} \leftarrow \mathcal{E}(\mathsf{pp}_{\mathsf{IVC}}, \mathsf{aux}) \\ \Pi(\mathsf{T}) = 1 & (\mathsf{pk}_\Pi, \mathsf{vk}_\Pi) \leftarrow \mathsf{KeyGen}(\mathsf{pp}_{\mathsf{IVC}}, \Pi) \end{array}\right] \geq 1 - \mathrm{negl}(\lambda) \ .$$

Lemma 20. The construction of vF in Section 6.11.4 satisfies the completeness property (Definition 10).

Proof. Our construction of $\mathsf{vF.Run}$ straightforwardly ensures that $(y, \Delta_S) = F^S(\mathsf{tx})$ and $t' = t+1$. We are then only left to argue that $\mathsf{vS.VerifyAll}(S, A) = 1$ implies $\mathsf{vS.VerifyAll}(S', A') = 1$, where (S', A') are the updated states. The discussion below assumes familiarity with our construction of $\mathsf{vS.VerifyAll}$. It suffices to argue the following points: (a) the IVC proof in A' is valid; (b) the Merkle trees in A' are well-formed; (c) the commitment in A' contains root hashes of the Merkle trees; (d) the state Merkle tree is over S'.

(a) IVC proof is valid. We argue that $\mathsf{IVC.Verify}(\mathsf{vk}_\Pi, (t^{\mathsf{new}}, \mathsf{cm}^{\mathsf{new}}), \pi^{\mathsf{new}}) = 1$ where $(t^{\mathsf{new}}, \mathsf{cm}^{\mathsf{new}}, \pi^{\mathsf{new}})$ are in the new auxiliary state A'. Observe that the condition $\mathsf{vS.VerifyAll}(S, A) = 1$ implies that $S = A = \bot$ or $\mathsf{IVC.Verify}(\mathsf{vk}_\Pi, (t^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}}), \pi^{\mathsf{old}}) = 1$ where $(t^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}}, \pi^{\mathsf{old}})$ are in A. Therefore, by the completeness property of IVC (Definition 18), we are left to argue that $\Pi((t^{\mathsf{new}}, \mathsf{cm}^{\mathsf{new}}), w, z^{\mathsf{old}}) = 1$ where $z^{\mathsf{old}} = \bot$ if $A = \bot$ or $z^{\mathsf{old}} = (t^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}})$ if $A \neq \bot$, for the local data $w = (\mathsf{trans}, [\mathsf{raddr}_i]_1^N, [\mathsf{rdata}_i]_1^N, \pi_{\mathsf{read}}, \mathsf{tx}, [\mathsf{waddr}_i]_1^M, [\mathsf{wdata}_i]_1^M, \pi_{\mathsf{write}}, \pi_{\mathsf{insert}})$ supplied by $\mathsf{vF.Run}$. We discuss these two cases separately.

- For the base case ($z^{\mathsf{old}} = \bot$), Π is satisfied because $\mathsf{vF.Run}$ ensures the following conditions:
  - (index is correct) $t^{\mathsf{new}} = 1$.
  - (execution is correct) trans is a valid transcript of F over an empty state.
  - (reads are correct) $[\mathsf{rdata}_i]_1^N = \bot$.
  - (writes are performed) π_{write} proves that T_{state} is updated according to trans.
  - (commitment is correct) $\mathsf{cm}^{\mathsf{new}}$ matches the new Merkle trees.

- Otherwise ($z^{\mathsf{old}} \neq \bot$), Π is satisfied because $\mathsf{vF.Run}$ ensures the following conditions:
  - (index is correct) $t^{\mathsf{new}} = t^{\mathsf{old}} + 1$.

- (execution is correct) trans is a valid transcript of F over $([\mathsf{raddr}_i]_1^N, [\mathsf{rdata}_i]_1^N)$.
- (reads are correct) π_{read} proves that $([\mathsf{raddr}_i]_1^N, [\mathsf{rdata}_i]_1^N)$ matches the state committed by $\mathsf{cm}^{\mathsf{old}}$.
- (writes are performed) π_{write} proves that T_{state} is updated according to trans.
- (commitment is added to history) π_{insert} proves that T_{history} now has $\mathsf{cm}^{\mathsf{old}}$ at position t^{old}.
- (commitment is correct) $\mathsf{cm}^{\mathsf{new}}$ matches the updated Merkle trees.

(b) Merkle trees are well-formed. Completeness properties of Merkle trees (Definitions 28 and 30) ensure that changes from the old state S to the new state S' have been applied correctly to the state tree in T_{state} in A, and also that the insertion of the commitment to the history tree T_{history} has been applied correctly.

(c) State commitment contains correct root hashes. Our construction of $\mathsf{vF.Run}$ stores the new commitment $\mathsf{cm}^{\mathsf{new}}$ in the new auxiliary state A', so this condition holds.

(d) New state Merkle tree is over new state. The changes to the state S, $([\mathsf{waddr}_i]_1^M, [\mathsf{wdata}_i]_1^M)$, are converted into Merkle tree update $\Delta_T^{\mathsf{state}}$ applied to A, so the new state Merkle tree is over the new state. □

Lemma 21. The construction of vF in Section 6.11.4 satisfies the knowledge soundness property (Definition 11).

Proof. Let $\mathcal{A}$ be an adversary against the knowledge soundness of vF (Definition 11). This means that on input public parameters $\mathsf{pp} \in \mathsf{Setup}(1^\lambda)$, $\mathcal{A}$ outputs a tuple $(F, t, \mathsf{cm}, \pi_{\mathsf{F}})$. We assume that $t > 0$; otherwise there is nothing to prove.

We construct an adversary $\mathcal{A}'$ against the knowledge soundness of IVC (Definition 19), using an auxiliary input aux that is interpreted as $(\mathsf{pp_{MT}}, \mathsf{pp_{ARK}})$, as follows:

> $\mathcal{A}'$ receives as input public parameters $\mathsf{pp_{IVC}} \in \mathsf{IVC.Setup}(1^\lambda)$ and the auxiliary input aux, then it works as follows: (1) parse aux as $(\mathsf{pp_{MT}}, \mathsf{pp_{ARK}})$ and assemble $\mathsf{pp} = (\mathsf{pp_{MT}}, \mathsf{pp_{IVC}}, \mathsf{pp_{ARK}})$; (2) compute $(F, t, \mathsf{cm}, \pi_{\mathsf{F}}) := \mathcal{A}(\mathsf{pp})$; (3) use $(\mathsf{pp_{MT}}, F)$ to create the IVC predicate Π as in Section 6.11.4; (4) create the IVC message $z := (t, \mathsf{cm})$; (5) create the IVC proof $\pi := \pi_{\mathsf{F}}$; (6) output (Π, z, π).

Recall that, for $t > 0$, $\mathsf{vF.Verify}(t, \mathsf{cm}, \pi_{\mathsf{F}}) = \mathsf{IVC.Verify}(\mathsf{vk}_\Pi, (t, \mathsf{cm}), \pi_{\mathsf{F}})$ where vk_Π is hardcoded in vF. This means that $\mathcal{A}'$ produces an accepting output whenever $\mathcal{A}$ does.

Let $\mathcal{E}'$ be the extractor for $\mathcal{A}'$ guaranteed by the knowledge soundness property of IVC (Definition 19). We construct an extractor $\mathcal{E}$ for $\mathcal{A}$:

> $\mathcal{E}$ receives as input public parameters $\mathsf{pp} \in \mathsf{Setup}(1^\lambda)$ and then works as follows: (1) parse $\mathsf{pp} = (\mathsf{pp_{MT}}, \mathsf{pp_{IVC}}, \mathsf{pp_{ARK}})$; (2) set the auxiliary input $\mathsf{aux} := (\mathsf{pp_{MT}}, \mathsf{pp_{ARK}})$; (3) compute the IVC transcript $\mathsf{T} = ([w_i]_1^n, [z_i]_1^n) := \mathcal{E}'(\mathsf{pp_{IVC}}, \mathsf{aux})$; (4) for each i, find in the local data w_i a transaction tx_i (in the execution transcript of F contained in w_i); (5) output all transactions $[\mathsf{tx}_i]_1^n$.

We are left to argue that the extractor $\mathcal{E}$ works for $\mathcal{A}$.

The knowledge soundness property of IVC tells us that the IVC transcript output by $\mathcal{E}'$ is Π-compliant and ends in the message z, whenever the output (Π, z, π) of $\mathcal{A}'$ is accepting. This, together with the binding properties of Merkle trees, tells us that the state $S := F(\mathsf{tx}_1, \ldots, \mathsf{tx}_n)$ is such that $\mathsf{vS.VerifyCM}(S, \mathsf{cm}) = 1$, whenever the output $(F, t, \mathsf{cm}, \pi_{\mathsf{F}})$ of $\mathcal{A}$ is accepting. □

6.12.3 Security of vC

The construction of vC appears in Section 6.11.5, and its two main subroutines are MT (a Merkle tree) and ARK (a non-interactive argument of knowledge). We state the properties of ARK that we use (omitting the tuple prefix ARK), and then prove the security properties of vC.

Definition 22 (completeness of ARK)**.** For every polynomial-size adversary $\mathcal{A}$ and security parameter λ,

$$\Pr\left[\begin{array}{c|r} (\mathbb{x}, \mathbb{w}) \in R & \mathsf{pp}_{\mathsf{ARK}} \leftarrow \mathsf{Setup}(1^\lambda) \\ \downarrow & (R, \mathbb{x}, \mathbb{w}) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{ARK}}) \\ \mathsf{Verify}(\mathsf{vk}_R, \mathbb{x}, \pi) = 1 & (\mathsf{pk}_R, \mathsf{vk}_R) \leftarrow \mathsf{KeyGen}(\mathsf{pp}_{\mathsf{ARK}}, R) \\ & \pi \leftarrow \mathsf{Prove}(\mathsf{pk}_R, \mathbb{x}, \mathbb{w}) \end{array}\right] = 1 \ .$$

Definition 23 (knowledge soundness of ARK)**.** For every polynomial-size adversary $\mathcal{A}$ there exists a polynomial-size extractor $\mathcal{E}$ such that for every sufficiently large security parameter λ,

$$\Pr\left[\begin{array}{c|r} \mathsf{Verify}(\mathsf{vk}_R, \mathbb{x}, \pi) = 1 & \mathsf{pp}_{\mathsf{ARK}} \leftarrow \mathsf{Setup}(1^\lambda) \\ \downarrow & (R, \mathbb{x}, \pi) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{ARK}}) \\ (\mathbb{x}, \mathbb{w}) \in R & \mathbb{w} \leftarrow \mathcal{E}(\mathsf{pp}_{\mathsf{ARK}}) \\ & (\mathsf{pk}_R, \mathsf{vk}_R) \leftarrow \mathsf{KeyGen}(\mathsf{pp}_{\mathsf{ARK}}, R) \end{array}\right] \geq 1 - \mathrm{negl}(\lambda) \ .$$

Lemma 24. The construction of vC in Section 6.11.5 satisfies the completeness property (Definition 12).

Proof. First note that $\mathsf{vC.Run}$ produces the output y by running C on input x with state S, and so clearly the condition $y = C^S(x)$ holds. Next we argue that, if the auxiliary state A is such that $\mathsf{vS.VerifyAll}(S, A) = 1$, then the proof π_{C} produced by $\mathsf{vC.Run}$ will be accepted by $\mathsf{vC.Verify}$. Since the proof π_{C} is produced and validated using the prover and verifier of ARK, by the completeness of ARK (Definition 22), it suffices to argue that the instance-witness pair constructed by $\mathsf{vC.Run}$ is in the NP relation R.

To see this, recall that R checks that the witness $\mathbb{w}$ for an instance $\mathbb{x} = (\mathsf{cm}, x, y)$ contains a valid execution transcript of a computation of C on input x leading to output y, with read queries and read answers authenticated against the state root hash contained in the state commitment $\mathsf{cm} = (\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$ (if any). The condition $\mathsf{vS.VerifyAll}(S, A) = 1$ in the completeness property ensures that T_{state} in A is a Merkle tree over the state S, and $\mathsf{cm} = (\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$ in A is such that $\mathsf{rh}_{\mathsf{state}}$ is the root of this Merkle tree. This means that the read queries and read answers are consistent with the state S, and the Merkle tree lookup proof placed in the witness $\mathbb{w}$ by $\mathsf{vC.Run}$ is valid (by the completeness of Merkle tree lookups, Definition 29). □

Lemma 25. The construction of vC in Section 6.11.5 satisfies the soundness property (Definition 13).

Proof. Suppose that an adversary outputs (S, A) such that $\mathsf{vS.VerifyAll}(S, A) = 1$ and also (x, y, π_{C}) such that, for the state commitment cm in A, it holds that $\mathsf{vC.Verify}(\mathsf{cm}, x, y, \pi_{\mathsf{C}}) = 1$. By construction, we know that the verifier of ARK accepts the instance $\mathbb{x} = (\mathsf{cm}, x, y)$ for the NP relation R.

By the knowledge soundness of ARK (Definition 23), there exists a corresponding extractor that outputs a valid witness $\mathbb{w}$ for $\mathbb{x}$, which contains a valid execution transcript of a computation of C on input x leading to output y, with read queries and read answers authenticated against the state root hash contained in the state commitment $\mathsf{cm} = (\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$ (if any). But we know from $\mathsf{VerifyAll}(S, A)$ that $\mathsf{rh}_{\mathsf{state}}$ is the root of a Merkle tree over the state S and so, by the soundness of Merkle tree lookup proofs (Definition 34), we know that the read queries and read answers are consistent with S, and conclude that $y = C^S(x)$. □

6.12.4 Security of History

Lemma 26. The construction of $\mathsf{History}$ in Section 6.11.6 satisfies the completeness property (Definition 14).

Proof. Suppose that an adversary outputs (S, A) such that $\mathsf{vS.VerifyAll}(S, A) = 1$ and also transactions $(\mathsf{tx}_1, \ldots, \mathsf{tx}_n)$. Let t be the state index and cm_t the state commitment contained in the auxiliary state A. The condition $\mathsf{vS.VerifyAll}(S, A) = 1$ implies that the history Merkle tree T_{history} in A is well-formed, and its root is stored in cm_t.

When $\mathsf{vF.Run}$ is invoked to apply the first of these transactions, cm_t will be placed at location t in the history Merkle tree. When $\mathsf{vF.Run}$ is invoked to apply each of the other transactions, the t-th location in the history Merkle tree continues to store cm_t (while subsequent locations in the history Merkle tree are allocated to store the subsequent commitments). This is due to the completeness of Merkle tree modifications (Definitions 31 and 32).

This means that $\mathsf{History.Verify}$ will accept the history proof. Indeed, $\mathsf{History.Prove}$ produces a history proof π_{H} for a state index t by authenticating cm_t as being at location t in the current history Merkle tree, and $\mathsf{History.Verify}$ validates a history proof by checking this authentication against the history root hash in the current state commitment. The completeness property of Merkle tree lookups (Definition 29) implies that $\mathsf{History.Verify}$ accepts. □

Lemma 27. The construction of $\mathsf{History}$ in Section 6.11.6 satisfies the soundness property (Definition 15).

Proof. Suppose that an adversary outputs a state commitment cm, state index t, two state commitments cm_t and cm'_t, and two history proofs π_{H} and π'_{H} such that $\mathsf{History.Verify}(\mathsf{pp}, \mathsf{cm}, t, \mathsf{cm}_t, \pi_{\mathsf{H}}) = 1$ and $\mathsf{History.Verify}(\mathsf{pp}, \mathsf{cm}, t, \mathsf{cm}'_t, \pi'_{\mathsf{H}}) = 1$. We argue that, except with negligible probability, $\mathsf{cm}_t = \mathsf{cm}'_t$.

If $t = 0$, then we know by construction of $\mathsf{History.Verify}$ that it is always the case that $\mathsf{cm}_t = \mathsf{cm}'_t = \bot$.

If $t > 0$, $\mathsf{History.Verify}$ parses cm as a tuple $(\mathsf{rh}_{\mathsf{state}}, \mathsf{rh}_{\mathsf{history}})$ and checks that the history proof authenticates the past state commitment as being at location t of a Merkle tree with root $\mathsf{rh}_{\mathsf{history}}$. This means that the adversary has output proofs π_{H} and π'_{H} that respectively authenticate cm_t and cm'_t for the same location t of a Merkle tree with the same root $\mathsf{rh}_{\mathsf{history}}$. By the binding property (Definition 33) and soundness of lookups (Definition 34) of Merkle trees, we know that $\mathsf{cm}_t = \mathsf{cm}'_t$ except with negligible probability. □

6.12.5 Merkle tree properties

We formally state the properties of a Merkle tree that we use. We omit the tuple prefix MT in the text below.

Definition 28 (completeness for an empty tree)**.** For every security parameter λ,

$$\Pr\left[\ \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T) = 1 \ \middle|\ \begin{array}{c} \mathsf{pp}_{\mathsf{MT}} \leftarrow \mathsf{Setup}(1^{\lambda}) \\ T \leftarrow \mathsf{New}(\mathsf{pp}_{\mathsf{MT}}) \end{array} \right] = 1 \ .$$

Definition 29 (completeness for a lookup)**.** For every polynomial-size adversary $\mathcal{A}$ and security parameter λ,

$$\Pr\left[\begin{array}{c} \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T) = 1 \\ (\mathsf{addr}_i, \mathsf{data}_i) \in T \text{ for } i \in [n] \\ \downarrow \\ b = 1 \end{array} \ \middle|\ \begin{array}{r} \mathsf{pp}_{\mathsf{MT}} \leftarrow \mathsf{Setup}(1^{\lambda}) \\ (T, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{MT}}) \\ \pi \leftarrow \mathsf{Lookup}^T(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{addr}_i]_1^n) \\ \mathsf{rh} \leftarrow \mathsf{Root}^T(\mathsf{pp}_{\mathsf{MT}}) \\ b \leftarrow \mathsf{VerifyLookup}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \pi) \end{array} \right] = 1 \ .$$

Definition 30 (completeness for a modification)**.** For every polynomial-size adversary $\mathcal{A}$ and security parameter λ,

$$\Pr\left[\begin{array}{c} \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T^{\mathsf{old}}) = 1 \\ \downarrow \\ \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T^{\mathsf{new}}) = 1 \\ \mathsf{rh}^{\mathsf{new}} = \mathsf{Root}^{T^{\mathsf{new}}}(\mathsf{pp}_{\mathsf{MT}}) \\ b = 1 \end{array} \ \middle|\ \begin{array}{r} \mathsf{pp}_{\mathsf{MT}} \leftarrow \mathsf{Setup}(1^{\lambda}) \\ (T^{\mathsf{old}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{MT}}) \\ \mathsf{rh}^{\mathsf{old}} \leftarrow \mathsf{Root}^{T^{\mathsf{old}}}(\mathsf{pp}_{\mathsf{MT}}) \\ (\mathsf{rh}^{\mathsf{new}}, \Delta_T, \pi) \leftarrow \mathsf{Modify}^{T^{\mathsf{old}}}(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n) \\ T^{\mathsf{new}} := T^{\mathsf{old}} + \Delta_T \\ b \leftarrow \mathsf{VerifyModify}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}^{\mathsf{old}}, \mathsf{rh}^{\mathsf{new}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \pi) \end{array} \right] = 1 \ .$$

Definition 31 (completeness for a modification on the modified location)**.** For every polynomial-size adversary $\mathcal{A}$ and security parameter λ,

$$\Pr\left[\begin{array}{c} \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T^{\mathsf{old}}) = 1 \\ \mathsf{addr}_q \in [\mathsf{addr}_i]_1^n \\ \downarrow \\ \mathsf{addr}_q \text{ is up-to-date in } T^{\mathsf{new}} \end{array} \ \middle|\ \begin{array}{r} \mathsf{pp}_{\mathsf{MT}} \leftarrow \mathsf{Setup}(1^{\lambda}) \\ (T^{\mathsf{old}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \mathsf{addr}_q) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{MT}}) \\ (\cdot, \Delta_T, \cdot) \leftarrow \mathsf{Modify}^{T^{\mathsf{old}}}(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n) \\ T^{\mathsf{new}} := T^{\mathsf{old}} + \Delta_T \end{array} \right] = 1 \ .$$

where "addr_q is up-to-date" means that the highest index $1 \leq j \leq N$ such that $\mathsf{addr}_j = \mathsf{addr}_q$ in $[\mathsf{addr}_i]_1^n$ satisfies that data_j in $[\mathsf{data}_i]_1^n$ matches the value at addr_q in T^{new}.

