Secure Computation Systems for Confidential Data Analysis

Rishabh Poddar

Electrical Engineering and Computer Sciences
University of California at Berkeley

December 17, 2020

Secure Computation Systems for Confidential Data Analysis

by

Rishabh Poddar

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 Ion Stoica
Professor Sylvia Ratnasamy
Professor Deirdre Mulligan

Fall 2020

Secure Computation Systems for Confidential Data Analysis

Abstract

Secure Computation Systems for Confidential Data Analysis

by

Rishabh Poddar

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Raluca Ada Popa, Chair

A large number of services today are built around processing data that is collected from or shared by customers. While such services are typically able to protect the data when it is in transit or in storage using standard encryption protocols, they are unable to extend this protection to the data *when it is being processed*, making it vulnerable to breaches. This not only threatens data confidentiality in existing services, it also prevents customers from availing such services altogether for sensitive workloads, in that they are unwilling / unable to share their data out of privacy concerns, regulatory hurdles, or business competition.

Existing solutions to this problem are unable to meet the requirements of advanced data analysis applications. Systems that are efficient do not provide strong enough security guarantees, and approaches with stronger security are often not efficient.

To address this problem, the work in this dissertation develops new systems and protocols for securely computing on encrypted data, that attempt to bridge the gap between security and efficiency. We distill design principles based on the properties of the two primary approaches for secure computation—advanced cryptographic protocols and trusted execution environments. Informed by these principles, we design novel cryptographic protocols and algorithms with strong and provable security guarantees, using which we show how to build systems that are both secure and efficient.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation is a result of the teaching, advice, inspiration, and support of a number of people, to whom I wish to express my sincerest gratitude.

Raluca Ada Popa, my advisor, will always remain one of the most brilliant people I have had the privilege of knowing and learning from. Raluca has been a tireless teacher, guide, and mentor. This dissertation is a testament to the time and energy she invested in my success.

I have benefited from the wisdom and scholarship of many, who have influenced my research outlook, vision, and work ethic in various ways: Ganesh Ananthanarayanan, Alessandro Chiesa, Joseph Gonzalez, Joseph Hellerstein, Deirdre Mulligan, Vern Paxson, Koushik Sen, Srinath Setty, Ion Stoica, Sylvia Ratnasamy, and Stavros Volos.

I am equally grateful for the mentorship of Niloy Ganguly and Debdeep Mukhopadhyay at IIT Kharagpur, as well as Vijay Mann and Mohan Dhawan at IBM Research, who played a formative role in preparing me for this journey.

I am thankful to the RISE Lab staff and administrators, who afforded me the privilege of not having to worry about anything except my research, and ensured that my experience was made as smooth as possible: Kattt Atchley, Shane Knapp, Jon Kuroda, Dave Schonenberg, and Boban Zarkovich.

The work in this dissertation would not have been possible without the support and contributions of my collaborators: Tobias Boelter, Ryan Deng, Sukrit Kalra, Andrew Law, Chang Lan, Chester Leung, Jianan Lu, Pratyush Mishra, Chenyu Shi, Octavian Sima, Jeongseok Son, Stephanie Wang, Avishay Yanai, David Yi, Chaofan Yu, Xingmeng Zhang, and Wenting Zheng. I'm also grateful to Jethro Beekman, Aurojit Panda, and Chia-Che Tsai for their contributions to open source, as well as advice and assistance.

I am fortunate to have had friends, companions, and colleagues who made this journey a joy: Tobias Boelter, Grant Ho, Frank Li, Nathan Malkin, Pratyush Mishra, Austin Murdock, Patrick O'Hara, Richard Shin, Yifan Wu, Wenting Zheng, and all the members of the RISE Lab at UC Berkeley.

Finally, I remain indebted to my parents and my family, for all their love. Their strength made this possible, and their faith made it worthwhile.

Chapter 1

Introduction

This dissertation shows how to build secure and efficient systems for analyzing confidential data, by developing novel cryptographic protocols and algorithms along with principles for system design.

1.1 Motivation

A large number of services and applications today are built around analyzing data that is collected from or shared by customers. At the same time, data breaches are becoming commonplace [Bek20, Bea18], and the public concern over data privacy is likely at one of its peaks today. To protect the confidentiality of user data, the current mode of operation is to encrypt the data when it is at rest (using a standard encryption scheme applied to the data storage) or when it is in transit (using protocols for secure communication such as TLS). However, there are no common deployments of solutions today that protect the data when *in use*—that is, in current deployments the data lies exposed, unencrypted, when it is being processed.