Definition 32 (completeness for a modification on an unmodified location)**.** For every polynomial-size adversary $\mathcal{A}$ and security parameter λ,

$$\Pr\left[\begin{array}{c|r} \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T^{\mathsf{old}}) = 1 & \mathsf{pp}_{\mathsf{MT}} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{addr}_q \notin [\mathsf{addr}_i]_1^n & (T^{\mathsf{old}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \mathsf{addr}_q) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{MT}}) \\ \downarrow & (\cdot, \Delta_T, \cdot) \leftarrow \mathsf{Modify}^{T^{\mathsf{old}}}(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n) \\ T^{\mathsf{old}} \text{ and } T^{\mathsf{new}} \text{ is the same at } \mathsf{addr}_q & T^{\mathsf{new}} := T^{\mathsf{old}} + \Delta_T \end{array}\right] = 1 \enspace .$$

Definition 33 (binding of root hash)**.** For every polynomial-size adversary $\mathcal{A}$ and sufficiently large security parameter λ,

$$\Pr\left[\begin{array}{c|c} \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T_1) = 1 & \\ \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T_2) = 1 & \\ \mathsf{Root}^{T_1}(\mathsf{pp}_{\mathsf{MT}}) = \mathsf{rh} & \mathsf{pp}_{\mathsf{MT}} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{Root}^{T_2}(\mathsf{pp}_{\mathsf{MT}}) = \mathsf{rh} & (T_1, T_2, \mathsf{rh}) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{MT}}) \\ \downarrow & \\ T_1 = T_2 & \end{array}\right] \geq 1 - \mathrm{negl}(\lambda) \enspace .$$

Definition 34 (soundness of a lookup)**.** For every polynomial-size adversary $\mathcal{A}$ and sufficiently large security parameter λ,

$$\Pr\left[\begin{array}{c|r} \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T) = 1 & \mathsf{pp}_{\mathsf{MT}} \leftarrow \mathsf{Setup}(1^\lambda) \\ b = 1 & (T, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \pi) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{MT}}) \\ \downarrow & \mathsf{rh} \leftarrow \mathsf{Root}^T(\mathsf{pp}_{\mathsf{MT}}) \\ (\mathsf{addr}_i, \mathsf{data}_i) \in T \text{ for } i \in [n] & b \leftarrow \mathsf{VerifyLookup}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \pi) \end{array}\right] \geq 1 - \mathrm{negl}(\lambda) \enspace .$$

Definition 35 (soundness of a modification)**.** For every polynomial-size adversary $\mathcal{A}$ and sufficiently large security parameter λ,

$$\Pr\left[\begin{array}{c|r} \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T^{\mathsf{old}}) = 1 & \mathsf{pp}_{\mathsf{MT}} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{Validate}(\mathsf{pp}_{\mathsf{MT}}, T^{\mathsf{new}}) = 1 & (T^{\mathsf{old}}, T^{\mathsf{new}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \pi) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{MT}}) \\ b = 1 & (\mathsf{rh}^{\mathsf{new}}, \Delta_T, \cdot) \leftarrow \mathsf{Modify}^{T^{\mathsf{old}}}(\mathsf{pp}_{\mathsf{MT}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n) \\ \downarrow & \mathsf{rh}^{\mathsf{old}} \leftarrow \mathsf{Root}^{T^{\mathsf{old}}}(\mathsf{pp}_{\mathsf{MT}}) \\ T^{\mathsf{new}} := T^{\mathsf{old}} + \Delta_T & b \leftarrow \mathsf{VerifyModify}(\mathsf{pp}_{\mathsf{MT}}, \mathsf{rh}^{\mathsf{old}}, \mathsf{rh}^{\mathsf{new}}, [\mathsf{addr}_i]_1^n, [\mathsf{data}_i]_1^n, \pi_{\mathsf{modify}}) \end{array}\right]$$
$$\geq 1 - \mathrm{negl}(\lambda) \enspace .$$

Chapter 7

N-for-1 Auth: N-wise Decentralized Authentication via One Authentication

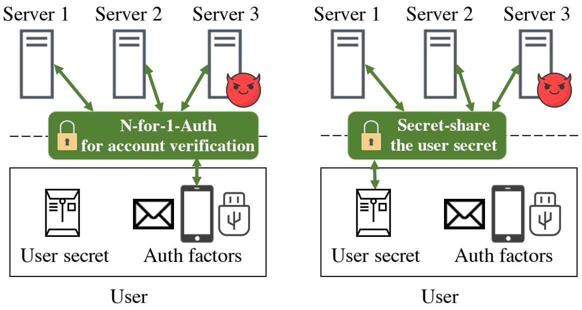
Decentralizing trust is a fundamental principle in the design of end-to-end encryption and cryptocurrency systems. A common issue in these applications is that users possess critical secrets. If these secrets are lost, users can lose precious data or assets. This issue remains a pain point in the adoption of these systems. Existing approaches such as backing up user secrets through a centralized service or distributing them across N mutually distrusting servers to preserve decentralized trust are either introducing a central point of attack or face usability issues by requiring users to authenticate N times, once to each of the N servers.

We present N-for-1-Auth, a system that preserves distributed trust by enabling a user to authenticate to N servers independently, with the work of only one authentication, thereby offering the same user experience as in a typical centralized system.

7.1 Introduction

Decentralizing trust is a fundamental principle in designing modern security applications. For example, there is a proliferation of end-to-end encrypted systems and cryptocurrencies, which aim to remove a central point of trust [136, 463, 534–542]. In these applications, users find themselves owning critical secrets, such as the secret keys to decrypt end-to-end encrypted data or the secret keys to spend digital assets. If these secrets are lost, the user permanently loses access to his/her precious data or assets.

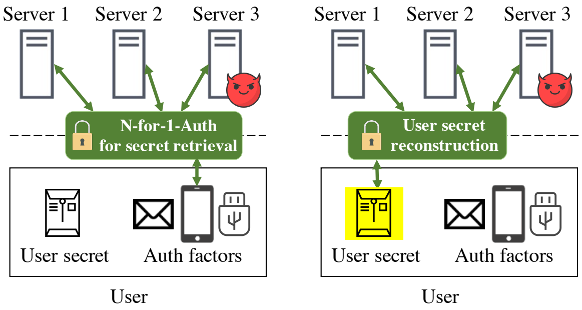
To explain the problem that N-for-1-Auth addresses, let us take the example of Alice, a user of an end-to-end encryption application denoted as the “E2EE App” (or similarly, a cryptocurrency system). For such an E2EE App, Alice installs an E2EE App Client on her device, such as her cell phone. The client holds her secret key to decrypt her data. For the sake of usability and adoption, Alice should not have to deal with backing up the key herself. We are concerned with the situation when Alice loses her cell phone. Though she can get a new SIM card by contacting the provider, she loses the secret keys stored on the phone. With WhatsApp [536] and Line [539], end-



(a) Servers authenticate the user through authentication factors.

(b) The user secret-shares the user secret among the servers.

Figure 7.1: Enrollment workflow.



(a) Servers authenticate the user via authentication factors.

(b) The user reconstructs the secret from shares.

Figure 7.2: Authentication workflow.

to-end encrypted chat applications, Alice can use centralized services such as Google Drive and iCloud to backup her chat history. However, such a strategy jeopardizes the end-to-end encryption guarantees of these systems because users' chats become accessible to services that are central points of attack. This is further reaffirmed by Telegram's CEO Pavel Durov who said in a blog post: *"(Centralized backup) invalidates end-to-end encryption for 99% of private conversations"*. To preserve decentralized trust, many companies [136, 543–547] and academic works [548–551] have proposed to secret-share the user's secrets across N servers, so that compromising some of the servers does not reveal her secrets.

However, a significant issue with this approach is the burden of authentication. After Alice loses her cell phone with all her secrets for the E2EE App, she can only authenticate with other factors, such as email, short message services (SMS), universal second-factor (U2F), and security questions. How does Alice authenticate to the N servers to retrieve her secret? If Alice authenticates to only one server and the other servers trust this server, the first server now becomes a central point of attack. To avoid centralized trust, as the N servers cannot trust each other, Alice has to authenticate to each server separately. For email verification, Alice has to perform N times the work—reading N emails. To avoid a central point of attack, the E2EE App should require multiple factors, which further multiplies Alice's effort.

One might think that doing N times the work, albeit undesirable for the user, is acceptable in catastrophic situations such as losing one's devices. The issue here is that Alice has to perform this work not only when she is recovering her secrets, but also *when she is joining the system,* because her key's secret shares must be registered with the N servers using the multiple factors of authentication, and those servers must check that Alice indeed is the person controlling those factors. Even for $N = 2$ in which there is only one additional email and text message, it is a completely different user experience that adds friction to a space already plagued by usability issues (e.g., "Why Johnny can't encrypt?" [552, 553]). Many academic works [554, 555] reaffirm the importance of the consistency of user experience and minimizing user effort for better adoption.

Therefore, this initial bar of entry is a deterrent against widespread adoption. To validate that this is an important and recurring problem, we presented N-for-1-Auth to prominent companies in the end-to-end encryption and cryptocurrency custody spaces, who supported our thesis. We summarize their feedback in Section 7.1.2.

A few strawman designs seem to address this burden for Alice but are actually unviable. One strawman is to build a client app that automatically performs the N authentications for Alice. In the case of email/SMS authentication, the client app parses the emails or text messages Alice receives from the N servers. However, this either requires the client app to have intrusive permissions (e.g., reading Alice's email) that can affect the security of other applications Alice uses or requires very specific APIs available on the email/SMS server side (e.g., Gmail offering such APIs), which do not exist today for major providers and we find unreasonable. Another strawman [548–551, 556] is to have Alice possess or remember a master secret and then authenticate to each of the N servers by deriving a unique secret to each server, thereby avoiding the issue of having to do the work surrounding email/SMS authentication. However, Alice has to then safeguard this secret, as losing it could lead to an attacker impersonating her to the N servers. In this case, we return to the original problem of Alice needing a reliable way to store this authentication secret.

7.1.1 N-for-1-Auth

We present N-for-1-Auth, which alleviates this burden by enabling a user to authenticate to *N servers independently* by doing only the work of authenticating with *one*, as illustrated in Figures 7.1 and 7.2. This matches the usual experience of authenticating to an application with centralized trust.

N-for-1-Auth supports many authentication factors that users are accustomed to, including email, SMS, U2F, and security questions, as discussed in Section 7.4. Specifically, N-for-1-Auth requires no changes to the protocols of these forms of authentication.

N-for-1-Auth offers the same security properties as the underlying authentication protocols even in the presence of a malicious adversary that can compromise up to $N-1$ of the N servers. N-for-1-Auth additionally offers relevant privacy properties for the users. Users of authentication factors may want to hide their email address, phone number, and security questions from the authentication servers. This is difficult to achieve in traditional (centralized) authentication. We discuss the concrete privacy properties of N-for-1-Auth for each factor in Section 7.4.

N-for-1-Auth provides an efficient implementation for several factors and is $8\times$ faster than a naive implementation without our application-specific optimizations. For example, when $N=5$, our email authentication protocol takes 1.38 seconds to perform the TLS handshake and takes an additional 0.43 seconds to send the email payload, which is efficient enough to avoid a TLS timeout and successfully communicate with an unmodified TLS email server.

7.1.2 The case for N-for-1-Auth

We presented N-for-1-Auth to top executives of several prominent companies (which we are not ready to disclose at this moment) in the end-to-end encryption or cryptocurrency custody space.

We received valuable feedback from them, which we used to improve N-for-1-Auth.

- Many companies mentioned the need for fault tolerance, which can be addressed in two steps. First, N-for-1-Auth can incorporate threshold secret-sharing, as discussed in Section 7.8, which enables a subset of servers to recover the secret. Second, each server can set up backup machines within their trust domain/cloud.

- Some companies mentioned the need for more authentication factors, e.g., TOTP (time-based one-time passcodes) and SSO (single sign-on) [557, 558], which can be integrated into N-for-1-Auth in similar ways—TOTP follows a similar format to security questions in which the client stores the TOTP key, and SSO can be integrated using N-for-1-Auth's TLS-in-SMPC protocol.

- Some companies mentioned the need to hide user contact information from the other servers, which we address in Section 7.4.

Overall, we hope that N-for-1-Auth can provide practical value to this space of distributed trust.

7.1.3 Summary of techniques

We now describe how N-for-1-Auth maintains the same user experience while decentralizing trust.

One passcode, N servers. Consider email authentication. How do N servers coordinate to send one email with an authentication passcode that they agree on?

First, no server should know the passcode, otherwise this server can impersonate the user. We want to ensure that N-for-1-Auth provides the same security as the traditional solution in which N servers each send a different email passcode to the user.

N-for-1-Auth's solution is to have the N servers jointly generate a random passcode for email authentication in secure multiparty computation (SMPC) [81, 82, 207, 208]. In this way, none of them learn the passcode. However, an immediate question arises: how do they send this jointly generated passcode to the user's email address securely?

In the traditional workflow for sending email, one party connects to the user's email service provider (e.g., Gmail) via TLS. The TLS server endpoint is at the email service provider, and the TLS client endpoint is at the sender's email gateway. The mismatch in our setting is that the sender now comprises N servers who must not see the contents of the email.

Sending TLS-encrypted traffic from SMPC. Our insight is that using a new primitive—TLS-in-SMPC—with which the N servers can jointly act as a single TLS client endpoint to communicate with the user's email server over a TLS connection, as Figure 7.3 shows. When connecting with the user's email server, the N servers run a maliciously secure SMPC that takes the place of a traditional TLS client. What comes out of the SMPC is TLS-encrypted traffic, which one of the servers simply forwards to the user's email provider. N-for-1-Auth's TLS-in-SMPC protocol stores TLS secrets inside SMPC such that none of the servers can break the security guarantees of TLS. Therefore, the server that forwards the traffic can be arbitrary and does not affect security.

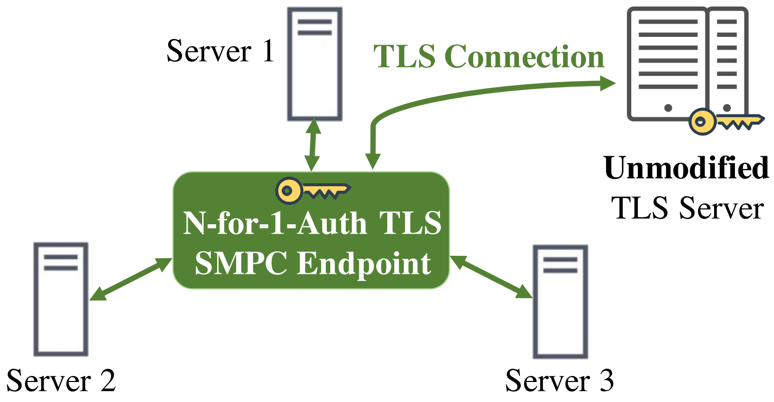


Figure 7.3: TLS-in-SMPC's system architecture.

The user's email server, which is unmodified and runs an unmodified version of the TLS server protocol, then decrypts the traffic produced by the TLS-in-SMPC protocol and receives the email. The email is then seen by the user, who can enter the passcode into a client app to authenticate to the N servers, thereby completing N-for-1-Auth's email authentication.

Support for different authentication factors. Beyond email, N-for-1-Auth supports SMS, U2F, and security questions. In addition, due to the generality of N-for-1-Auth's TLS-in-SMPC construct, N-for-1-Auth can also support any web-based authentication such as OAuth [558], which we discuss how to incorporate in Section 7.7. However, we focus on the aforementioned factors in this work. Each factor has its unique challenges for N-for-1-Auth, particularly in ensuring N-for-1-Auth does not reduce the security of these factors. More specifically, replay attacks are a common threat to authentication protocols. In our system, when a malicious server receives a response from the user, this server may attempt to use the response to impersonate the user and authenticate with the other servers. We systematically discuss how to defend against such replay attacks in Section 7.4.

End-to-end implementation for TLS-in-SMPC. TLS is an intricate protocol that involves many cryptographic operations. If we run the TLS endpoint using a maliciously secure SMPC library off the shelf, our experiments in Section 7.6.5 show that it would be at least $8\times$ more expensive than our protocol. We designed our TLS-in-SMPC protocol and optimized its efficiency with a number of insights based on the TLS protocol itself, and integrated it with the `wolfSSL` library.

7.2 System overview

In this section we describe the system at a high level.

System setup. An *N-for-1* authentication system consists of many *servers* and *users*. Each user has a number of *authentication factors* they can use to authenticate. N-for-1-Auth recommends users to use multiple second factors when authenticating to avoid having any single factor act as a central point of attack. The user holds a secret that they wish to distribute among the N servers. Based on our discussion with companies in Section 7.1.2, these companies wished that one of the N servers be the actual server of the application, and the other $N - 1$ servers be other helper servers. This is natural because the app server is providing the service, and importantly, the app server is reassured that if it behaves well, the $N - 1$ helper servers cannot affect security. Each user can download a *stateless* client application or use a web client to participate in these protocols. This minimalist client app *does not retain secrets* or demand intrusive permissions to data in other applications such as a user's emails or text messages; it simply serves as an interface between the user and the servers. We place such limitations on the client app since we assume the device hosting the app can be lost or stolen, and we want to hide the user's sensitive data from our client app.

Workflow. The system consists of two phases:

- *Enrollment (Figure 7.1).* When the user wants to store a secret on the servers, the user provides the servers with a number of authentication factors, which the servers verify using N-for-1-Auth's authentication protocols described in Section 7.4. Then, after authenticating with these factors, the client secret-shares the secret and distributes the shares across the servers.
- *Authentication (Figure 7.2).* The user runs the N-for-1-Auth protocols for the authentication factors. Once the user is authenticated, the N servers can perform computation over the secret for the user, which is application-specific, as we describe in Section 7.5.

Though in use cases such as key recovery, the authentication phase only occurs in catastrophic situations, users must enroll their factors when joining the system, which typically requires N times the effort and is a different user experience from enrolling to a centralized system.

N-for-1 Authentications. We describe N-for-1-Auth's authentication protocols for several factors in Section 7.4.

- *Email:* The N servers jointly send *one* email to the user's email address with a passcode. During authentication, the servers expect the user to enter this passcode.
- *SMS:* The N servers jointly send *one* message to the user's phone number with a passcode. During authentication, the servers expect the user to enter this passcode.
- *U2F:* The N servers jointly initiate *one* request to a U2F device. During authentication, the servers expect a signature, signed by the U2F device, over this request.
- *Security questions:* The user initially provides a series of questions and answers to the servers. During authentication, the servers ask the user these questions and expect answers consistent with those that are set initially. Passwords are a special case of security questions and can also be verified using this protocol.

Applications. We describe how N-for-1-Auth supports two common applications in Section 7.5 , but N-for-1-Auth can also be used in other systems with decentralized trust.

- *Key recovery:* The user can backup critical secrets by secret-sharing them among the N servers. Upon successful authentication, the user can then retrieve these secrets from the servers.

– *Digital signatures:* The user can backup a signing key (e.g., secret key in Bitcoin) by secret-sharing it among the N servers. Upon successful authentication, the servers can sign a signature over a message the user provides, such as a Bitcoin transaction.

Example. We illustrate how to use N-for-1-Auth with a simple example. Alice enrolls into N-for-1-Auth through the client app. She provides three authentication factors: her email address, her phone number, and her U2F token. The client app then contacts the N servers and enrolls these factors. The N servers then send *one* email and *one* text message, both containing a random passcode, and *one* request to Alice's U2F device. Alice then enters the passcodes on the client app and confirms on her U2F device. When all the N servers have verified Alice, the client app then secret-shares the key with the servers, and the servers store the shares. Alice performs the same authentication when she wants to recover the secrets.

7.2.1 Threat model

N-for-1-Auth's threat model, illustrated in Figure 7.4, is as follows. Up to $N-1$ of the N servers can be ***malicious*** and collude with some users, but at least one server is ***honest*** and does not collude with any other parties. The honest users do not know which server is honest. The malicious servers may deviate from the protocol in arbitrary ways, including impersonating the honest user, as Figure 7.4 shows. For ease of presentation, we assume that servers do not perform denial-of-service (DoS) attacks, but we discuss how to handle these attacks in Section 7.8.

Users can also be ***malicious*** and collude with malicious servers. Malicious users may, for example, try to authenticate as an honest user. We assume that an honest user uses an uncompromised client app, but a malicious user may use a modified one. The client app does not carry any secrets, but it must be obtained from a trusted source, as in the case of the software clients in E2EE or cryptocurrency systems. The client app either has hardcoded the TLS certificates of the N servers, or obtains them from a trusted certificate authority or a transparency ledger [559, 560]. This enables clients and servers to connect to one another securely using the TLS protocol.

Security properties. N-for-1-Auth is built on top of existing authentication factors and maintains the same security properties that the existing factors provide under this threat model. This assertion rests on the security of N-for-1-Auth's TLS-in-SMPC protocol. Formally, we define in Section 7.9 an ideal functionality $\mathcal{F}_{\mathsf{TLS}}$ that models the TLS client software that communicates with a trusted, unmodified TLS server. Based on $\mathcal{F}_{\mathsf{TLS}}$, we define the security of our TLS-in-SMPC protocol using a standard definition for (standalone) malicious security [200]:

Definition 36 (Security of TLS-in-SMPC)**.** A protocol Π is said to *securely compute* $\mathcal{F}_{\mathsf{TLS}}$ in the presence of static malicious adversaries that compromise up to $N-1$ of the N servers, if, for every non-uniform probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ in the real world, there exists a non-uniform PPT adversary $\mathcal{S}$ in the ideal world, such that for every $I \subseteq \{1, 2, ..., N\}$,

$$\{\text{IDEAL}_{\mathcal{F}_{\mathsf{TLS}},I,\mathcal{S}(z)}(\vec{x})\}_{\vec{x},z} \overset{c}{\approx} \{\text{REAL}_{\Pi,I,\mathcal{A}(z)}(\vec{x})\}_{\vec{x},z}$$

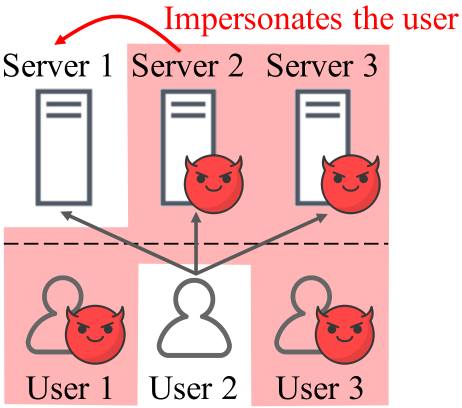


Figure 7.4: N-for-1-Auth's threat model. The **red** area indicates a group of malicious parties who collude with one another.

where $\vec{x}$ denotes all parties' input, z denotes an auxiliary input for the adversary $\mathcal{A}$, $\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{TLS}},I,\mathcal{S}(z)}(\vec{x})$ denotes the joint output of $\mathcal{S}$ and the honest parties, and $\mathrm{REAL}_{\Pi,I,\mathcal{A}}(\vec{x})$ denotes the joint output of $\mathcal{A}$ and the honest parties.

We present our TLS-in-SMPC protocol in Section 7.3, and we prove that it securely realizes $\mathcal{F}_{\mathsf{TLS}}$ in Section 7.9.

7.3 TLS in SMPC

In N-for-1-Auth's email/SMS authentication protocols, the N servers need to establish a secure TLS connection with an *unmodified* TLS server. In this section, we describe TLS-in-SMPC, a protocol that achieves this goal.

Background: secure multiparty computation. The goal of secure multiparty computation (SMPC) [81, 82, 207, 208] is to enable N parties to collaboratively compute a function $f(x_1, x_2, ..., x_N)$, in which the i-th party has private input x_i, without revealing x_i to the other parties.

SMPC protocols are implemented using either arithmetic circuits such as in SPDZ [**spdz2**, 206] or boolean circuits such as in AG-MPC [258, 437]. These protocols consist of an offline phase and an online phase. The offline phase is independent of the function's input and can be run beforehand to reduce the online phase latency.

7.3.1 Overview

In TLS-in-SMPC, N servers jointly participate in a TLS connection with an unmodified TLS server. Since these N servers do not trust each other, any one of them must not be able to decrypt the traffic sent over the TLS connection. Therefore, the insight is for these N servers to jointly create a TLS client endpoint within SMPC that can communicate with the TLS server over a TLS connection.

As Figure 7.3 shows, the N servers run a TLS client within SMPC, which establishes a TLS connection with the unmodified TLS server. The TLS session keys are only known by the TLS server and the TLS client within SMPC. Hence, the N servers must work together to participate in this TLS connection.

All packets are forwarded between the SMPC and the unmodified TLS server via one of the servers. The specific server that forwards the packets does not affect security since none of the servers know the TLS session keys. That is, none of the servers can decrypt the packets being forwarded or inject valid packets into the TLS connection. The TLS-in-SMPC protocol consists of two phases:

- **TLS connection establishment:** The N servers jointly generate the client-side secret for Diffie-Hellman key exchange. After receiving the server-side secret, they derive the TLS session keys inside SMPC.

- **Data exchange:** The N servers, within SMPC, use the session keys to encrypt or decrypt a message.

Challenge. A straightforward implementation of the TLS-in-SMPC protocol is to use any malicious SMPC protocol off the shelf. If this protocol does not support offline precomputation or is ill-suited for the type of computation being performed, the online latency may cause a timeout that terminates the connection. For example, we found that Gmail's SMTP servers have a TLS handshake timeout of 10 s. Our implementation is efficient enough to consistently meet this timeout, as discussed in Section 7.6.

7.3.2 TLS connection establishment

We discuss how N-for-1-Auth's TLS-in-SMPC protocol handles key exchange and how it differs from traditional Diffie-Hellman key exchange. We do not discuss RSA key exchange as it is not supported in TLS 1.3.

Background: Diffie-Hellman key exchange [561]. Let G be the generator of a suitable elliptic curve of prime order p. The key exchange consists of three steps:

1. In the `ClientHello` message, the TLS client samples $\alpha \leftarrow\!\$\, \mathbb{Z}_p^+$ and sends $\alpha \cdot G$ to the TLS server.

2. In the `ServerHello` message, the TLS server samples $\beta \leftarrow\!\$\, \mathbb{Z}_p^+$ and sends $\beta \cdot G$ to the TLS client.

3. The TLS client and server compute $\alpha\beta \cdot G$ and—with other information—derive the TLS session keys, as specified in the TLS standards [562, 563].

Step 1: Distributed generation of client randomness $\boldsymbol{\alpha \cdot G}$. To generate the client randomness $\alpha \cdot G$ used in the `ClientHello` message without revealing α, each server samples a share of α and provides a corresponding share of $\alpha \cdot G$, as follows:

1. For all servers, the i-th server $\mathcal{P}_i$ samples $\alpha_i \leftarrow\$ \mathbb{Z}_p^+$ and broadcasts $\alpha_i \cdot G$, by first committing to $\alpha_i \cdot G$ and then revealing it.
2. $\mathcal{P}_1$ computes and sends $\sum_{i=1}^{N} \alpha_i \cdot G$ to the TLS server.

This step can be done before the connection starts.

Step 2: Distributed computation of key exchange result $\boldsymbol{\alpha\beta \cdot G}$. The N servers need to jointly compute $\alpha\beta \cdot G$, which works as follows: each server computes $\alpha_i(\beta G)$ first, and then the SMPC protocol takes $\alpha_i(\beta G)$ as input from server $\mathcal{P}_i$ and computes $\alpha\beta \cdot G = \sum_{i=1}^{n} \alpha_i(\beta G)$. The result is used to derive the TLS session keys, which we discuss next.

Step 3: Distributed key derivation. The next step is to compute the TLS session keys inside SMPC using a key derivation function [564]. The N servers, within SMPC, derive the handshake secrets from $\alpha\beta \cdot G$ and the hash of the `ClientHello` and `ServerHello` messages, and then derive the handshake keys and IVs within SMPC.

We identify that the hashes of the communication transcript messages, which is needed for key derivation, can be computed outside SMPC, which reduces the overhead. That is, the forwarding server broadcasts these TLS messages to the other servers. Each server computes the hashes, and all servers input these hashes to the SMPC instance. This approach is secure because TLS already prevents against man-in-the-middle attacks, which means that these messages are not sensitive.

Step 4: Verifying the TLS connection. The TLS server sends a response containing its certificate, a signature over $\beta \cdot G$, and verification data, which the TLS client verifies and replies. Performing this verification in SMPC is slow because (1) the certificate format is difficult to parse without revealing access patterns and (2) verifying signatures involves hashing and prime field computation, both of which are slow in SMPC.

In N-for-1-Auth, we are able to remove this task from SMPC. The insight is that the handshake keys, which encrypts the response, are only designed to hide the TLS endpoints' identity, which is unnecessary because in our setting, the servers must confirm the TLS server's identity. Several works show that revealing the keys does not affect other guarantees of TLS [565–568]. We formalize this insight in our definition of the ideal functionality $\mathcal{F}_{\mathsf{TLS}}$, as described in Section 7.9.2.

Therefore, verifying the TLS server's response is as follows: after all the N servers receive and acknowledge all the messages from `ServerHello` to `ServerFinished` sent by the TLS server and forwarded by the first server, the SMPC protocol reveals the TLS handshake keys to all the N servers. Each server decrypts the response and verifies the certificate, signature, and verification data within it. Then, the N servers can use the handshake key, and then within SMPC assemble the client handshake verification data, which is then sent to the TLS server. Lastly, the N servers derive the application keys, which are used for data exchange, from the handshake secrets and the hash of the transcript inside SMPC.

Step 5: Precomputation for authenticated encryption. The authenticated encryption scheme used in data exchange may allow some one-time precomputation that can be done as part of the TLS connection establishment. For example, for AES-GCM N-for-1-Auth can precompute the AES key schedule and secret-share the GCM power series. We provide more details in Section 7.3.3.

Efficient implementation. The key exchange protocol consists of point additions and key derivations. We observe that point additions can be efficiently expressed as an arithmetic circuit whose native field is exactly the point's coordinate field, and key derivations can be efficiently expressed as a boolean circuit. Our insight to achieve efficiency here is to mix SMPC protocols by first implementing point additions with SPDZ using MASCOT [269] for the offline phase, and then transferring the result to AG-MPC [258, 437] for key derivation via a maliciously secure mixing protocol [569–571]. Both SPDZ and AG-MPC support offline precomputation, which helps reduce the online latency and meet the TLS handshake timeout.

We chose MASCOT instead of other preprocessing protocols [**spdz2**, 206, 270] based on homomorphic encryption because many curves used in TLS have a coordinate field with low "2-arity", which is incompatible with the packing mechanisms in homomorphic encryption schemes.

7.3.3 Data exchange

The rest of the TLS-in-SMPC protocol involves data encryption and decryption. An opportunity to reduce the latency is to choose the TLS ciphersuites carefully, as shown by both our investigation and prior work [566, 572].

During key exchange, typically the TLS server offers several TLS ciphersuites that it supports, and the TLS client selects one of them to use. In order to minimize latency, when given the choice, our protocol always selects the most SMPC-friendly ciphersuite that is also secure.

Cost of different ciphersuites in SMPC. The cost of TLS ciphersuites in SMPC has rarely been studied. Here, we implement the boolean circuits of two commonly used ciphersuites, AES-GCM-128 and Chacha20-Poly1305—which are part of the TLS 1.3 standard and supported in many TLS 1.2 implementations—and measure their cost.

After common optimizations, the main overhead rests on the amortized cost of (1) AES without key schedule and (2) Chacha20 in terms of the number of AND gates in boolean circuits. The amortized cost per 128 bits for AES is 5120 AND gates while Chacha20 takes 96256 AND gates due to integer additions. Thus, it is preferred to choose AES-GCM-128 when available.

Efficient GCM tag computation. We adapt from DECO [566] a protocol to efficiently compute the GCM tag to N parties. After deriving the TLS application key within SMPC, the servers compute the GCM generator $H = E_K(0)$ and the power series of H: $H, H^2, H^3, ..., H^L$ within SMPC. The power series is secret-shared among the N servers. To compute the GCM tag for some data $S_1, S_2, ..., S_L$ (authenticated data or ciphertexts), each server computes a share of the polynomial $\sum_{i=1}^{L} S_i \cdot H^i$ and combines these shares with the encryption of initialization vector (IV) within SMPC.

We optimize the choice of L, which has not been done in DECO. For efficiency, L needs to be chosen carefully. A small L will increase the number of encryption operations, and a large L will increase the cost of computing the GCM power series. Formally, to encrypt message of N bytes with AES (the block size is 16 bytes), we find L that minimizes the overall encryption cost:

$$L_{\text{opt}} = \text{argmin}_L \left[\begin{array}{l} (L-1)\cdot 16384 + 1280 + 5120 \\ +M \cdot 5120 + \lceil \frac{N+M}{16} \rceil \cdot 5120 \end{array} \right].$$

where $M = \lceil \frac{N}{16\cdot(L-2)-1} \rceil$ is the number of data packets in the TLS layer.[1] For example, for $N = 512$, choosing $L = N/16 = 32$ is 2.3× the cost compared with $L_{\text{opt}} = 5$.

Circuit implementation. We synthesize the circuit files in TLS-in-SMPC using Synopsys's Design Compiler and tools in TinyGarble [260], SCALE-MAMBA [217], and ZKCSP [432]. The circuits have been open-sourced here.

https://github.com/n-for-1-auth/circuits

7.4 N-for-1-Auth authentication

In this section we describe how a user, using the client app, authenticates to N servers via various authentication factors. We also describe the enrollment phase needed to set up each protocol. After passing the authentication, the user can invoke the applications described in Section 7.5.

General workflow. In general, N-for-1-Auth's authentication protocols consist of two stages:

- The servers jointly send ***one*** challenge to the client.

- The client replies with a response to each server, which will be different for each server.

Depending on the application, users may want to change their authentication factors, in which they would need to authenticate with the servers beforehand.

Preventing replay attacks. The client needs to provide each server a *different* response to defend against replay attacks. If the user sends *the same response* to different servers, a malicious server who receives the response can attempt to beat the user to the honest servers. The honest servers will expect the same message that the malicious server sends, and if the malicious server's request reaches the honest servers first, the honest servers will consider the malicious server authenticated instead of the honest user. Since up to $N-1$ of the servers can collude with one another, in this scenario, the malicious server can reconstruct the shares and obtain the secret.

To prevent replay attacks, we designed the authentication protocols in a way such that no efficient attacker, knowing $N-1$ out of the N responses from an honest user, can output the remaining response correctly with a non-negligible probability.

[1] Besides the actual payload data, the GCM hash also adds on two additional blocks (record header and the data size) and one byte (TLS record content type), which explains the term $16 \cdot (L-2) - 1$.

7.4.1 N-for-1-Auth Email

N-for-1-Auth's email authentication protocol sends the user only one email which contains a passcode. If the user proves knowledge of this passcode in some way, the N servers will consider the user authenticated. N-for-1-Auth's email authentication protocol is as follows:

1. The i-th server $\mathcal{P}_i$ generates a random number s_i and provides it as input to SMPC.

2. Inside SMPC, the servers computes $s = \bigoplus_i^N s_i$, where $\oplus$ is bitwise XOR, and outputs PRF(s, i) to $\mathcal{P}_i$, where PRF is a pseudorandom function.

3. The N servers run the TLS-in-SMPC protocol to create a TLS endpoint acting as an email gateway for some domain. The TLS endpoint opens a TLS connection with the user's SMTP server such as `gmail-smtp-in.l.google.com` for `abc@gmail.com`, and sends an email to the user with the passcode s over this TLS connection. Note that the protocol sends the email using the intergateway SMTP protocol, rather than the one used by a user to send an email.

4. The user receives the email and enters s into the client app, which computes PRF(s, i) and sends the result to $\mathcal{P}_i$. If the user response matches the output that the server received in step 2, then $\mathcal{P}_i$ considers the user authenticated.

Enrollment. The enrollment protocol is as follows:

1. The client opens a TLS connection with each of the N servers and secret-shares the user's email address and sends the i-th share to server $\mathcal{P}_i$.
2. The N servers reconstruct the user's email address within SMPC and then jointly send a confirmation email to the user, with a passcode.
3. The client proves knowledge of the passcode using N-for-1-Auth's email authentication protocol, converts the secret into N secret shares, and sends the i-th share to server $\mathcal{P}_i$.
4. If the user is authenticated, each server stores a share of the user's email address and a share of the secret.

Avoiding misclassification as spam. A common issue is that this email might be misclassified as spam, which can be handled using standard practices as follows.

- *Sender Policy Framework (SPF).* N-for-1-Auth can follow the SPF standard [573], in which the sender domain, registered during the setup of N-for-1-Auth, has a `TXT` record indicating the IP addresses of email gateways eligible to send emails from this sender domain.

- *Domain Keys Identified Mail (DKIM).* The DKIM standard [574] requires each email to have a signature from the sender domain under a public key listed in a `TXT` record. N-for-1-Auth can have the server generate the keys and sign the email, both in a distributed manner.

Our experiments show that supporting SPF is sufficient to avoid Gmail labeling N-for-1-Auth's email as spam.

Privacy. We continue with the example of Alice from the introduction, in which she authenticates to N servers, one of which is her E2EE App Server and the other $N - 1$ are helper servers. In this setting, the E2EE App Server delivers packets to the TLS server. Based on our interaction with companies, the E2EE App Server does not want to reveal Alice's contact information to the other helper servers. To achieve this, for N-for-1-Auth's email authentication protocol, Alice's email address is secret-shared among these N servers, and the E2EE App Server maintains a mapping from Alice's email address to the index at which the shares are stored. During authentication, the E2EE App Server provides this index to the other helper servers. In addition, the helper servers only need to know the email provider's mail gateway address instead of the full email address. This is because the N servers verify the TLS server certificate outside of SMPC for efficiency reasons and therefore the email provider's mail gateway address is revealed.[2] For example, many companies use Google or Microsoft for email service on their domains. In this case, for a user with email address `A@B.com`, the helper servers know neither `A` nor `B.com`, but only which email service provider is used by `B.com`. This property is useful in the case of data breaches as compromising the helper servers does not reveal Alice's email. We note that for the E2EE App Server to maintain the mapping, they only need to have the full information of only *one* of the user's second-factors, such as their email or phone number, whereas the information for their other registered factors can be secret-shared and hidden from the E2EE App Server as well.

7.4.2 N-for-1-Auth SMS

N-for-1-Auth's SMS protocol sends the user *one* text message, which contains a passcode. The enrollment and authentication protocols resemble the email authentication protocol except that the passcode is sent via SMS.

We leverage the fact that many mobile carriers, including AT&T [575], Sprint [576], and Verizon [577], provide commercial REST APIs to send text messages. The N servers, who secret-share the API key, can use N-for-1-Auth's TLS-in-SMPC protocol to send a text message to the user through the relevant API.

Privacy. Similar to email, N-for-1-Auth secret-shares the user's phone number among the N servers, allowing the user's phone number to be hidden from the helper servers. Here, only the sender's carrier and the user's carrier sees the user's phone number, but the helper servers cannot. If the N servers have the SMS API to the user's carrier, they can use that API, so that only one mobile carrier sees the message.

7.4.3 N-for-1-Auth U2F

Universal second factor (U2F) [578] is an emerging authentication standard in which the user uses U2F devices to produce signatures to prove the user's identity. Devices that support U2F include YubiKey [579] and Google Titan [580]. The goal of N-for-1-Auth's U2F protocol is to have the user operate on the U2F device *once*.

[2] If the certification verification is done in SMPC, the gateway address can be hidden, but with a high overhead, as discussed in Section 7.3.2.

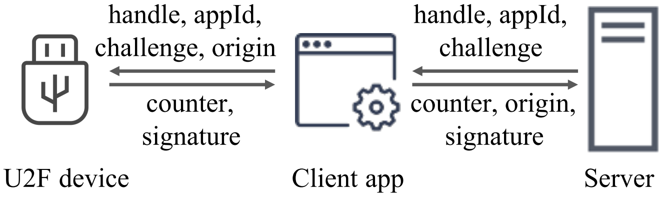


Figure 7.5: Protocol of universal second factor (U2F).

Background: U2F. A U2F device attests to a user's identity by generating a signature on a challenge requested by a server under a public key that the server knows. The U2F protocol consists of an enrollment phase and an authentication phase, described as follows.

In the enrollment phase, the U2F device generates an application-specific keypair and sends a key handle and the public key to the server. The server stores the key handle and the public key.

In the authentication phase, as Figure 7.5 shows, the server generates a random challenge and sends over the key handle, the application identifier (appId), and a challenge to a U2F interface such as a client app, which is then, along with the origin name of the server, forwarded to the U2F device. Then, upon the user's confirmation, such as tapping a button on the device [579, 580], the U2F device generates a signature over the request. The signature also includes a monotonic counter to discover cloning attacks. The server receives the signature and verifies it using the public key stored in the enrollment phase.

To avoid the user having to perform a U2F authentication for each server, an intuitive approach is to have the servers generate a joint challenge which is then signed by the U2F device. The client can secret-share the signature, and the servers can then reconstruct and verify the signature within SMPC. However, signature verification in SMPC can be prohibitively expensive.

An insecure strawman. We now describe an insecure strawman, which will be our starting point in designing the secure protocol. Let the N servers jointly generate a random challenge. The strawman lets the client obtain a signature over this challenge from U2F and sends the signature to each server. Then, each server verifies the signature, and the servers consider the user authenticated if the verification passes for each server.

This approach suffers from the replay attack described in Section 7.4. When a malicious server receives the signature from the client, this server can impersonate the honest user by sending this signature to the other servers.