This state of affairs can lead to a complete violation of confidentiality, especially when the servers that process the data belong to entities that do not own the data. Specifically, there are two related scenarios where this concern arises.

The first scenario is that of *outsourced computation*. For instance, cloud-based services allow customers to avail the benefits of cloud computing by offloading their compute to the service. Examples include services for data analytics that process customer data to draw business insights, or services for network security that filter the network traffic of customer organizations. Beyond customer organizations, several services cater to end users as well. For example, modern home monitoring services (*e.g.*, [Kun]) transmit videos from users' homes to the service platform in the cloud, where the video streams are analyzed for suspicious activities.

The second scenario is that of *collaborative computation*. The value of data has also led many organizations to be increasingly interested in collaborating with each other towards a common aim. That is, multiple parties jointly analyze their collective data in order to draw mutually beneficial insights. For instance, such collaboration can lead to better medical studies [BEE$^+$17, KBV13]; it

can help in identification of criminal activities (*e.g.*, fraud and money laundering) [SvHA$^+$19]; it can also enable more robust financial services [SvHA$^+$19, BFLV12, AKL12, PNH17].

In both the above computation scenarios, entities that process the data either do not own the data, or own only a part of it. This is problematic because much of the collected / shared data is sensitive in nature, and exposes the data owners to the risks of breaches in privacy. In many cases, privacy concerns may preclude potential customers from using such services altogether, due to the fear of hackers, or lack of trust in the service's infrastructure and employees. Similarly, organizations may be unwilling / unable to collaborate and share their data with each other due to privacy concerns, regulations, or business competition.

This begets the question: how can we enable parties to analyze sensitive data in a way that *does not reveal the plaintext data*?

While researchers have developed various approaches for this task, there continues to exist a large gap between the capabilities of existing solutions and the demands of complex data analysis workloads. Approaches that are efficient do not provide strong enough security guarantees, and approaches with stronger security are often not efficient. A significant challenge lies in meeting both requirements simultaneously. Navigating the security-efficiency tradeoff space thus requires new design points. To address this problem, the work in this dissertation develops new systems and protocols for securely analyzing encrypted data, bridging the gap between security and efficiency.

1.2 Approaches for Secure Computation

Broadly, there are two primary techniques for secure computation. One approach is to use general-purpose cryptographic protocols that enable parties to directly process encrypted data without ever decrypting it, such as homomorphic encryption [Gen09] or secure multi-party computation [Yao82, GMW87, BGW88]. Cryptographic protocols offer precise and provable security guarantees against attackers. However, they typically also come with high overhead. This is especially true for protocols that are secure against malicious adversaries. Such protocols offer very strong security guarantees in that even if the adversary misbehaves and deviate from the protocol arbitrarily, the confidentiality of the data is not compromised. On the flipside, the performance overhead makes them prohibitively expensive in many cases. At the other end of the spectrum are semi-honest protocols. Usually, semi-honest protocols are significantly faster than maliciously secure protocols, but they also provide much weaker security guarantees—they assume that the adversary continues to follow the protocol faithfully, which may not be realistic in practice.

An alternate approach is to rely on the use of specialized hardware to create *trusted execution environments* (or "enclaves") within which sensitive data can be securely loaded and processed, such as Intel SGX [MAB$^+$13]. The root of trust for the enclaves is the hardware manufacturer. Hardware enclaves are faster than purely cryptographic approaches, but they come with their own set of challenges. First, fully exploiting the potential of enclaves requires careful system design. While researchers have proposed solutions for executing arbitrary applications in enclaves [BPH14, TPV17], these solutions come with security and performance downsides. Second, enclaves

	Cryptographic execution		Trusted execution
Outsourced computation	Arx [PBP19]	Oblix [MPC+18]	SafeBricks [PLPR18]
			Visor [PAS+20]
Collaborative computation	Senate [PKY+21]		Secure XGBoost [LLP+20]

Figure 1.1: Classification of the systems we built, by computation scenario and the secure computation approach used by the system.

also provide weaker security guarantees, in that they are known to be vulnerable to side-channel attacks. General-purpose techniques for mitigating such attacks induce high overhead.

We provide a detailed discussion of both approaches in Chapter 2. Both approaches have their own limitations, which in practice often makes them unsuitable for complex data analysis on highly sensitive data if used "out of the box." Overall, given the limitations of the general-purpose approaches described above, building systems that are both secure and efficient is challenging and requires care.

1.3 Building Systems using Secure Computation

Our approach is to *co-design* the systems and their underlying algorithms / protocols. The system's requirements govern the design of the protocols, and the protocols in turn impact the design of the system. To guide the development of secure and efficient systems, we distill a design strategy based on the properties of the different approaches for secure computation. Our strategy is effective, improving performance by several orders of magnitude compared to alternate approaches while still providing strong security guarantees.