N-for-1-Auth U2F's protocol. Assuming server $\mathcal{P}_i$ chooses a random challenge value s_i, our protocol must satisfy two requirements: (1) the challenge signed by the U2F device is generated using all the servers' randomness $s_1, s_2, ..., s_N$; and (2) the client can prove to server $\mathcal{P}_i$ that the signed challenge uses s_i without revealing information about other parties' randomness.

We identify that aggregating the servers' randomness via a Merkle tree combined with cryptographic commitments, as Figure 7.6 shows, satisfies these requirements. We now briefly describe these two building blocks.

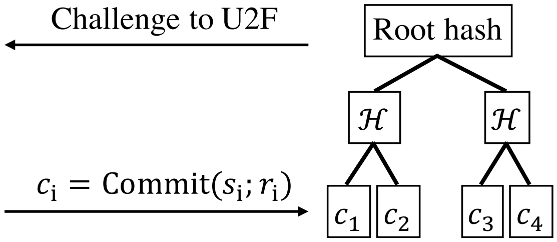


Figure 7.6: The Merkle tree for U2F challenge generation.

In a Merkle tree, if the client places the servers' randomness $s_1, s_2, ..., s_N$ into the leaf nodes, as Figure 7.6 shows, then the root hash value is a collision-resistant representation of all the servers' randomness, which we will use as the challenge for the U2F device to sign over.

However, Merkle trees are not guaranteed to hide the leaf nodes' values. To satisfy the second requirement, as Figure 7.6 shows, we use cryptographic commitments $c_i = \texttt{Commit}(s_i;\ r_i)$ instead of s_i as the leaf nodes' values, in which r_i is a random string chosen by the client. The commitments provide two guarantees: (1) the server, from the commitment c_i, does not learn s_i and (2) the client cannot open c_i to a different $s_i' \neq s_i$.

Next, the client obtains the signature of the root hash from U2F and sends each server the following response: (1) the signature, (2) a Merkle tree lookup proof that the i-th leaf node has value c_i, and (3) commitment opening secrets r_i and s_i. Here, only the client and the i-th server know the server randomness s_i.

The detailed authentication protocol is as follows:

1. Each server opens a TLS connection with the client and sends over a random value s_i.

2. The client builds a Merkle tree as described above and in Figure 7.6 and obtains the root hash. The client requests the U2F device to sign the root hash as the challenge, as Figure 7.5 shows, following the U2F protocol.

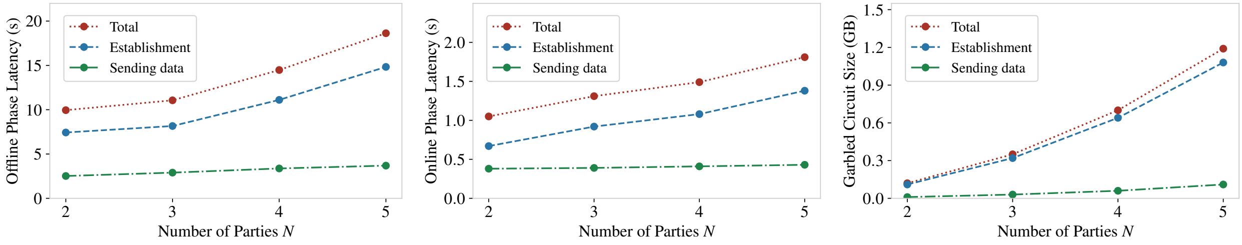
3. The user then operates on the U2F device *once*, which produces a signature over the root hash. The client app then sends the signature, the Merkle tree lookup proof, and the commitment opening information to each server.

4. Each server verifies the signature, opens the commitment, verifies that the commitment is indeed over the initial value s_i provided by server $\mathcal{P}_i$, and checks the Merkle tree lookup proof. If everything is verified, then $\mathcal{P}_i$ considers the user authenticated.

This protocol prevents replay attacks as described above since the client's response to $\mathcal{P}_i$ contains the opening secret s_i; other servers cannot determine this value with a non-negligible probability.

Enrollment. The enrollment protocol is as follows:

1. The client and the servers engage in the standard U2F enrollment protocol [578], in which the servers obtain the key handle and the public key.



(a) Offline phase latency. (b) Online phase latency. (c) Garbled circuit size.

Figure 7.7: The overall online/offline phase latencies and the garbled circuit size of the TLS-in-SMPC protocol for $N = 2, 3, 4, 5$ servers when sending an email with passcode (the mail body is 34 bytes).

2. The client and the servers run N-for-1-Auth's U2F authentication protocol as described above.

Privacy. The U2F protocol already provides measures to hide a device's identity [581], which N-for-1-Auth leverages to provide privacy for the user.

7.4.4 N-for-1-Auth security questions

The last N-for-1-Auth authentication factor we present is security questions. Although many of the properties provided by N-for-1-Auth's security questions protocol have been achieved by prior works [548–551, 556], we briefly describe this protocol for completeness.

Typically, security questions involve the user answering a series of questions that (ideally) only the user knows all of the answers to [582–585]. During enrollment, the user provides several questions and their answers to the client app. The client then hashes the answers, and then sends secret-shares of the hashes to the N servers. Then, during authentication, the user is asked to answer these questions. The client, similar to before, hashes the provided answers and provides secret-shares to the N servers who then, within SMPC, reconstruct these hashes and compare with the hashes originally stored from enrollment. If all the hashes match, the user is considered authenticated.

Privacy and other benefits over traditional security questions. Traditionally, security questions avoid asking users for critical personal secrets, such as their SSN, because the user may feel uncomfortable sharing such personal information. Hashing and other cryptographic techniques do not help much since the answer is often in a small domain and can be found via in an offline brute-force attack. However in N-for-1-Auth, since the hashes of the answers are secret-shared among the N servers, no server knows the full hash and therefore offline brute force attacks are impossible. The privacy of the user's answers (or their hash) from the servers can encourage users to choose more sensitive questions and enter more sensitive answers that the user would otherwise be uncomfortable sharing. To hide these more sensitive questions, we can leverage an existing industry practice, *factor sequencing* [586], by showing these more sensitive questions only after

Component	**Offline Phase Latency (s)**				**Online Phase Latency (s)**			
	$N = 2$	$N = 3$	$N = 4$	$N = 5$	$N = 2$	$N = 3$	$N = 4$	$N = 5$
TLS connection establishment	7.43	8.16	11.11	14.83	0.67	0.92	1.08	1.38
◇ Client randomness generation	0.30	0.30	0.30	0.30	—	—	—	—
◇ Key exchange result computation	0.02	0.06	0.09	0.15	0.25	0.35	0.37	0.47
◇ Key derivation	6.55	7.05	9.73	13.1	0.37	0.51	0.64	0.83
◇ GCM power series ($L = 5$)	0.49	0.65	0.87	1.15	0.03	0.04	0.05	0.06
◇ AES key schedule	0.07	0.10	0.12	0.13	0.02	0.02	0.02	0.02
Sending an email of 34 bytes in TLS	2.52	2.90	3.37	3.69	0.38	0.39	0.41	0.43
Sending a SMTP heartbeat in TLS	0.43	0.49	0.57	0.63	0.06	0.07	0.07	0.07

Table 7.1: Breakdown of the TLS-in-SMPC latencies for sending an email with passcode (the mail body is 34 bytes).

the user correctly answers less sensitive questions. Furthermore, to mitigate online brute force attacks, in addition to standard rate-limiting mechanisms which N-for-1-Auth supports, they can set other authentication factors as *prerequisites*. That is, only when the user authenticates against prerequisite factors can the user even see the security questions.

7.5 Applications

Once the N servers have authenticated the user, they can perform some operations for the user using their secret that is secret-shared during enrollment, such as key recovery as in our motivating example. To show the usefulness of N-for-1-Auth, we now describe two applications that can benefit from N-for-1-Auth.

Key recovery. The user can backup a key by secret-sharing it as the user secret during the enrollment phase. When the user needs to recover the key, the servers can send the shares back to the user, who can then reconstruct the key from the shares. Key recovery is widely used in end-to-end encrypted messaging apps, end-to-end encrypted file sharing apps, and cryptocurrencies.

Digital signatures. Sometimes, it is preferred to obtain a signature under a secret key, rather than retrieving the key and performing a signing operation with it. This has wide applications in cryptocurrencies, in which the user may not want to reconstruct the key and have it in the clear. Instead, the user delegates the key to several servers, who sign a transaction only when the user is authenticated. The user can also place certain restrictions on transactions, such as the maximum amount of payment per day, which can be enforced by the N servers within SMPC. In N-for-1-Auth, the user secret-shares the signing key in the enrollment phase. Before performing a transaction, the user authenticates with the servers. Once authenticated, the user presents a transaction to the N servers, who then sign it using a multi-party ECDSA protocol [587–591]. An alternative solution is to use multisignatures [592], which N-for-1-Auth can also support, but this option in unavailable in certain cryptocurrencies [535] and may produce very long transactions when N is large.

7.6 Evaluation

In this section we discuss N-for-1-Auth's performance by answering the following questions:

1. Is N-for-1-Auth's TLS-in-SMPC protocol practical? Can it meet the TLS handshake timeout? (Section 7.6.3)
2. How efficient are N-for-1-Auth's authentication protocols? (Section 7.6.4)
3. How does N-for-1-Auth compare with baseline implementations and prior work? (Section 7.6.5 and Section 7.6.6)

7.6.1 Implementation

We use MP-SPDZ [215], emp-toolkit [259, 437] and wolfSSL [593] to implement N-for-1-Auth's TLS-in-SMPC protocol. We implemented the online phase of elliptic-curve point additions within SMPC from scratch in C++.

7.6.2 Setup

We ran our experiments on `c5n.2xlarge` instances on EC2, each equipped with a 3.0 GHz 8-core CPU and 21 GB memory. To model a cross-state setup, we set a 20 ms round-trip time and a bandwidth of 2 Gbit/s between servers (including the TLS server) and 100 Mbit/s between clients and servers.

7.6.3 TLS-in-SMPC's performance

For the TLS-in-SMPC protocol, we measured the offline and online phases' latencies and the size of the garbled circuits sent in the offline phase and show the results in Figure 7.7. From the figure, we see that the offline and online phase latencies and the total circuit size grow roughly linearly to the number of servers.

We consider N from 2 to 5 in this experiment. In practice, companies with decentralized key management such as Curv [543] and Unbound Tech [544] currently use two mutually distrusting parties, and Keyless [547] uses three in their protocol. For all values of N that we tested, the protocol always meets the TLS handshake timeout.

A large portion of the offline cost is in transmitting the garbled circuits used in AG-MPC, as Figure 7.7 shows. N-for-1-Auth's servers run the offline phase before the TLS connection is established to avoid this extra overhead. To load these circuits to the memory efficiently, one can use a memory planner optimized for secure computation [594]. Malicious users can perform DoS attacks by wasting computation done in the offline phase. N-for-1-Auth can defend against such attacks using well-studied techniques, such as proof-of-work or payment [183, 239].

Latency breakdown. In Table 7.1 we show a breakdown of the offline and online phase latencies for the TLS-in-SMPC protocol. From the table, we see that most of the computation is in the offline phase, and the online phase has a small latency. Therefore, if we run an SMPC protocol off

	Offline Phase Latency (s)	**Online Phase Latency (s)**
Email	10.96 (2.90)	1.29 (0.39)
SMS	12.26 (4.10)	1.48 (0.56)
U2F	—	0.03
Security Questions	0.03	0.04

Table 7.2: Latencies of N-for-1-Auth ($N = 3$). Numbers in parentheses are the cost given an established TLS connection.

the shelf that does not precompute the offline phase, from Table 7.1 we see that for $N = 5$, the key exchange has a latency of 14.83 s and cannot meet a TLS handshake timeout of 10 s.

We see from Table 7.1 that the latency for establishing the TLS connection dominates. However, N-for-1-Auth can create a persistent connection with the email receiving gateway server, allowing this to be a one-time cost, which is particularly useful for popular email service providers like Gmail. With an established connection, sending an email with $N = 5$ only takes 3.69 s in the offline phase and 0.43 s in the online phase, which is drastically smaller. To maintain this connection, N-for-1-Auth servers can send SMTP heartbeats (a `NOOP` command). Our experiment with Gmail show that one heartbeat per 120 s is sufficient to maintain a long-term connection for at least 30 minutes.

7.6.4 N-for-1-Auth's authentication performance

We measured the offline and online phase latencies of the N-for-1-Auth protocols and present the results in Table 7.2. We now discuss the results in more detail.

Email/SMS. Using a message of 34 characters, the N-for-1-Auth email protocol (without DKIM signatures) sends 165 bytes via TLS-in-SMPC, and N-for-1-Auth's SMS protocol sends 298 bytes via TLS-in-SMPC, estimated using AT&T's SMS API documentation [575].

U2F. We implement the collision-resistant hash and commitments with SHA256. The computation time for the client and the server is less than 1 ms. The protocol incurs additional communication cost, as the client sends each server a Merkle proof of 412 bytes. We note that all of the overhead comes from the online phase.

Security questions. Checking the hashed answer of one security question can be implemented in AG-MPC, which takes 255 AND gates.

7.6.5 Comparison with off-the-shelf SMPC

We compare N-for-1-Auth's implementation with an off-the-shelf implementation in emp-toolkit [259, 437]. We estimate this cost by implementing the computation of $\alpha\beta \cdot G$ in key exchange, which offers a lower bound on its performance of TLS and is already much slower than

N-for-1-Auth. With $N = 5$ servers, the overall latency is at least $8\times$ slower compared with N-for-1-Auth's TLS-in-SMPC implementation. This is because computing $\alpha\beta \cdot G$ involves expensive prime field operations, which use 10^7 AND gates. With $N = 5$ servers, this step already takes 150 s in the offline phase and 8.6 s in the online phase.

7.6.6 Comparison with DECO

DECO [566] is a work that runs TLS in secure two-party computation. As discussed in Section 7.7, their implementation is not suitable for N-for-1-Auth because it is restricted to two parties and has extra leakage due to a different target setting. During the TLS handshake, DECO uses a customized protocol based on multiplicative-to-additive (MtA) [589] to add elliptic curve points, while N-for-1-Auth uses SPDZ. We are unaware of how to extend DECO's protocol to $N \geq 3$. In addition, when comparing with DECO, N-for-1-Auth's AES implementation reuses the AES key schedule across AES invocations, which reduces the number of AND gates per AES invocation from 6400 to 5120, an improvement of 20%.

7.7 Related work

Decentralized authentication. Decentralized authentication has been studied for many years and is still a hot research topic today. The main goal is to avoid having centralized trust in the authentication system. One idea is to replace centralized trust with trust relationships among different entities [595, 596], which has been used in the PGP protocol in which individuals prove the identities of each other by signing each other's public key [597, 598]. Another idea is to make the authentication system transparent to the users. For example, blockchain-based authentication systems, such as IBM Verify Credentials [599], BlockStack [600], and Civic Wallet [601], and certificate/key transparency systems [481, 496, 559, 560, 602, 603] have been deployed in the real world.

A recent concurrent work [604] also addresses decentralized authentication for cryptocurrency by integrating U2F and security questions with smart contracts. Their construction does not support SMS/email authentication due to limitations of smart contracts, and does not work with cryptocurrency that does not support smart contracts like Bitcoin. In sum, their approach targets a different setting than N-for-1-Auth, as we focus on the usability issues of having the user perform N-times the work.

Password-based secret generation. There has been early work on generating strong secrets from passwords using several mutually distrusting servers, such as [556]. However, [556] focuses on passwords while N-for-1-Auth focuses on second factors, which brings its own set of unique challenges as N-for-1-Auth needs to be compatible with the protocols of these second-factors. In addition, [556] requires the user to remember a secret, which has its own issues as the secret can be lost. We note that these works are complementary to N-for-1-Auth, which can provide secure key backup for these passwords.

Decentralized storage of secrets. In industry, there are many companies that use decentralized trust to store user secrets, such as Curv [543], Partisia [605], Sepior [546], Unbound Tech [544], and Keyless [547]. These companies use SMPC to store, reconstruct, and apply user secrets in a secure decentralized manner. However, in principle a user still needs to authenticate with each of these company's servers since theses servers do not trust each other. Therefore, in these settings a user still needs to do N times the work in order to access their secret. N-for-1-Auth's protocols can assist these commercial decentralized solutions to minimize the work of their users in authentication.

TLS and SMPC. There are works using TLS with secure two-party computation (S2PC), but in a prover-verifier setting in which the prover proves statements about information on the web. BlindCA [572] uses S2PC to inject packets in a TLS connection to allow the prover to prove to the certificate authority that they own a certain email address. However, it issue is that the prover possesses all of the secrets of the TLS connection, and all of their traffic sent to the server must go through a proxy owned by the verifier. DECO [566] uses TLS within S2PC, but its approach also gives the prover the TLS encryption key, which our setting does not allow. Overall, both of these works are restricted to two parties based on their intended settings, while N-for-1-Auth supports an arbitrary number of parties.

In addition, a concurrent work [567] also enables running TLS in secure multiparty computation, and their technical design in this module is similar to ours[3], but [567] does not propose or contribute authentication protocols for distributed trust settings and their applications. N-for-1-Auth offers contributions beyond the TLS-in-SMPC module, proposing the idea of performing N authentications with the work of one, showing how this can be achieved by running inside SMPC the SMTP protocol or the HTTP protocol in addition to TLS, to support authentication factors, and demonstrating applications in the end-to-end encryption and cryptocurrency space. In addition, within the TLS-in-SMPC protocol, we provide an end-to-end implementation compatible with an existing TLS library, wolfSSL, and show that it works for N-for-1-Auth's authentication protocols. Specifically, [567] emulated certain parts of the TLS protocol, and they only evaluated the online phase and did not measure the offline cost, which is important for real-world deployment. In contrast, we also benchmark the offline phase of our protocol.

OAuth. OAuth [558] is a standard protocol used for access delegation, which allows users to grant access to applications without giving out their passwords. While OAuth has several desirable properties, it does not work for all of N-for-1-Auth's second factors, notably SMS text messages and email service providers that do not support OAuth, and is therefore less general and flexible than N-for-1-Auth. In addition, if a user authenticates through OAuth and wants distributed trust, they have to perform the authorization N times, once for each server. N-for-1-Auth can incorporate OAuth as a separate authentication factor—the N servers can secret-share the OAuth client secret and then, using TLS-in-SMPC, obtain the identity information through the OAuth API.

[3] We implemented additional optimizations in AES and GCM.

7.8 Discussion

Handling denial-of-service attacks. In this paper, we consider denial-of-service attacks by the servers to be out of scope, as discussed in Section 7.2.1. There are some defenses against these types of attacks, as follows:

– *Threshold secret sharing.* A malicious server can refuse to provide its share of the secret to prevent the user from recovering it. To handle this, the user can share the secret in a *threshold* manner with a threshold parameter t which will allow the user's secret to be recoverable as long as t servers provide their shares. This approach has a small cost, as a boolean circuit for Shamir secret sharing only takes 10240 AND gates by using characteristic-2 fields for efficient computation.

– *Identifiable abort.* Some new SMPC protocols allow for identifiable abort, in which parties who perform DoS attacks by aborting the SMPC protocol can be identified [370, 606]. N-for-1-Auth can support identifiable abort by incorporating these SMPC protocols and standard identification techniques in its authentication protocols.

7.9 Security proof of TLS-in-SMPC

In this section we provide a security proof for TLS-in-SMPC, following the definition in Section 7.2.1.

7.9.1 Overview

We model the security in the real-ideal paradigm [200], which considers the following two worlds:

- **In the real world,** the N servers run protocol Π, N-for-1-Auth's TLS-in-SMPC protocol, which establishes, inside SMPC, a TLS client endpoint that connects to an unmodified, trusted TLS server. The adversary $\mathcal{A}$ can statically compromise up to $N-1$ out of the N servers and can eavesdrop and modify the messages being transmitted in the network, although some of these messages are encrypted.
- **In the ideal world,** the honest servers, including the TLS server, hand over their information to the ideal functionality $\mathcal{F}_{\mathsf{TLS}}$. The simulator $\mathcal{S}$ obtains the input of the compromised parties in $\vec{x}$ and can communicate with $\mathcal{F}_{\mathsf{TLS}}$. $\mathcal{F}_{\mathsf{TLS}}$ executes the TLS 1.3 protocol, which is assumed to provide a secure communication channel.

We then prove the security in the $\{\mathcal{F}_{\mathsf{SMPC}}, \mathcal{F}_{\mathsf{rpRO}}\}$-hybrid model, in which we abstract the SPDZ protocol and the AG-MPC protocol as one ideal functionality $\mathcal{F}_{\mathsf{SMPC}}$ and abstract the random oracle used in commitments with an ideal functionality for a *restricted programmable random oracle* $\mathcal{F}_{\mathsf{rpRO}}$, which is formalized in [607, 608].

Remark: revealing the server handshake key is safe. In the key exchange protocol described in Section 7.3.2, the protocol reveals the server handshake key and IV to all the N-for-1-Auth servers after they have received and acknowledged the handshake messages. This has benefits for both simplicity and efficiency as TLS-in-SMPC does not need to validate a certificate inside SMPC, which would be expensive.

Informally, revealing the server handshake key is secure because these keys are designed only to hide the server's identity [562], which is a new property of TLS 1.3 that does not exist in TLS 1.2. This property is unnecessary in our setting in which the identity of the unmodified TLS server is known.

Several works have formally studied this problem and show that revealing the keys does not affect other guarantees of TLS [565–568]. Interested readers can refer to these works for more information.

7.9.2 Ideal functionalities

Ideal functionality. In the ideal world, we model the TLS interaction with the unmodified, trusted TLS server as an ideal functionality $\mathcal{F}_{\mathsf{TLS}}$. We adopt the workflow of the standard secure message transmission (SMT) functionality $\mathcal{F}_{\mathrm{SMT}}$ defined in [491].

Given the input $\vec{x}$, $\mathcal{F}_{\mathsf{TLS}}$ runs the TLS client endpoint, which connects to the TLS server, and allows the adversary to be a man-in-the-middle attacker by revealing the messages in the connection to the attacker and allowing the attacker to modify such messages. In more detail,

1. To start, all the N servers must first provide their parts of the TLS client input $\vec{x}$ to $\mathcal{F}_{\mathsf{TLS}}$.

2. For each session id sid, $\mathcal{F}_{\mathsf{TLS}}$ launches the TLS client with input $\vec{x}$ and establishes the connection between the TLS client and the TLS server.

3. The adversary can ask $\mathcal{F}_{\mathsf{TLS}}$ to proceed to the next TLS message by sending a $(\texttt{Proceed}, sid)$ message. Then, $\mathcal{F}_{\mathsf{TLS}}$ generates the next message by continuing the TLS protocol and sends this message to the adversary for examination. The message is in the format of a backdoor message $(\texttt{Sent}, sid, S, R, m)$ where S and R denote the sender and receiver. When the adversary replies with $(\texttt{ok}, sid, m', R')$, $\mathcal{F}_{\mathsf{TLS}}$ sends out this message m' to the receiver R'.

4. The adversary can send $(\texttt{GetHandshakeKeys}, sid)$ to $\mathcal{F}_{\mathsf{TLS}}$ for the server handshake key and IV after the server's handshake response has been delivered. This is secure as discussed in Section 7.9.4. $\mathcal{F}_{\mathsf{TLS}}$ responds with $(\texttt{reveal}, sid, skey, siv, ckey, civ)$ where $skey$ and siv are the server handshake key and IV, and $ckey$ and civ are the client handshake key and IV.

5. If any one of the TLS client and server exits, either because there is an error due to invalid messages or because the TLS session ends normally, $\mathcal{F}_{\mathsf{TLS}}$ considers the session with session ID sid ended and no longer handles requests for this sid.

6. $\mathcal{F}_{\mathsf{TLS}}$ ignores other inputs and messages.

Multiparty computation functionality. In the hybrid model, we abstract SPDZ and AG-MPC as an ideal functionality $\mathcal{F}_{\mathsf{SMPC}}$, which provides the functionality of multiparty computation with abort. We require $\mathcal{F}_{\mathsf{SMPC}}$ to be reactive, meaning that it can take some input and reveal some output midway through execution, as specified in the function f being computed. A reactive SMPC can be constructed from a non-reactive SMPC scheme by secret-sharing the internal state among the N parties in a non-malleable manner, as discussed in [609]. $\mathcal{F}_{\mathsf{SMPC}}$ works as follows:

1. For each session sid, $\mathcal{F}_{\mathsf{SMPC}}$ waits for party $\mathcal{P}_i$ to send $(\texttt{input}, sid, i, x_i, f)$, in which sid is the session ID, i is the party ID, x_i is the party's input, and f is the function to be executed.

2. Once $\mathcal{F}_{\mathsf{SMPC}}$ receives all the N inputs, it checks if all parties agree on the same f, if so, it computes the function $f(x_1, x_2, ..., x_N) \to (y_1, y_2, ..., y_N)$ and sends $(\texttt{output}, sid, i, y_i)$ to party $\mathcal{P}_i$. Otherwise, it terminates this session and sends $(\texttt{abort}, sid)$ to all the N parties.

3. If $\mathcal{F}_{\mathsf{SMPC}}$ receives $(\texttt{Abort}, sid)$ from any of the N parties, it sends $(\texttt{abort}, sid)$ to all the N parties.

4. $\mathcal{F}_{\mathsf{SMPC}}$ ignores other inputs and messages.

Restricted programmable random oracle. We use commitments in Section 7.3.2 to ensure that in Diffie-Hellman key exchange, the challenge $\alpha \cdot G$ is a random element. This is difficult to do without commitments because the adversary can control up to $N-1$ parties to intentionally affect the result of $\alpha \cdot G = \sum_{i=1}^{N} \alpha_i \cdot G$. In our security proof, we leverage a restricted programmable random oracle [607, 608], which is described as follows:

1. $\mathcal{F}_{\mathsf{rpRO}}$ maintains an initially empty list of (m, h) for each session, identified by session ID sid, where m is the message, and h is the digest.

2. Any party can send a query message $(\texttt{Query}, sid, m)$ to $\mathcal{F}_{\mathsf{rpRO}}$ to ask for the digest of message m. If there exists h such that (m, h) is already in the list for session sid, $\mathcal{F}_{\mathsf{rpRO}}$ returns $(\texttt{result}, sid, m, h)$ to this party. Otherwise, it samples h from random, stores (m, h) in the list for sid, and returns $(\texttt{result}, sid, m, h)$.

3. Both the simulator $\mathcal{S}$ and the real-world adversary $\mathcal{A}$ can send a message $(\texttt{Program}, m, h)$ to $\mathcal{F}_{\mathsf{rpRO}}$ to program the random oracle at an unspecified point h, meaning that there does not exist m such that (m, h) is on the list.

4. In the real world, all the parties can check if a hash is programmed, which means that if $\mathcal{A}$ programs a point, other parties would discover. However, in the ideal world, only $\mathcal{S}$ can perform such a check, and thus $\mathcal{S}$ can forge the adversary's state as if no point had been programmed.

7.9.3 Simulator

We now describe the simulator $\mathcal{S}$. Without loss of generality, we assume the attacker compromises exactly $N-1$ servers and does not abort the protocol, and we also assume that $\mathcal{A}$ does not program the random oracle, since in the real world, any parties can detect that and can then abort. We now follow the TLS workflow to do simulation. As follows, we use I to denote the set of identifiers of the compromised servers.

1. Simulator $\mathcal{S}$ provides the inputs of the compromised servers to $\mathcal{F}_{\mathsf{TLS}}$, which would start the TLS protocol.
2. $\mathcal{S}$ lets $\mathcal{F}_{\mathsf{TLS}}$ proceed in the TLS protocol and obtains the `ClientHello` message, which contains a random $\alpha \cdot G$. Now, $\mathcal{S}$ simulates the distributed generation of $\alpha \cdot G$ as follows:
   a) $\mathcal{S}$ samples a random h in the digest domain, pretends that it is the honest party's commitment, and generates the commitments of $\alpha_i \cdot G$ for $i \in I$.
   b) $\mathcal{S}$ sends $(\texttt{Program}, r||(\alpha \cdot G - \sum_{i \in I} \alpha_i \cdot G), h)$ to $\mathcal{F}_{\mathsf{rpRO}}$, where r is the randomness used for making a commitment, and $||$ is concatenation. As a result, $\mathcal{S}$ can open the commitment h to be $\alpha \cdot G - \sum_{i \in I} \alpha_i \cdot G$.
   c) $\mathcal{S}$ continues with the TLS-in-SMPC protocol, in which the N parties open the commitments and construct $\alpha \cdot G$ as the client challenge.
3. $\mathcal{S}$ lets $\mathcal{F}_{\mathsf{TLS}}$ proceed in the TLS protocol and obtains the messages from `ServerHello` to `ClientFinished`, which contain $\beta \cdot G$ and ciphertexts of the server's certificate, the server's signature of $\beta \cdot G$, and the server verification data. Now $\mathcal{S}$ needs to simulate the rest of the key exchange.
   a) $\mathcal{S}$ sends $(\texttt{GetHandshakeKeys}, sid)$ to $\mathcal{F}_{\mathsf{TLS}}$ to obtain the server/client handshake key and IV.
   b) $\mathcal{S}$ simulates the computation of the handshake keys in SMPC by pretending that the SMPC output is the handshake keys. Note: we already assume that without loss of generality, the compromised servers provide the correct $\alpha\beta \cdot G$. If they provide incorrect values, $\mathcal{S}$ would have detected this and can replace the output with an incorrect key.
   c) $\mathcal{S}$ then simulates the remaining operations of key exchange in SMPC, which checks the server verification data and produces the client verification data.
4. $\mathcal{S}$ simulates the message encryption and decryption of the application messages by simply pretending the SMPC output is exactly the ciphertexts taken from actual TLS messages, also provided by $\mathcal{F}_{\mathsf{TLS}}$.
5. In the end, $\mathcal{S}$ outputs whatever the adversary $\mathcal{A}$ would output in the real world.

7.9.4 Proof of indistinguishability

We now argue that the two worlds' outputs are computationally indistinguishable. The outputs are almost identical, so we only need to discuss the differences.

1. In distributed generation of $\alpha \cdot G$, the only difference in the simulated output compared with Π's is that the honest party chooses its share as $\alpha \cdot G - \sum_{i \in I} \alpha_i G$ and uses a programmed hash value h for commitment. Since $\alpha \cdot G$ is sampled from random by the TLS client inside $\mathcal{F}_{\mathsf{TLS}}$, it has the same distribution as the $\alpha_i \cdot G$ sampled by an honest party. The properties of restricted programmable random oracle $\mathcal{F}_{\mathsf{rpRO}}$ show that no parties can detect that h has been programmed.

2. For the remaining operations, the main difference is that the SMPC is simulated without the honest party's secret (in the real-world protocol Π, such secret is a share of the internal SMPC state that contains the TLS session keys). The properties of SMPC show that such simulation is computationally indistinguishable.

As a result, we have the following theorem.

Theorem 37. *Assuming secure multiparty computation, random oracle, and other standard cryptographic assumptions, the TLS-in-SMPC protocol Π with N parties securely realizes the TLS client ideal functionality $\mathcal{F}_{\mathsf{TLS}}$ in the presence of a malicious attacker that statically compromises up to $N - 1$ out of the N parties.*

Bibliography

[1] *CVS health faces data breach, 1b search records exposed*, https://healthitsecurity.com/news/cvs-health-faces-data-breach1b-search-records-exposed.

[2] *Phish leads to breach at Calif. State Controller*, https://krebsonsecurity.com/2021/03/phish-leads-to-breach-at-calif-state-controller/.

[3] *Enterprise software developer exposed 82 million logging records, among them Amazon-owned company*, https://cooltechzone.com/leaks/enterprise-software-developer-exposed-millions-of-logging-records-of-amazon-owned-company.

[4] *Geico admits fraudsters stole customers' driver's license numbers for months*, https://techcrunch.com/2021/04/19/geico-driver-license-numbers-scraped/.

[5] *Data breach warning after California DMV contractor hit by file-stealing ransomware*, https://techcrunch.com/2021/02/18/california-motor-vehicles-afts-ransomware/.

[6] *Another data leak for Experian; credit scores of americans were available to anyone due to api security issue*, https://www.cpomagazine.com/cyber-security/another-data-leak-for-experian-credit-scores-of-americans-were-available-to-anyone-due-to-api-security-issue/.

[7] *Facebook data on 533 million users posted online*, https://www.zdnet.com/article/facebook-data-on-533-million-users-posted-online/.

[8] *Microsoft: These Exchange Server zero-day flaws are being used by hackers, so update now*, https://www.zdnet.com/article/update-immediately-microsoft-rushes-out-patches-for-exchange-server-zero-day-attacks/.

[9] *Another 500 million accounts have leaked online, and LinkedIn's in the hot seat*, https://www.theverge.com/2021/4/8/22374464/linkedin-data-leak-500-million-accounts-scraped-microsoft.

[10] *Hackers breach U.S. cellular customer database after scamming employees*, https://www.forbes.com/sites/leemathews/2021/01/30/hackers-breach-us-cellular-customer-database-after-scamming-employees/.

[11] *Clubhouse data leak: 1.3 million scraped user records leaked online for free*, https://cybernews.com/security/clubhouse-data-leak-1-3-million-user-records-leaked-for-free-online/.

[12] *Volkswagen, Audi notify 3.3 million of data breach*, https://www.bankinfosecurity.com/volkswagen-audi-notify-33-million-people-data-breach-a-16875.

[13] *Check your permissions: Default settings in Microsoft tool exposes 38 million user records online*, https://www.theverge.com/2021/8/24/22639106/microsoft-power-apps-default-permissions-settings-user-records-exposed-38-million-upgard.

[14] *Annual number of data breaches and exposed records in the United States from 2005 to 2020*, https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/.

[15] *6 times when hackers forced companies to go bankrupt and shut down*, https://privacysavvy.com/security/business/6-times-hackers-forced-companies-to-go-bankrupt-shut-down/.

[16] *Cimcor: 60 percent of small companies close within 6 months of being hacked: Small to mid-sized companies should be monitoring networks for suspicious activity*, https://cybersecurityventures.com/60-percent-of-small-companies-close-within-6-months-of-being-hacked/.

[17] *2021 cyber security statistics: The ultimate list of stats, data & trends*, https://purplesec.us/resources/cyber-security-statistics/.

[18] *Survey: Phishing & ransomware attacks are top concerns*, https://www.trendmicro.com/en_us/research/21/g/survey-phishing-ransomware-attacks-are-top-concerns.html.

[19] *The rise of ransomware during COVID-19*, https://home.kpmg/xx/en/home/insights/2020/05/rise-of-ransomware-during-covid-19.html.

[20] *Why ransomware attacks are on the rise–and what can be done to stop them*, https://www.pbs.org/newshour/nation/why-ransomware-attacks-are-on-the-rise-and-what-can-be-done-to-stop-them.

[21] *DHS, FBI say Russian hackers targeting US state and local systems*, https://thehill.com/policy/cybersecurity/522368-dhs-fbi-say-russian-hackers-targeting-us-state-and-local-systems.

[22] *The U.S. government spent billions on a system for detecting hacks: The Russians outsmarted it*, https://www.washingtonpost.com/national-security/ruusian-hackers-outsmarted-us-defenses/2020/12/15/3deed840-3f11-11eb-9453-fc36ba051781_story.html.

[23] *The state of consumer data privacy laws in the US (and why it matters)*, https://www.nytimes.com/wirecutter/blog/state-of-privacy-laws-in-us/.

[24] *Securing the defense industrial base: CMMC 2.0*, https://www.acq.osd.mil/cmmc/.

[25] *Sp 800-171 rev. 2: Protecting controlled unclassified information in nonfederal systems and organizations*, https://csrc.nist.gov/publications/detail/sp/800-171/rev-2/final.

[26] *Federal information security modernization act*, https://www.cisa.gov/federal-information-security-modernization-act.

[27] *The federal risk and authorization management program (FedRAMP)*, https://www.fedramp.gov/.

[28] *Post-quantum cryptography*, https://csrc.nist.gov/projects/post-quantum-cryptography.

[29] *ZKProof standards*, https://zkproof.org/.

[30] *18 new cybersecurity bills introduced as us congressional interest heats up*, https://www.csoonline.com/article/3626908/18-new-cybersecurity-bills-introduced-as-us-congressional-interest-heats-up.html.

[31] *Infrastructure bill includes $1.9 billion for cybersecurity*, https://www.csoonline.com/article/3639019/whats-next-in-congress-for-cybersecurity-after-enactment-of-the-infrastructure-bill.html.

[32] W. Zheng, R. Deng, W. Chen, R. A. Popa, A. Panda, and I. Stoica, "Cerebro: A platform for multi-party cryptographic collaborative learning," in *SEC '21*.

[33] W. Chen, K. Sotiraki, I. Chang, M. Kantarcioglu, and R. A. Popa, "HOLMES: A platform for detecting malicious inputs in secure collaborative computation," in *IACR ePrint 2021/1517*.

[34] W. Chen, A. Chiesa, E. Dauterman, and N. P. Ward, "Reducing participation costs via incremental verification for ledger systems," in *IACR ePrint 2020/1522*.

[35] *Aleo: Where applications become private*, https://aleo.org/.

[36] W. Chen and R. A. Popa, "Metal: A metadata-hiding file-sharing system," in *NDSS '20*.

[37] W. Chen, R. Deng, and R. A. Popa, "N-for-1 auth: N-wise decentralized authentication via one authentication," in *IACR ePrint 2021/342*.

[38] W. Chen, T. Hoang, J. Guajardo, and A. A. Yavuz, "Titanium: A metadata-hiding file-sharing system with malicious security," in *NDSS '22*.

[39] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *NDSS'03*.

[40] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *FAST '03*.

[41] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, “DepSky: Dependable and secure storage in a cloud-of-clouds,” in *EuroSys ’11*.

[42] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan, “Building web applications on top of encrypted data using Mylar,” in *NSDI ’14*.

[43] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, “ShadowCrypt: Encrypted web applications for everyone,” in *CCS ’14*.

[44] B. Lau, S. 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 *SEC ’14*, 2014.

[45] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan, “Sieve: Cryptographically enforced access control for user data in untrusted clouds,” in *NSDI ’16*.

[46] A. Rusbridger, “The Snowden leaks and the public,” in *The New York Review of Books—NYR Daily November 21, 2013*.

[47] D. Cole, “We kill people based on metadata,” in *The New York Review of Books—NYR Daily May 10, 2014*.

[48] *World health organization (WHO): Health statistics and information systems*, `https://www.who.int/healthinfo/en/`.

[49] C. C. Aggarwal, “On k-anonymity and the curse of dimensionality,” in *VLDB ’05*.

[50] L. Backstrom and J. Dwork Cynthia and Kleinberg, “Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography,” in *WWW ’07*.

[51] A. Narayanan and V. Shmatikov, “Robust de-anonymization of large sparse datasets,” in *S&P ’08*.

[52] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, “A practical attack to de-anonymize social network users,” in *S&P ’10*.

[53] S. Nilizadeh, A. Kapadia, and Y.-Y. Ahn, “Community-enhanced de-anonymization of online social networks,” in *CCS ’14*.

[54] S. Ji, W. Li, P. Mittal, X. Hu, and R. Beyah, “SecGraph: A uniform and open-source evaluation system for graph data anonymization and de-anonymization,” in *SEC ’15*.

[55] S. Ji, W. Li, N. Z. Gong, P. Mittal, and R. Beyah, “On your social network de-anonymizablity: Quantification and large scale evaluation with seed knowledge,” in *NDSS’15*.

[56] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov, “Breaking web applications built on top of encrypted data,” in *CCS ’16*.

[57] M. Islam, M. Kuzu, and M. Kantarcioglu, “Access pattern disclosure on searchable encryption: Ramification, attack and mitigation,” in *NDSS ’12*.

[58] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, “Leakage-abuse attacks against searchable encryption,” in *CCS ’15*.

[59] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are [*sic*] belong to us: The power of file-injection attacks on searchable encryption," in *SEC '16*.

[60] L. Wang, P. Grubbs, J. Lu, V. Bindschaedler, D. Cash, and T. Ristenpart, "Side-channel attacks on shared search indexes," in *S&P '17*.

[61] M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Improved reconstruction attacks on encrypted data using range query leakage," in *S&P '18*.

[62] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *CCS '18*.

[63] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Learning to reconstruct: Statistical learning theory and encrypted database attacks," in *S&P '19*.

[64] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs, "Private anonymous data access," in *EUROCRYPT '19*.

[65] M. Backes, A. Herzberg, A. Kate, and I. Pryvalov, "Anonymous RAM," in *ESORICS '16*.

[66] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, "Privacy and access control for outsourced personal records," in *S&P '15*.

[67] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, "Maliciously secure multi-client ORAM," in *ACNS '17*.

[68] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *STOC '87*.

[69] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *STOC '90*.

[70] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," in *CCS '13*.

[71] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *S&P '17*.

[72] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia, "Splinter: Practical private queries on public data," in *NSDI'17*, 2017.

[73] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics," in *NSDI '17*.

[74] N. Kilbertus, A. Gascón, M. Kusner, M. Veale, K. Gummadi, and A. Weller, "Blind Justice: Fairness with encrypted sensitive attributes," in *ICML '18*.

[75] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((logN)^3)$ worst-case cost," in *ASIACRYPT '11*.

[76] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: A parallel oblivious file system," in *CCS '12*.

[77] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious cloud storage," in *S&P '13*.

[78] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward," in *CCS '15*.

[79] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, "TaoStore: Overcoming asynchronicity in oblivious data storage," in *S&P '16*.

[80] A. Chakraborti and R. Sion, "ConcurORAM: High-throughput stateless parallel multi-client ORAM," in *NDSS '19*.

[81] A. C.-C. Yao, "How to generate and exchange secrets," in *FOCS'86*.

[82] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game: A completeness theorem for protocols with honest majority," in *STOC '87*.

[83] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *STOC'90*.

[84] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis, "Secure two-party computation in sublinear (amortized) time," in *CCS '12*.

[85] S. Lu and R. Ostrovsky, "Distributed oblivious RAM for secure two-party computation," in *TCC '13*.

[86] J. B. Nielsen and S. Ranellucci, "Reactive garbling: Foundation, instantiation, application," in *ASIACRYPT '16*.

[87] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," in *CACM '66*.

[88] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *SEC '04*.

[89] H. Corrigan-Gibbs, D. Boneh, and D. Mazières, "Riposte: An anonymous messaging system handling millions of users," in *S&P '15*.

[90] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, "Vuvuzela: Scalable private messaging resistant to traffic analysis," in *SOSP '15*.

[91] S. Angel and S. Setty, "Unobservable communication over fully untrusted infrastructure," in *OSDI'16*.

[92] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich, "Stadium: A distributed metadata-private messaging system," in *SOSP '17*.

[93] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford, "Atom: Horizontally scaling strong anonymity," in *SOSP '17*.

[94] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The Loopix anonymity system," in *SEC '17*.

[95] S. Angel, H. Chen, K. Laine, and S. Setty, "PIR with compressed queries and amortized query processing," in *S&P '18*.

[96] A. Kwon, D. Lu, and S. Devadas, "XRD: Scalable messaging system with cryptographic privacy," in *NSDI '20*.

[97] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *ICALP '08*.

[98] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *S&P '13*.

[99] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole: Reducing data transfer in garbled circuits using half gates," in *EUROCRYPT '15*.

[100] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound," in *CCS '15*.

[101] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square-root ORAM: Efficient random access in multi-party computation," in *S&P '16*.

[102] J. Doerner and A. Shelat, "Scaling ORAM for secure computation," in *CCS '17*.

[103] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *NDSS '12*.

[104] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing ORAM and using it efficiently for secure computation," in *PETS '13*.

[105] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *CRYPTO '84*.

[106] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *PKC '06*.

[107] M. Hamburg, "Decaf: Eliminating cofactors through point compression," in *CRYPTO '15*.

[108] *The Ristretto group*, `https://ristretto.group/ristretto.html`.

[109] D. S. Roche, A. Aviv, and S. G. Choi, "A practical oblivious map data structure with secure deletion and history independence," in *S&P '16*.

[110] M. O. Rabin, "How to exchange secrets with oblivious transfer," in *Technical Report TR-81, Aiken Computation Lab, Harvard University '81*.

[111] S. Even, O. Goldreich, and A. Lempel, "A randomized protocol for signing contracts," in *CACM '85*.

[112] E. V. Mangipudi, K. Rao, J. Clark, and A. Kate, "Towards automatically penalizing multimedia breaches," in *EuroS&PW '19*.

[113] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: A constant bandwidth blowup oblivious RAM," in *TCC '16*.

[114] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT '99*.

[115] S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *STOC'82*.

[116] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *ITCS '12*.

[117] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *PKC '10*.

[118] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," in *Designs, Codes and Cryptography '14*.

[119] *HElib: An implementation of homomorphic encryption*, `https://github.com/homenc/HElib`.

[120] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine learning classification over encrypted data," in *NDSS '15*.

[121] *OpenPGP*, `https://www.openpgp.org/`.

[122] *Autocrypt: Convenient end-to-end encryption for e-mail*, `https://autocrypt.org/`.

[123] B. Kulynych, W. Lueks, M. Isaakidis, G. Danezis, and C. Troncoso, "ClaimChain: Improving the security and privacy of in-band key distribution for messaging," in *WPES '18*.

[124] S. Zahur and D. Evans, "Obliv-C: A language for extensible data-oblivious computation," in *IACR ePrint 2015/1153*.

[125] C. Guo, J. Katz, X. Wang, C. Weng, and Y. Yu, "Better concrete security for half-gates garbling (in the multi-instance setting)," in *CRYPTO '20*.

[126] C. Guo, J. Katz, X. Wang, and Y. Yu, "Efficient and secure multiparty computation from fixed-key block ciphers," in *S&P '20*, 2020.

[127] J. Doerner, *Absentminded Crypto Kit*, `https://bitbucket.org/jackdoerner/absentminded-crypto-kit`, 2018.

[128] M. Bellare, P. Rogaway, and D. Wagner, "The EAX mode of operation," in *FSE '04*.

[129] *Benchmark network throughput between Amazon EC2 Linux instances in the same VPC*, `https://aws.amazon.com/premiumsupport/knowledge-center/network-throughput-benchmark-linux-ec2/`.

[130] T.-H. H. Chan and E. Shi, "Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs," in *TCC '17*.

[131] C.-P. Schnorr, "Efficient identification and signatures for smart cards," in *CRYPTO '89*.

[132] R. Cramer, I. Damgård, and B. Schoenmakers, "Proofs of partial knowledge and simplified design of witness hiding protocols," in *CRYPTO '94*.

[133] M. Jakobsson and A. Juels, "Millimix: Mixing in small batches," in *DIMACS TR '99*.

[134] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," in *PODC'90*.

[135] *Keybase filesystem (KBFS)*, `https://github.com/keybase/kbfs`.

[136] *Preveil: Encrypted email and file sharing for the enterprise*, `https://www.preveil.com/`.

[137] *Tresorit: Secure file sharing & content collaboration with encryption*, `https://tresorit.com/secure-file-sharing`.

[138] R. Anderson, "The Eternity service," in *PRAGOCRYPT '96*.

[139] M. Waldman, A. D. Rubin, and L. F. Cranor, "Publius: A robust, tamper-evident, censorship-resistant web publishing system," in *SEC '00*.

[140] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *International Workshop on Designing Privacy Enhancing Technologies '01*.

[141] R. Dingledine, M. J. Freedman, and D. Molnar, "The Free Haven Project: Distributed anonymous storage service," in *PET '01*.

[142] *Mojo Nation*, `https://sourceforge.net/projects/mojonation/`.

[143] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *CRYPTO '10*.

[144] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious RAM simulation," in *ICALP '11*.

[145] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "PHANTOM: Practical oblivious computation in a secure processor," in *CCS '13*.

[146] J. Dautrich, E. Stefanov, and E. Shi, "Burst ORAM: Minimizing ORAM response times for bursty access patterns," in *SEC '14*.

[147] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious RAM," in *SEC '15*.

[148] E.-O. Blass, T. Mayberry, and G. Noubir, "Multi-client oblivious RAM secure against malicious servers," in *ACNS '17*.

[149] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *CCS '13*.

[150] J. Zhang, W. Zhang, and D. Qiao, "MU-ORAM: Dealing with stealthy privacy attacks in multi-user data outsourcing services," in *IACR ePrint 2016/073*.

[151] N. P. Karvelas, A. Peter, and S. Katzenbeisser, "Using oblivious RAM in genomic studies," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology '17*.

[152] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," in *JACM '96*.

[153] *Dropbox*, `https://www.dropbox.com/`.

[154] X. Wang, Y. Huang, T.-H. H. Chan, A. shelat, and E. Shi, "SCORAM: Oblivious RAM for secure computation," in *CCS '14*.

[155] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *S&P '18*.

[156] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "OBLIVIATE: A data oblivious file system for Intel SGX," in *NDSS'18*.

[157] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious memory primitives from Intel SGX," in *NDSS '18*.

[158] *Boxcryptor: No. 1 cloud encryption made in Germany*, `https://www.boxcryptor.com/en/`.

[159] *Icedrive: Secure encrypted cloud storage*, `https://icedrive.net/encrypted-cloud-storage`.

[160] *Mega: Secure cloud storage and communication*, `https://mega.io/`.

[161] *pCloud encryption: Best secure encrypted cloud storage*, `https://www.pcloud.com/encrypted-cloud-storage.html`.

[162] *Sync: Secure cloud storage, privacy guaranteed*, `https://www.sync.com/`.

[163] B. Auxier, L. Rainie, M. Anderson, A. Perrin, M. Kumar, and E. Turner, *Americans and privacy: Concerned, confused and feeling lack of control over their personal information*, Pew Research Center (2019).

[164] R. Brandom, G. Blackmon, and W. Joel, *Ten years of breaches in one image: Nearly 8 billion usernames have leaked since june 2011*, Available at `https://www.theverge.com/22518557/data-breach-infographic-leaked-passwords-have-i-been-pwned`.

[165] J. Mayer, P. Mutchler, and J. C. Mitchell, "Evaluating the privacy properties of telephone metadata," in *PNAS '16*.

[166] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *S&P '08*.

[167] G. G. Gulyás, B. Simon, and S. Imre, "An efficient and robust social network de-anonymization attack," in *WPES '16*.

[168] J. Feng, M. Zhang, H. Wang, Z. Yang, C. Zhang, Y. Li, and D. Jin, "DPLink: User identity linkage via deep neural network from heterogeneous mobility data," in *WWW '19*.

[169] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *NDSS '12*.

[170] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Inference attack against encrypted range queries on outsourced databases," in *CODASPY '14*.

[171] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *CCS '15*.

[172] G. Kellaris, G. Kollios, K. Nissim, and A. O'neill, "Generic attacks on secure outsourced databases," in *CCS '16*.

[173] C. V. Rompay, R. Molva, and M. Önen, "A leakage-abuse attack against multi-user searchable encryption," in *PETS '17*.

[174] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks," in *NDSS '20*.

[175] X. Zhuang, T. Zhang, and S. Pande, "HIDE: An infrastructure for efficiently protecting information leakage on the address bus," in *ACM SIGOPS Operating Systems Review '04*.

[176] T. M. John, S. K. Haider, H. Omar, and M. Van Dijk, "Connecting the dots: Privacy leakage via write-access patterns to the main memory," in *IEEE TDSC '17*.

[177] S. S. Chow, K. Fech, R. W. Lai, and G. Malavolta, "Multi-client oblivious RAM with poly-logarithmic communication," in *ASIACRYPT '20*.

[178] D. Mazières and D. Shasha, "Building secure file systems out of Byzantine storage," in *PODC '02*.

[179] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *OSDI '04*.

[180] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group collaboration using untrusted cloud resources," in *OSDI '10*.

[181] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun, "Verena: End-to-end integrity protection for web applications," in *S&P '16*.

[182] A. Tomescu and S. Devadas, "Catena: Efficient non-equivocation via Bitcoin," in *S&P '17*.

[183] Y. Hu, S. Kumar, and R. A. Popa, "Ghostor: Toward a secure data-sharing system from decentralized trust," in *NSDI '20*.

[184] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *FAST '11*.

[185] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A capacity-optimized SSD cache for primary storage," in *ATC '14*.

[186] O. Goldreich, "Towards a theory of software protection," in *CRYPTO '86*.

[187] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," in *CCS '13*.

[188] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits," in *ESORICS '13*.

[189] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Zero-knowledge proofs on secret-shared data via fully linear PCPs," in *CRYPTO '19*.

[190] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou, "Prio+: Privacy preserving aggregate statistics via Boolean shares," in *IACR ePrint 2021/576*.

[191] J. T. Schwartz, "Fast probabilistic algorithms for verification of polynomial identities," in *JACM '80*.

[192] R. Zippel, "Probabilistic algorithms for sparse polynomials," in *EUROSAM '79*.

[193] R. A. Demillo and R. J. Lipton, "A probabilistic remark on algebraic program testing," in *Information Processing Letters '78*.

[194] E. Boyle, N. Gilboa, and Y. Ishai, “Function secret sharing: Improvements and extensions,” in *CCS ’16*.

[195] J. Kilian, “Founding cryptography on oblivious transfer,” in *STOC ’88*.

[196] M. Naor and B. Pinkas, “Oblivious transfer with adaptive queries,” in *CRYPTO ’99*.

[197] C. Dwork, M. Naor, O. Reingold, and L. Stockmeyer, “Magic functions,” in *FOCS ’99*.

[198] J. Camenisch, G. Neven, and A. Shelat, “Simulatable adaptive oblivious transfer,” in *EUROCRYPT ’07*.

[199] Y. Lindell and B. Pinkas, “Secure two-party computation via cut-and-choose oblivious transfer,” in *TCC ’11*.

[200] Y. Lindel, “How to simulate it: A tutorial on the simulation proof technique,” in *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*. 2017.

[201] T. Hoang, J. Guajardo, and A. A. Yavuz, “MACAO: A maliciously-secure and client-efficient active ORAM framework,” in *NDSS ’20*.

[202] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, “Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset,” in *PETS ’19*.

[203] T. Hoang, R. Behnia, Y. Jang, and A. A. Yavuz, “MOSE: Practical multi-user oblivious storage via secure enclaves,” in *CODASPY ’20*.

[204] E. Boyle, K.-M. Chung, and R. Pass, “Oblivious parallel RAM and applications,” in *TCC ’16-A*.

[205] B. Chen, H. Lin, and S. Tessaro, “Oblivious parallel RAM: Improved efficiency and generic constructions,” in *TCC ’16-A*.

[206] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, “Multiparty computation from somewhat homomorphic encryption,” in *CRYPTO ’12*.

[207] M. Ben-Or, S. Goldwasser, and A. Wigderson, “Completeness theorems for non-cryptographic fault-tolerant distributed computation,” in *STOC ’88*.

[208] D. Chaum, C. Crépeau, and I. Damgård, “Multiparty unconditionally secure protocols,” in *STOC ’88*.

[209] D. Beaver, S. Micali, and P. Rogaway, “The round complexity of secure protocols,” in *STOC ’90*.

[210] C. Liu, X. Wang, K. Nayak, Y. Huang, and E. Shi, “ObliVM: A programming framework for secure computation,” in *S&P ’15*.

[211] *ObliVMGC: The garbled circuit backend for the ObliVM framework*, `https://github.com/oblivm/ObliVMGC`.

[212] *FlexSC: A flexible efficient secure computation backend*, `https://github.com/wangxiao1254/FlexSC`.

[213] *ORAM library for Obliv-C*, `https://github.com/samee/sqrtOram`.

[214] *The absentminded crypto kit*, `https://bitbucket.org/jackdoerner/absentminded-crypto-kit/src`.

[215] *Multi-protocol SPDZ*, https://github.com/data61/MP-SPDZ/.

[216] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *CCS '20*.

[217] *SCALE and MAMBA*, `https://github.com/KULeuven-COSIC/SCALE-MAMBA`.

[218] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO '86*.

[219] T. P. Jakobsen, J. B. Nielsen, and C. Orlandi, "A framework for outsourcing of secure computation," in *CCSW '14*.

[220] I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft, "Confidential benchmarking based on multiparty computation," in *FC '16*.

[221] *Efficient multiparty computation toolkit (EMP-toolkit)*, https://github.com/emp-toolkit.

[222] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh, "Express: Lowering the cost of metadata-hiding communication with cryptographic privacy," in *SEC '21*.

[223] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *CRYPTO '12*.

[224] S. Halevi and V. Shoup, "Bootstrapping for HElib," in *EUROCRYPT '15*.

[225] P. Grubbs, T. Ristenpart, and V. Shmatikov, "Why your encrypted database is not secure," in *HotOS '17*.

[226] H. Chen, I. Chillotti, and L. Ren, "Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE," in *CCS '19*.

[227] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen, "S3ORAM: A computation-efficient and constant client bandwidth blowup ORAM with Shamir secret sharing," in *CCS '17*.

[228] T.-H. H. Chan, J. Katz, K. Nayak, A. Polychroniadou, and E. Shi, "More is less: Perfectly secure oblivious algorithms in the multi-server setting," in *ASIACRYPT '18*.

[229] S. D. Gordon, J. Katz, and X. Wang, "Simple and efficient two-server ORAM," in *ASIACRYPT '18*.

[230] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu, "Toward robust hidden volumes using write-only oblivious RAM," in *CCS '14*.

[231] D. S. Roche, A. Aviv, S. G. Choi, and T. Mayberry, "Deterministic, stash-free write-only ORAM," in *CCS '17*.

[232] A. J. Aviv, S. G. Choi, T. Mayberry, and D. S. Roche, "ObliviSync: Practical oblivious file backup and synchronization," in *NDSS '17*.

[233] T.-H. H. Chan, K.-M. Chung, and E. Shi, "On the depth of oblivious parallel RAM," in *ASIACRYPT '17*.

[234] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman, "Shroud: Ensuring private access to large-scale data in the data center," in *FAST '13*.

[235] S. Eskandarian and M. Zaharia, "ObliDB: Oblivious query processing for secure databases," in *VLDB '19*.

[236] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for Intel SGX," in *NDSS '18*.

[237] C. Bao and A. Srivastava, "Exploring timing side-channel attacks on Path-ORAMs," in *HOST '17*.

[238] C. W. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs," in *HPCA '14*.

[239] D. Lazar and N. Zeldovich, "Alpenhorn: Bootstrapping secure communication without leaking metadata," in *OSDI '16*.

[240] D. Chaum, "Blind signatures for untraceable payments," in *CRYPTO '82*.

[241] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *CRYPTO '92*, 1992.

[242] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols," in *CMS'99*, 1999.

[243] A. Juels and J. Brainard, "Client puzzles: A cryptographic defense against connection depletion attacks," in *NDSS'99*, 1999.

[244] TensorFlow, *Federated learning*, `https://www.tensorflow.org/federated/federated_learning`.

[245] S. Bhattacharya, *The new dawn of AI: Federated learning*, `https://towardsdatascience.com/the-new-dawn-of-ai-federated-learning-8ccd9ed7fc3a`, 2019.

[246] A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," in *IEEE Intelligent Systems '09*.

[247] *GDPR*, Official Journal of the European Union '16.

[248] *California Consumer Privacy Act (CCPA) 2018*, `https://oag.ca.gov/privacy/ccpa`, 2018.

[249] I. Giacomelli, S. Jha, M. Joye, C. D. Page, and K. Yoon, "Privacy-preserving ridge regression with only linearly-homomorphic encryption," in *ACNS '18*.

[250] W. Zheng, R. A. Popa, J. Gonzalez, and I. Stoica, "Helen: Maliciously secure coopetitive learning for linear models," in *S&P '19*.

[251] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via MiniONN transformations," in *CCS '17*.

[252] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *SEC '18*.

[253] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, "XONN: XNOR-based oblivious deep neural network inference," in *SEC '19*.

[254] A. Tueno, F. Kerschbaum, and S. Katzenbeisser, "Private evaluation of decision trees using sublinear cost," in *PETS '19*.

[255] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *S&P '13*.

[256] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans, "Privacy-preserving distributed linear regression on high-dimensional data.," in *PETS '17*.

[257] A. B. Alexandru, K. Gatsis, Y. Shoukry, S. A. Seshia, P. Tabuada, and G. J. Pappas, "Cloud-based quadratic optimization with partially homomorphic encryption," in *IEEE Transactions on Automatic Control '20*.

[258] X. Wang, S. Ranellucci, and J. Katz, "Global-scale secure multiparty computation," in *CCS '17*.

[259] *Emp-toolkit: Efficient multiparty computation toolkit.* `https://github.com/emp-toolkit`.

[260] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly compressed and scalable sequential garbled circuits," in *S&P '15*.

[261] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *S&P '14*.

[262] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning," in *EuroS&P '19*.

[263] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "HyCC: Compilation of hybrid protocols for practical secure computation," in *CCS '18*.

[264] Y. Zhang, A. Steele, and M. Blanton, "PICCO: A general-purpose compiler for private distributed computation," in *CCS '13*.

[265] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *EuroS&P '16*.

[266] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, "CBMC-GC: An ANSI C compiler for secure two-party computations," in *CC '14*.

[267] D. Demmler, T. Schneider, and M. Zohner, "ABY: A framework for efficient mixed-protocol secure two-party computation," in *NDSS '15*.

[268] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros, "Conclave: Secure multi-party computation on big data," in *EuroSys '19*.

[269] M. Keller, E. Orsini, and P. Scholl, “MASCOT: Faster malicious arithmetic secure computation with oblivious transfer,” in *CCS ’16*.

[270] M. Keller, V. Pastro, and D. Rotaru, “Overdrive: Making SPDZ great again,” in *EUROCRYPT ’18*.

[271] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song, “The secret sharer: Evaluating and testing unintended memorization in neural networks,” in *SEC ’19*.

[272] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Stealing machine learning models via prediction APIs,” in *SEC ’16*.

[273] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart, “Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing,” in *SEC ’14*.

[274] M. Fredrikson, S. Jha, and T. Ristenpart, “Model inversion attacks that exploit confidence information and basic countermeasures,” in *CCS ’15*.

[275] X. Wu, M. Fredrikson, S. Jha, and J. F. Naughton, “A methodology for formalizing model-inversion attacks,” in *CSF ’16*.

[276] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, “Membership inference attacks against machine learning models,” in *S&P ’17*.

[277] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, “Targeted backdoor attacks on deep learning systems using data poisoning,” in *ArXiv 1712.05526*.

[278] C. Song, T. Ristenpart, and V. Shmatikov, “Machine learning models that remember too much,” in *CCS ’17*.

[279] Y. Long, V. Bindschaedler, L. Wang, D. Bu, X. Wang, H. Tang, C. A. Gunter, and K. Chen, “Understanding membership inferences on well-generalized learning models,” in *ArXiv 1802.04889*.

[280] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, “A set of level 3 basic linear algebra subprograms,” in *ACM Transactions on Mathematical Software ’90*.

[281] *BLAS (Basic Linear Algebra Subprograms)*, `http://www.netlib.org/blas/`.

[282] *Intel Math Kernel Library*, `https://software.intel.com/en-us/mkl`.

[283] S. Palkar, J. J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. P. Amarasinghe, S. Madden, and M. A. Zaharia, “Evaluating end-to-end optimization for data analytics applications in Weld,” in *VLDB ’18*.

[284] S. Diamond and S. Boyd, “CVXPY: A Python-embedded modeling language for convex optimization,” in *Journal of Machine Learning Research ’16*.

[285] C. Dwork, “Differential privacy,” in *ICALP ’06*.

[286] K. Chaudhuri and C. Monteleoni, “Privacy-preserving logistic regression,” in *NeurIPS ’09*.

[287] X. Wu, F. Li, A. Kumar, K. Chaudhuri, S. Jha, and J. Naughton, “Bolt-on differential privacy for scalable stochastic gradient descent-based analytics,” in *SIGMOD ’17*.

[288] R. Iyengar, J. P. Near, D. Song, O. Thakkar, A. Thakurta, and L. Wang, "Towards practical differentially private convex optimization," in *S&P '19*.

[289] G. Brassard, D. Chaum, and C. Crépeau, "Minimum disclosure proofs of knowledge," in *Journal of Computer and System Sciences '88*.

[290] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO '91*.

[291] M. Ajtai, "Generating hard instances of lattice problems," in *STOC '96*.

[292] O. Goldreich, S. Goldwasser, and S. Halevi, "Collision-free hashing from lattice problems," in *IACR ePrint 1996/9*.

[293] J.-Y. Cai and A. Nerurkar, "An improved worst-case to average-case connection for lattice problems," in *FOCS '97*.

[294] D. Micciancio, "Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions," in *FOCS '02*.

[295] D. Micciancio and O. Regev, "Worst-case to average-case reductions based on Gaussian measures," in *FOCS '04*.

[296] C. Peikert and A. Rosen, "Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices," in *TCC '06*.

[297] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," in *CRYPTO '14*.

[298] J. Camenisch, S. Krenn, and V. Shoup, "A framework for practical universally composable zero-knowledge protocols," in *ASIACRYPT '11*.

[299] *AWS inter-region ping*, `https://www.cloudping.co`.

[300] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," in *Foundations and Trends in Machine Learning '10*.

[301] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang, "MadLINQ: Large-scale distributed matrix computation for the cloud," in *EuroSys '12*.

[302] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, *et al.*, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *IPDPSW '11*.

[303] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on MapReduce," in *ICDE '11*.

[304] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *NeurIPS Workshop on Machine Learning Systems '15*.

[305] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *MM '14*.

[306] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, "Mllib: Machine learning in Apache Spark," in *Journal of Machine Learning Research '16*.

[307] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *KDD '16*.

[308] G. B. Team, "TensorFlow: A system for large-scale machine learning," in *OSDI '16*.

[309] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *NSDI '17*.

[310] M. Jarke and J. Koch, "Query optimization in database systems," in *ACM Computer Survey '84*.

[311] S. Viglas and J. F. Naughton, "Rate-based query optimization for streaming information sources," in *SIGMOD '02*.

[312] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," in *ACM Transactions of Database Systems '05*.

[313] A. Friedley and A. Lumsdaine, "Communication optimization beyond MPI," in *IPDPSW '11*.

[314] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," in *PLDI '11*.

[315] M. Wang, C.-C. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *EuroSys '18*.

[316] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *POPL '81*.

[317] M. Ishaq, A. L. Milanova, and V. Zikas, "Efficient MPC via program analysis: A framework for efficient optimal mixing," in *CCS '19*.

[318] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, M. Musuvathi, and T. Mytkowicz, "CHET: An optimizing compiler for fully-homomorphic neural-network inferencing," in *PLDI '19*.

[319] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *PLDI '20*.

[320] A. Rastogi, N. Swamy, and M. Hicks, "Wys*: A DSL for verified secure multi-party computations," in *POST '19*.

[321] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow2: Practical 2-party secure inference," in *CCS '20*.

[322] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved mixed-protocol secure two-party computation," in *SEC '21*.

[323] P. Mohassel and P. Rindal, “Aby3: A mixed protocol framework for machine learning,” in *CCS ’18*.

[324] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, “Chameleon: A hybrid secure computation framework for machine learning applications,” in *ASIACCS ’18*.

[325] A. Baumann, M. Peinado, and G. Hunt, “Shielding applications from an untrusted cloud with Haven,” *OSDI ’14*,

[326] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, “Ryoan: A distributed sandbox for untrusted computation on secret data,” in *OSDI ’15*.

[327] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, “Opaque: An oblivious and encrypted distributed analytics platform,” in *NSDI ’17*.

[328] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, “Chiron: Privacy-preserving machine learning as a service,” in *ArXiv 1803.05961*.

[329] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, “Oblivious multi-party machine learning on trusted processors,” in *SEC ’16*.

[330] K. Grover, S. Tople, S. Shinde, R. Bhagwan, and R. Ramjee, “Privado: Practical and secure DNN inference with enclaves,” in *ArXiv 1810.00602*.

[331] Y. Xu, W. Cui, and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems,” in *S&P ’15*.

[332] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, “Software grand exposure: SGX cache attacks are practical,” in *WOOT ’17*.

[333] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma, “Observing and preventing leakage in MapReduce,” in *CCS ’15*.

[334] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *S&P ’19*.

[335] D. Froelicher, J. R. Troncoso-Pastoriza, A. Pyrgelis, S. Sav, J. S. Sousa, J.-P. Bossuat, and J.-P. Hubaux, “Scalable privacy-preserving distributed learning,” in *PETS ’21*.

[336] S. Tan, B. Knott, Y. Tian, and D. J. Wu, “CryptGPU: Fast privacy-preserving machine learning on the GPU,” in *S&P ’21*.

[337] S. Wagh, D. Gupta, and N. Chandran, “SecureNN: 3-party secure computation for neural network training,” in *PETS ’19*.

[338] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, “BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning,” in *ATC ’20*.

[339] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, “Delphi: A cryptographic inference service for neural networks,” in *SEC ’20*.

[340] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "MP2ML: A mixed-protocol machine learning framework for private inference," in *ARES '20*.

[341] A. Patra and A. Suresh, "BLAZE: Blazing fast privacy-preserving machine learning," in *NDSS '20*.

[342] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4PC framework for privacy preserving machine learning," in *NDSS '20*.

[343] H. Chen, I. Chillotti, Y. Dong, O. Poburinnaya, I. Razenshteyn, and M. S. Riazi, "SANNS: Scaling up secure approximate k-nearest neighbors search," in *SEC '20*.

[344] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *AAAI '19*.

[345] K. Nandakumar, N. Ratha, S. Pankanti, and S. Halevi, "Towards deep neural network training on encrypted data," in *CVPR '19*.

[346] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: Fast and robust framework for privacy-preserving machine learning," in *PETS '20*.

[347] J. H. Ziegeldorf, J. Metzke, and K. Wehrle, "SHIELD: A framework for efficient and secure machine learning classification in constrained environments," in *ACSAC '18*.

[348] V. Chen, V. Pastro, and M. Raykova, "Secure computation for machine learning with SPDZ," in *NeurIPS '18*.

[349] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *CCS '17*.

[350] S. Sav, A. Pyrgelis, J. R. Troncoso-Pastoriza, D. Froelicher, J.-P. Bossuat, J. S. Sousa, and J.-P. Hubaux, "POSEIDON: Privacy-preserving federated neural network learning," in *NDSS '21*.

[351] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High throughput 3PC over rings with application to secure prediction," in *CCSW '19*.

[352] N. Kumar, M. Rathee, N. Chandran, D. Gupta, and R. Rastogi Aseem an Sharma, "CrypTFlow: Secure TensorFlow inference," in *S&P '20*.

[353] M. Abspoel, D. Escudero, and N. Volgushev, "Secure training of decision trees with continuous attributes," in *PETS '21*.

[354] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren, "EPIC: Efficient private image classification (or: Learning from the masters)," in *CT-RSA '19*.

[355] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference," in *CCS '19*.

[356] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón, "QUOTIENT: Two-party secure neural network training and prediction," in *CCS '19*.

[357] S. Li, K. Xue, B. Zhu, C. Ding, X. Gao, D. Wei, and T. Wan, "FALCON: A Fourier transform based approach for fast and secure convolutional neural network predictions," in *CVPR '20*.

[358] A. Dalskov, D. Escudero, and M. Keller, "Fantastic four: Honest-majority four-party secure computation with malicious security," in *SEC '21*.

[359] K. Mandal and G. Gong, "PrivFL: Practical privacy-preserving federated regressions on high-dimensional data over mobile networks," in *CCSW '19*.

[360] S. Bian, T. Wang, M. Hiromoto, Y. Shi, and T. Sato, "ENSEI: Efficient secure inference via frequency-domain homomorphic convolution for privacy-preserving visual recognition," in *CVPR '20*.

[361] H. Chen, M. Kim, I. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh, "Maliciously secure matrix multiplication with applications to private deep learning," in *ASIACRYPT '20*.

[362] P. Mohassel, M. Rosulek, and N. Trieu, "Practical privacy-preserving k-means clustering," in *PETS '20*.

[363] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, "Muse: Secure inference resilient to malicious clients," in *SEC '21*.

[364] C. A. Choquette-Choo, N. Dullerud, A. Dziedzic, Y. Zhang, S. Jha, N. Papernot, and X. Wang, "CaPC learning: Confidential and private collaborative learning," in *ICLR '21*.

[365] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning," in *PETS '21*.

[366] X. Jiang, M. Kim, K. Lauter, and Y. Song, "Secure outsourced matrix computation and application to neural networks," in *CCS '18*.

[367] M. Chase, R. Gilad-Bachrach, K. Laine, K. E. Lauter, and P. Rindal, "Private collaborative neural network learning," in *IACR ePrint 2017/762*.

[368] J. Frankle, S. Park, D. Shaar, S. Goldwasser, and D. Weitzner, "Practical accountability of secret processes," in *SEC '18*.

[369] Y. Ishai, R. Ostrovsky, and H. Seyalioglu, "Identifying cheaters without an honest majority," in *TCC '12*.

[370] Y. Ishai, R. Ostrovsky, and V. Zikas, "Secure multi-party computation with identifiable abort," in *CRYPTO '14*.

[371] C. Baum, B. David, and R. Dowsley, "Insured MPC: Efficient secure computation with financial penalties," in *FC '20*.

[372] K. Hao, *AI is sending people to jail—and getting it wrong*, `https://www.technologyreview.com/2019/01/21/137783/algorithms-criminal-justice-ai/`.

[373] G. Kesari, *AI can now detect depression from your voice, and it's twice as accurate as human practitioners*, `https://bit.ly/32HdcUQ`.

[374] *Ellipsis health*, `https://www.ellipsishealth.com/`.

[375] *Qbtech*, `https://www.qbtech.com/`.

[376] K. Hale, *A.I. bias caused 80% of black mortgage applicants to be denied*, `https://bit.ly/34gzgpN`.

[377] *Mortgage algorithms perpetuate racial bias in lending, study finds*, `https://bit.ly/34kTBdr`.

[378] E. Martinez and L. Kirchner, *The secret bias hidden in mortgage-approval algorithms*, `https://bit.ly/3s4tGiq`.

[379] J. Larson, S. Mattu, L. Kirchner, and J. Angwin, *How we analyzed the compas recidivism algorithm*, `https://bit.ly/3oclWtP`.

[380] M. G. Institute, *Tackling bias in artificial intelligence (and in humans)*, `https://mck.co/3Ge65B8`.

[381] Z. Yu, J. Chakraborty, and T. Menzies, "FairBalance: Improving machine learning fairness on multiple sensitive attributes with data balancing," in *Arxiv:2107.08310*.

[382] A. Wang, A. Narayanan, and O. Russakovsky, "REVISE: A tool for measuring and mitigating bias in visual datasets," in *ECCV '20*.

[383] F. Kamiran and T. Calders, "Data preprocessing techniques for classification without discrimination," *Knowledge and Information Systems*, vol. 33, no. 1, pp. 1–33, 2012.

[384] T. Davidson, D. Bhattacharya, and I. Weber, "Racial bias in hate speech and abusive language detection datasets," in *Workshop on Abusive Language Online '19*.

[385] Nature Communications Editorial, "Data sharing and the future of science," in *Nature Communications '18*.

[386] H. A. Piwowar and T. J. Vision, "Data reuse and the open data citation advantage," in *PeerJ '13*.

[387] M. Packer, "Data sharing in medical research," in *British Medical Journal '18*.

[388] L. Kamm, D. Bogdanov, S. Laur, and J. Vilo, "A new way to protect privacy in large-scale genome-wide association studies," in *Bioinformatics '13*.

[389] E. A. Abbe, A. E. Khandani, and A. W. Lo, "Privacy-preserving methods for sharing financial risk exposures," in *American Economic Review '12*.

[390] *Rights related to automated decision making including profiling*, `https://bit.ly/3ALUJ6m`.

[391] *Assessing data quality for healthcare systems data used in clinical research*, `https://dcricollab.dcri.duke.edu/sites/NIHKR/KR/Assessing-data-quality_V1%200.pdf`.

[392] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein, "Senate: A maliciously-secure MPC platform for collaborative analytics," in *SEC '20*.

[393] S. A. Cook, “The complexity of theorem-proving procedures,” in *STOC ’71*.

[394] K. Yang, P. Sarkar, C. Weng, and X. Wang, “QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field,” in *CCS ’21*.

[395] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang, “Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning,” in *SEC ’21*.

[396] C. Weng, K. Yang, X. Wang, and J. Katz, “Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits,” in *S&P ’21*.

[397] S. Dittmer, Y. Ishai, and R. Ostrovsky, “Line-point zero knowledge and its applications,” in *ITC ’21*.

[398] 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 *CRYPTO ’21*.

[399] E. Boyle, G. Couteau, N. Gilboa, and Y. Ishai, “Compressing vector OLE,” in *CCS ’18*.

[400] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, “Efficient pseudorandom correlation generators: Silent OT extension and more,” in *CRYPTO ’19*.

[401] A. Z. Broder, “On the resemblance and containment of documents,” in *Compression and Complexity of SEQUENCES ’97*.

[402] R. Morris, “Counting large numbers of events in small registers,” in *CACM ’78*.

[403] J. I. Munro and M. S. Paterson, “Selection and sorting with limited storage,” in *FOCS ’78*.

[404] P. Flajolet and G. N. Martin, “Probabilistic counting algorithms for data base applications,” in *JCSS ’85*.

[405] N. Alon, Y. Matias, and M. Szegedy, “The space complexity of approximating the frequency moments,” in *STOC ’96*.

[406] A. Munteanu, S. Omlor, and D. Woodruf, “Oblivious sketching for logistic regression,” in *ICML ’21*.

[407] T. D. Ahle, M. Kapralov, J. B. T. Knudsen, R. Pagh, A. Velingker, D. Woodruff, and A. Zandieh, “Oblivious sketching of high-degree polynomial kernels,” in *SODA ’20*.

[408] W. B. Johnson and J. Lindenstrauss, “Extensions of Lipschitz mappings into a Hilbert space 26,” in *Contemporary mathematics ’84*.

[409] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, “Poseidon: A new hash function for zero-knowledge proof system,” in *SEC ’21*.

[410] A. Aly, T. Ashur, E. Ben-Sasson, S. Dhooghe, and A. Szepieniec, “Design of symmetric-key primitives for advanced cryptographic protocols,” in *FSE ’20*.

[411] I. Damgård, “On the randomness of Legendre and Jacobi sequences,” in *CRYPTO ’88*.

[412] L. Grassi, C. Rechberger, D. Rotaru, P. Scholl, and N. P. Smart, “MPC-friendly symmetric key primitives,” in *CCS ’16*.

[413] D. Khovratovich, “Key recovery attacks on the Legendre PRFs within the birthday bound,” in *IACR ePrint 2019/862*.

[414] A. May and F. Zweydinger, “Legendre PRF (multiple) key attacks and the power of preprocessing,” in *IACR ePrint 2021/645*.

[415] M. N. Wegman and J. L. Carter, “New hash functions and their use in authentication and set equality,” in *JCSS ’81*.

[416] M. Naor and M. Yung, “Universal one-way hash functions and their cryptographic applications,” in *STOC ’89*.

[417] S. Moro, P. Cortez, and P. Ritaa, “A data-driven approach to predict the success of bank telemarketing,” in *Decision Support Systems ’14*.

[418] “Bank marketing data set,” in `https://archive.ics.uci.edu/ml/datasets/Bank+Marketing`.

[419] B. Strack, J. P. DeShazo, C. Gennings, J. L. Olmo, S. Ventura, K. J. Cios, and J. N. Clore, “Impact of HbA1c measurement on hospital readmission rates: Analysis of 70,000 clinical database patient records,” in *BioMed Research International ’14*.

[420] “Diabetes 130-US hospitals for years 1999-2008 data set,” in `https://archive.ics.uci.edu/ml/datasets/Diabetes+130-US+hospitals+for+years+1999-2008`.

[421] “Real time advertiser’s auction,” in `https://www.kaggle.com/saurav9786/real-time-advertisers-auction`.

[422] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai, “Universally composable two-party and multi-party secure computation,” in *STOC ’02*.

[423] D. Evans, V. Kolesnikov, and M. Rosulek, “Defining multi-party computation,” in *A Pragmatic Introduction to Secure Multi-Party Computation*, 2018.

[424] M. Blum, “Coin flipping by telephone a protocol for solving impossible problems,” in *ACM SIGACT News ’83*.

[425] C. F. Gauss, “Theoria combinationis obsevationum erroribus minimis obnoxiae,” in *Carl Friedrich Gauss Werke*, 1823.

[426] E. Arias-Castro, B. Pelletier, and V. Saligrama, “Remember the curse of dimensionality: The case of goodness-of-fit testing in arbitrary dimension,” in *Journal of Nonparametric Statistics ’18*.

[427] R. B. Davies, “Algorithm as 155: The distribution of a linear combination of chi-squared random variables,” *Applied Statistics*, pp. 323–333, 1980.

[428] J. Sheil and I. O’Muircheartaigh, “Algorithm as 106: The distribution of non-negative quadratic forms in normal variables,” *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 26, no. 1, pp. 92–98, 1977.

[429] J.-P. Imhof, “Computing the distribution of quadratic forms in normal variables,” *Biometrika*, vol. 48, no. 3/4, pp. 419–426, 1961.

[430] D. Achlioptas, “Database-friendly random projections: Johnson-Lindenstrauss with binary coins,” in *Journal of Computer and System Sciences ’03*.

[431] S. Venkatasubramanian and Q. Wang, “The Johnson-Lindenstrauss transform: An empirical study,” in *Workshop on Algorithm Engineering and Experiments (ALENEX) ’11*.

[432] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo, “Zero-knowledge contingent payments revisited: Attacks and payments for services,” in *CCS ’17*.

[433] G. Couteau, P. Rindal, and S. Raghuraman, “Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes,” in *CRYPTO ’21*.

[434] A. C.-C. Yao, “Protocols for secure computations,” in *FOCS ’82*.

[435] X. Wang, S. Ranellucci, and J. Katz, “Authenticated garbling and efficient maliciously secure two-party computation,” in *CCS ’17*.

[436] C. Hazay, P. Scholl, and E. Soria-Vazquez, “Low cost constant round MPC combining BMR and oblivious transfer,” in *ASIACRYPT ’17*.

[437] K. Yang, X. Wang, and J. Zhang, “More efficient MPC from improved triple generation and authenticated garbling,” in *CCS ’20*.

[438] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, “Ferret: Fast extension for correlated OT with small communication,” in *CCS ’20*.

[439] S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof-systems,” in *STOC ’85*.

[440] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit, “Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting,” in *EUROCRYPT ’16*.

[441] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, “Bulletproofs: Short proofs for confidential transactions and more,” in *S&P ’18*.

[442] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, “Libra: Succinct zero-knowledge proofs with optimal prover computation,” in *CRYPTO ’19*.

[443] J. Zhang, T. Xie, Y. Zhang, and D. Song, “Transparent polynomial delegation and its applications to zero knowledge proof,” in *S&P ’20*.

[444] S. Setty, “Spartan: Efficient and general-purpose zkSNARKs without trusted setup,” in *CRYPTO ’20*.

[445] A. Andoni, T. Malkin, and N. S. Nosatzki, “Two party distribution testing: Communication and security,” in *NeurIPS PPML Workshop ’18*.

[446] V. Narayanan, M. Mishra, and V. M. Prabhakaran, “Private two-terminal hypothesis testing,” in *ISIT ’20*.

[447] F. R. Hampel, E. M. Ronchetti, P. J. Rousseeuw, and W. A. Stahel, *Robust statistics: The approach based on influence functions*. John Wiley & Sons, 2011, vol. 196.

[448] P. J. Huber, *Robust statistics*. John Wiley & Sons, 2004, vol. 523.

[449] R. A. Maronna, R. D. Martin, V. J. Yohai, and M. Salibián-Barrera, *Robust statistics: Theory and methods (with R)*. John Wiley & Sons, 2019.

[450] W. Beullens, T. Beyne, A. Udovenko, and G. Vitto, "Cryptanalysis of the Legendre PRF and generalizations," in *FSE '20*.

[451] N. Kaluderović, T. Kleinjung, and D. Kostić, "Cryptanalysis of the generalised Legendre pseudorandom function," in *ANTS '20*.

[452] M. O. Rabin, "Probabilistic algorithms in finite fields," in *SIAM Journal on Computing '80*.

[453] C. F. Gauss, *Carl Friedrich Gauss' Untersuchungen uber hohere Arithmetik*. 1889.

[454] S. K. Chebolu and J. Mináč, "Counting irreducible polynomials over finite fields using the inclusion-exclusion principle," in *Mathematics Magazine '11*.

[455] C. Ding, T. Hesseseth, and W. Shan, "On the linear complexity of Legendre sequences," in *TIT '98*.

[456] V. Tóth, "Collision and avalanche effect in families of pseudorandom binary sequences," in *Periodica Mathematica Hungarica '07*.

[457] C. Mauduit and A. Sárközy, "On finite pseudorandom binary sequences I: Measure of pseudorandomness, the Legendre symbol," in *Acta Arithmetica '97*.

[458] "Legendre pseudo-random function," in `https://legendreprf.org/`.

[459] I. A. Seres, M. Horváth, and P. Burcsi, "The Legendre pseudorandom function as a multivariate quadratic cryptosystem: Security and applications," in *IACR ePrint 2021/182*.

[460] A. Dasgupta, R. Kumar, and T. Sarlos, "A sparse Johnson-Lindenstrauss transform," in *STOC '10*.

[461] D. M. Kane and J. Nelson, "A derandomized sparse Johnson-Lindenstrauss transform," *Arxiv 1006.3585*,

[462] H. Corrigan-Gibbs and D. Kogan, "The discrete-logarithm problem with preprocessing," in *EUROCRYPT '18*.

[463] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, `https://bitcoin.org/bitcoin.pdf`, 2008.

[464] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, "FlyClient: Super-light clients for cryptocurrencies," in *S&P '20*.

[465] P. Valiant, "Incrementally verifiable computation or proofs of knowledge imply time/space efficiency," in *TCC '08*.

[466] A. Kattis and J. Bonneau, "Proof of necessary work: Succinct state verification with fairness guarantees," in *IACR ePrint 2020/190*.

[467] I. Meckler and E. Shapiro, *Coda: Decentralized cryptocurrency at scale*, https://cdn.codaprotocol.com/v2/static/coda-whitepaper-05-10-2018-0.pdf, 2018.

[468] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro, "Coda: Decentralized cryptocurrency at scale," in *IACR ePrint 2020/352*.

[469] O(1) Labs, *Mina Cryptocurrency*, https://minaprotocol.com/.

[470] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: Preprocessing zkSNARKs with universal and updatable SRS," in *EUROCRYPT '20*.

[471] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge," in *IACR ePrint 2019/953*.

[472] S. Bowe, J. Grigg, and D. Hopwood, "Halo: Recursive proof composition without a trusted setup," in *IACR ePrint 2019/1021*.

[473] A. Chiesa, D. Ojha, and N. Spooner, "Fractal: Post-quantum and transparent recursive proofs from holography," in *EUROCRYPT '20*.

[474] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner, "Proof-carrying data from accumulation schemes," in *TCC '20*.

[475] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner, "Proof-carrying data without succinct arguments," in *CRYPTO '21*.

[476] I. Meckler, *Meet pickles SNARK*, https://medium.com/minaprotocol/meet-pickles-snark-enabling-smart-contract-on-coda-protocol-7ede3b54c250, 2020.

[477] A. Chiesa and E. Tromer, "Proof-carrying data and hearsay arguments from signature cards," in *ITCS '10*.

[478] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "Recursive composition and bootstrapping for SNARKs and proof-carrying data," in *STOC '13*.

[479] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from Bitcoin," in *S&P '14*.

[480] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," in *S&P '20*.

[481] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users," in *SEC '15*.

[482] D. Hopwood, *Scalable privacy*, https://www.youtube.com/watch?v=HNSf2Bw_YmM, Zcon, 2019.

[483] N. Tyagi, B. Fisch, J. Bonneau, and S. Tessaro, "Client-auditable verifiable registries," in *IACR ePrint 2021/627*.

[484] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza, "Secure sampling of public parameters for succinct zero knowledge proofs," in *S&P '15*.

[485] S. Bowe, A. Gabizon, and M. Green, “A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK,” in *IACR ePrint 2017/602*.

[486] S. Bowe, A. Gabizon, and I. Miers, “Scalable multi-party computation for zk-SNARK parameters in the random beacon model,” in *IACR ePrint 2017/1050*.

[487] J. Groth, “On the size of pairing-based non-interactive arguments,” in *EUROCRYPT ’16*.

[488] C. Badertscher, P. Gaži, A. Kiayias, A. Russell, and V. Zikas, “Ouroboros Genesis: Composable proof-of-stake blockchains with dynamic availability,” in *CCS ’18*.

[489] A. Ozdemir, R. S. Wahby, B. Whitehat, and D. Boneh, “Scaling verifiable computation using efficient set accumulators,” in *SEC ’20*.

[490] J. Lee, K. Nikitin, and S. Setty, “Replicated state machines without replicated execution,” in *S&P ’20*.

[491] R. Canetti, “Universally composable security: A new paradigm for cryptographic protocols,” in *FOCS ’01*.

[492] K. Wüst, S. Matetic, M. Schneider, I. Miers, K. Kostiainen, and S. Čapkun, “ZLiTE: Lightweight clients for shielded Zcash transactions using trusted execution,” in *FC ’19*.

[493] S. Matetic, K. Wüst, M. Schneider, K. Kostiainen, G. Karame, and S. Capkun, “BITE: Bitcoin lightweight client privacy using trusted execution,” in *SEC ’19*.

[494] D. V. Le, L. T. Hurtado, A. Ahmad, M. Minaei, B. Lee, and A. Kate, “A tale of two trees: One writes, and other reads: Optimized oblivious accesses to Bitcoin and other UTXO-based blockchains,” in *PETS ’20*.

[495] K. Qin, H. Hadass, A. Gervais, and J. Reardon, “Applying private information retrieval to lightweight Bitcoin clients,” in *CVCBT ’19*.

[496] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas, “Transparency logs via append-only authenticated dictionaries,” in *CCS ’19*, 2019.

[497] arkworks, *A Rust ecosystem for developing and programming with zkSNARKs*, `https://arkworks.rs`.

[498] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, *Zcash protocol specification*, `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf`, 2018.

[499] A. Naveh and E. Tromer, “PhotoProof: Cryptographic image authentication for any set of permissible transformations,” in *S&P ’16*.

[500] A. Chiesa, E. Tromer, and M. Virza, “Cluster computing in zero knowledge,” in *EUROCRYPT ’15*.

[501] S. Chong, E. Tromer, and J. A. Vaughan, “Enforcing language semantics using proof-carrying data,” in *IACR ePrint 2013/513*.

[502] E. Boyle, R. Cohen, and A. Goel, “Breaking the $O(\sqrt{n})$-bits barrier: Balanced Byzantine agreement with polylog bits per-party,” in *PODC ’21*.

[503] marlin, *A Rust library for the Marlin preprocessing zkSNARK*, `https://github.com/arkworks-rs/marlin`.

[504] A. Kate, G. M. Zaverucha, and I. Goldberg, “Constant-size commitments to polynomials and their applications,” in *ASIACRYPT ’10*.

[505] poly-commit, *A Rust library for polynomial commitments*, `https://github.com/arkworks-rs/poly-commit`.

[506] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno, “Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation,” in *S&P ’16*.

[507] A. E. Kosba, C. Papamanthou, and E. Shi, “xJsnark: A framework for efficient verifiable computation,” in *S&P ’18*.

[508] *Faster variable-base scalar multiplication in zk-SNARK circuits*, `https://github.com/zcash/zcash/issues/3924`.

[509] J. H. Cheon, “Security analysis of the strong Diffie–Hellman problem,” in *EUROCRYPT ’06*.

[510] J. Jaeger and S. Tessaro, “Expected-time cryptography: Generic techniques and applications to concrete soundness,” in *TCC ’20*.

[511] A. Gabizon, K. Gurkan, P. Jovanovic, G. Konstantopoulos, A. Oines, M. Olszewski, M. Straka, E. Tromer, and P. Vesely, *Plumo: Towards scalable interoperable blockchains using ultra light validation systems*, `https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-plumo_celolightclient.pdf`, 2020.

[512] P. Weißkirchner, *Evaluation and improvement of Ethereum light clients*, Technische Universität Wien, Diplomarbeit. `http://repositum.tuwien.ac.at/obvutwhs/content/titleinfo/4671631`, 2020.

[513] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, “DIZK: A distributed zero knowledge proof system,” in *SEC ’18*.

[514] A. Gluchowski, *World’s first practical hardware for zero-knowledge proofs acceleration*, `https://medium.com/matter-labs/worlds-first-practical-hardware-for-zero-knowledge-proofs-acceleration-72bf974f8d6e`, 2020.

[515] StarkWare, *Brining STARKs to Ethereum*, `https://www.starkdex.io/`.

[516] StarkWare, *When Lightning STARKs*, `https://medium.com/starkware/when-lightning-starks-a90819be37ba`.

[517] B. Whitehat, *Roll_up: Scale Ethereum with SNARKs*, `https://github.com/barryWhiteHat/roll_up`, 2018.

[518] Ethereum, *ZK-Rollups*, https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/.

[519] D. Leung, A. Suhl, Y. Gilad, and N. Zeldovich, "Vault: Fast bootstrapping for the Algorand cryptocurrency," in *NDSS '19*.

[520] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *SOSP '17*.

[521] G. Tankersley and J. Grigg, *Light client protocol for payment detection*, https://github.com/gtank/zips/blob/light_payment_detection/zip-XXX-light-payment-detection.rst, 2018.

[522] L. N. Lee, *Zcash reference wallet light client protocol*, https://www.electriccoin.co/blog/zcash-reference-wallet-light-client-protocol/, 2019.

[523] A. Kiayias, N. Lamprou, and A.-P. Stouka, "Proofs of proofs of work with sublinear complexity," in *FC '16*.

[524] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive proofs of proof-of-work," in *IACR ePrint 2017/963*.

[525] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "SoK: Consensus in the age of blockchains," in *AFT '19*.

[526] J. A. Garay and A. Kiayias, "SoK: A consensus taxonomy in the blockchain era," in *CT-RSA '20*.

[527] I. Eyal, A. E. Gencer, E. G. Sirer, and R. van Renesse, "Bitcoin-NG: A scalable blockchain protocol," in *NSDI '16*.

[528] Y. Sompolinsky and A. Zohar, "Accelerating Bitcoin's transaction processing: Fast money grows on trees, not chains," in *IACR ePrint 2013/881*.

[529] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "SPECTRE: A fast and scalable cryptocurrency protocol," in *IACR ePrint 2016/1159*.

[530] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *FOCS '95*.

[531] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instructions and software model for isolated execution," in *HASP '13*.

[532] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *HASP '13*.

[533] D. Fiore and A. Nitulescu, "On the (in)security of SNARKs in the presence of oracles," in *TCC '16-B*.

[534] *Ethereum*, https://ethereum.org/.

[535] *Zcash: Privacy-protecting digital currency*, https://z.cash/.

[536] *WhatsApp*, https://www.whatsapp.com/.

[537] *Signal*, `https://signal.org/`.

[538] *Telegram messenger*, `https://telegram.org`.

[539] *Line*, `https://www.line.me/`.

[540] *Keybase*, `https://keybase.io/`.

[541] *Algorand*, `https://www.algorand.com`.

[542] *Proton mail*, `https://protonmail.com/`.

[543] *Curv: The institutional standard for digital asset security*, `https://www.curv.co`.

[544] *Unbound tech: Secure cryptographic keys across any environment*, `https://www.unboundtech.com/`.

[545] *Bitgo*, `https://www.bitgo.com/`.

[546] *Sepior: Threshold cryptographic key management solutions with MPC*, `https://sepior.com`.

[547] *Keyless: Zero-trust passwordless authentication*, `https://keyless.io/`.

[548] A. Bagherzandi, S. Jarecki, N. Saxena, and Y. Lu, "Password-protected secret sharing," in *CCS '11*.

[549] M. Abdalla, M. Cornejo, A. Nitulescu, and D. Pointcheval, "Robust password-protected secret sharing," in *ESORICS '16*.

[550] P. D. MacKenzie, T. Shrimpton, and M. Jakobsson, "Threshold password-authenticated key exchange," in *CRYPTO '02*.

[551] M. D. Raimondo and R. Gennaro, "Provably secure threshold password-authenticated key exchange," in *EUROCRYPT '03*.

[552] A. Whitten and J. D. Tygar, "Why Johnny can't encrypt: A usability evaluation of PGP 5.0," in *SEC '99*.

[553] S. Ruoti, J. Andersen, D. Zappala, and K. E. Seamons, "Why Johnny still, still can't encrypt: Evaluating the usability of a modern PGP client," in *Arxiv 1510.08555*.

[554] C. S. Weir, G. Douglas, T. Richardson, and M. A. Jack, "Usable security: User preferences for authentication methods in eBanking and the effects of experience," in *Interacting with Computers '10*.

[555] C. Braz and J.-M. Robert, "Security and usability: The case of the user authentication methods," in *International Conference of the Association Francophone d'Interaction Homme-Machine '06*.

[556] W. Ford and B. S. Kaliski, "Server-assisted generation of a strong secret from a password," in *IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises '00*.

[557] *TOTP: Time-based one-time password algorithm*, `https://tools.ietf.org/html/rfc6238`.

[558] *OAuth*, `https://www.oauth.net/`.

[559] *Certificate transparency*, `https://www.certificate-transparency.org/`.

[560] *Key transparency*, `https://github.com/google/keytransparency`.

[561] W. Diffie and M. E. Hellman, "New directions in cryptography," in *TIT '76*.

[562] *The transport layer security (TLS) protocol version 1.3*, `https://tools.ietf.org/html/rfc8446`.

[563] *The illustrated TLS 1.3 connection*, `https://tls13.ulfheim.net/`.

[564] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *CRYPTO '10*.

[565] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *S&P '17*.

[566] F. Zhang, S. K. D. Maram, H. Malvai, S. Goldfeder, and A. Juels, "DECO: Liberating web data using decentralized oracles for TLS," in *CCS '20*.

[567] D. Abram, I. Damgård, P. Scholl, and S. Trieflinger, "Oblivious TLS via multi-party computation," in *CT-RSA '21*.

[568] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol candidates," in *CCS '15*.

[569] D. Rotaru and T. Wood, "MArBled circuits: Mixing arithmetic and boolean circuits with active security," in *INDOCRYPT '19*.

[570] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood, "Zaphod: Efficiently combining LSSS and garbled circuits in SCALE," in *WAHC '19*.

[571] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for MPC over mixed arithmetic-binary circuits," in *CRYPTO '20*.

[572] L. Wang, G. Asharov, R. Pass, T. Ristenpart, and a. shelat, "Blind certificate authorities," in *S&P ' 19*.

[573] *Sender policy framework (SPF) for authorizing use of domains in email*, `https://tools.ietf.org/html/rfc7208`.

[574] *DomainKeys identified mail (DKIM) signatures*, `https://tools.ietf.org/html/rfc6376`.

[575] *AT&T SMS API*, `https://developer.att.com/sms`.

[576] *Sprint enterprise messaging developer APIs*, `https://sem.sprint.com/developer-apis/`.

[577] *Verizon's enterprise messaging access gateway*, `https://ess.emag.vzw.com/emag/login`.

[578] *What is U2F?* `https://developers.yubico.com/U2F/`.

[579] *YubiKey strong two factor authentication*, `https://www.yubico.com/`.

[580] *Titan security key*, `https://cloud.google.com/titan-security-key`.

[581] *Why FIDO U2F was designed to protect your privacy*, `https://fidoalliance.org/fido-technotes-the-truth-about-attestation/`.

[582] M. Just and D. Aspinall, "Personal choice and challenge questions: A security and usability assessment," in *SOUPS '09*.

[583] S. E. Schechter, A. J. B. Brush, and S. Egelman, "It's no secret: Measuring the security and reliability of authentication via 'secret' questions," in *S&P '09*.

[584] A. Rabkin, "Personal knowledge questions for fallback authentication: Security questions in the era of facebook," in *SOUPS '08*.

[585] M. Toomim, X. Zhang, J. Fogarty, and J. A. Landay, "Access control by testing for shared knowledge," in *CHI '08*.

[586] *Mfa factor sequencing*, `https://help.okta.com/en/prod/Content/Topics/Security/mfa-factor-sequencing.htm`.

[587] R. Gennaro, S. Goldfeder, and A. Narayanan, "Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security," in *ACNS '16*.

[588] D. Boneh, R. Gennaro, and S. Goldfeder, "Using level-1 homomorphic encryption to improve threshold DSA signatures for Bitcoin wallet security," in *LATINCRYPT '17*.

[589] R. Gennaro and S. Goldfeder, "Fast multiparty threshold ECDSA with fast trustless setup," in *CCS '18*.

[590] J. Doerner, Y. Kondi, E. Lee, and A. Shelat, "Threshold ECDSA from ECDSA assumptions: The multiparty case," in *S&P '19*.

[591] R. Gennaro and S. Goldfeder, "One round threshold ECDSA with identifiable abort," in *IACR ePrint 2020/540*.

[592] *Bitcoin's multisignature*, `https://en.bitcoin.it/wiki/Multisignature`.

[593] *wolfSSL embedded SSL/TLS library — now supporting TLS 1.3*, `https://https://www.wolfssl.com/`.

[594] S. Kumar, D. Culler, and R. A. Popa, "Nearly zero-cost virtual memory for secure computation," in *OSDI '21*.

[595] R. Yahalom, B. Klein, and T. Beth, "Trust relationships in secure systems: A distributed authentication perspective," in *S&P '93*.

[596] T. Beth, M. Borcherding, and B. Klein, "Valuation of trust in open networks," in *ESORICS '94*.

[597] *OpenPGP message format*, `https://tools.ietf.org/html/rfc4880`.

[598] *Biglumber: Key signing coordination*, `http://www.biglumber.com/`.

[599] *IBM Verify Credentials: Transforming digital identity into decentralized identity*, `https://www.ibm.com/blockchain/solutions/identity`.

[600] *Blockstack*, https://www.blockstack.org/.

[601] *Civic wallet - digital wallet for money and cryptocurrency*, https://www.civic.com/.

[602] J. Bonneau, “EthIKS: Using Ethereum to audit a CONIKS key transparency log,” in *FC ’16*.

[603] S. Eskandarian, E. Messeri, J. Bonneau, and D. Boneh, “Certificate transparency with privacy,” in *PETS ’17*.

[604] F. Breuer, V. Goyal, and G. Malavolta, “Cryptocurrencies with security policies and two-factor authentication,” in *EuroS&P ’21*.

[605] *Partisia: Digital infrastructure with no single point of trust*, https://partisia.com/key-management/.

[606] C. Baum, E. Orsini, P. Scholl, and E. Soria-Vazquez, “Efficient constant-round MPC with identifiable abort and public verifiability,” in *CRYPTO ’20*.

[607] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven, “The wonderful world of global random oracles,” in *EUROCRYPT ’18*.

[608] R. Canetti, A. Jain, and A. Scafuro, “Practical UC security with a global random oracle,” in *CCS ’14*.

[609] H. K. Maji, M. Prabhakaran, and M. Rosulek, “Complexity of multi-party computation functionalities,” in *IACR ePrint 2013/042*.