We illustrate our design strategy by applying it towards system design for several complex workloads. Chapter 2 describes our approach in detail; here, we provide an overview of the systems we built by employing our strategy. Figure 1.1 summarizes these systems, and classifies them by the secure computation approach we employed in designing the systems as well as the computation scenario they enable.

Designing systems based on cryptographic protocols. We designed and implemented Arx

[PBP19], a practical and functionally rich database system that *always* keeps the data encrypted with semantically secure encryption schemes. Semantic security implies that no information about the data is leaked other than its size. Arx introduces new database indices built atop novel and efficient cryptographic protocols; these indices can be used to support a wide range of database operations such as keyword search, range queries, aggregations, and joins. Arx develops and carefully synthesizes a range of novel cryptographic protocols into a secure and efficient system.

While Arx supports general database operations, we also built Oblix [MPC$^+$18] that optimizes for the particular case of search operations. Specifically, Oblix enables a search index for encrypted data that is dynamic (*i.e.*, supports inserts and deletes), and has good efficiency. Oblix strengthens the security guarantees offered by Arx by additionally hiding the *memory access patterns* of query execution by building upon that is oblivious (provably hides access patterns), is dynamic (supports inserts and deletes), and has good efficiency.

Both Arx and Oblix target outsourced computation. In contrast, Senate [PKY$^+$21] is a platform for secure collaborative analytics. At the heart of Senate is a new and efficient cryptographic protocol for multi-party computation that provides the strong guarantee of *malicious security*. That is, even if $m - 1$ out of m parties fully misbehave and collude, an honest party is guaranteed that nothing leaks about their data other than the result of the agreed upon query. Senate employs a synergy of new cryptographic design and insights in query rewriting and planning. Its query planner efficiently plans the cryptographic execution, improving performance by over two orders of magnitude compared to the state-of-the-art.

High-performance systems using trusted execution. Though the above systems help support complex workloads in a secure and efficient manner, they still have relatively high overhead compared to their plaintext counterparts. This overhead is hard to eradicate completely using purely cryptographic approaches. To meet the demands of workloads with stricter performance requirements, we turn to systems that offset the cost of cryptographic protocols with the help of trusted execution environments (or enclaves). For instance, Oblix offloads a portion of its security-critical computation to an enclave at the cloud server, allowing it to scale to large datasets with millions of records.

However, architecting systems based on enclaves are accompanied by their own set of design challenges, and require principled system design. We illustrate these design principles using several systems that target workloads with high performance requirements. SafeBricks [PLPR18] shields the analysis of network traffic from an untrusted cloud provider, by providing an enclave-based framework for executing network functions.

We generalize our techniques to systems for machine learning (ML) as well. Visor [PAS$^+$20] provides a framework for ML workloads by unifying CPU and GPU enclaves into a single trust domain (which we refer to as a *hybrid* enclave). Within this framework, we develop an application for video analytics. We also extend enclaves to scenarios for collaborative computation in the Secure XGBoost framework [LLP$^+$20], that enables multiple parties to collaboratively train gradient boosted decision tree ensembles on their collective data.

Mitigating side-channel leakage in trusted execution. Though enclaves go a long way towards protecting the confidentiality of data, researchers have shown them to be vulnerable to side-channel

attacks. In particular, memory access pattern leakage remains the fundamental reason behind a large class of known attacks on enclaves—even though attackers cannot directly observe the data protected by the enclave, they can still monitor the enclave's memory access patterns during execution, and infer sensitive information about the data.

To prevent this leakage, we design our systems in a way such that the memory access patterns of the enclave code does not reveal any information about sensitive data, *i.e.*, we make the enclave's execution *data-oblivious*. Naïve approaches for achieving data-obliviousness can lead to slowdowns of several orders of magnitude. Instead, we carefully craft new algorithms that do not leak access patterns *by design*. In particular, we develop novel and efficient data-oblivious algorithms for video analytics in Visor; and design an oblivious training algorithm in Secure XGBoost.

1.4 Impact and Adoption

To help disseminate the ideas and systems contained in this dissertation, we have also developed the MC^2 platform which is available as open-source software [MC2]. MC^2 is a larger effort that integrates privacy-preserving platforms for data analytics and machine learning into a unified framework. MC^2 combines solutions based on hardware enclaves (with data-oblivious protocols for side-channel attack protection) and advanced cryptographic protocols such as secure multi-party computation, including systems developed as part of this dissertation.

MC^2 has already had impact, and we are collaborating with several teams in industry who have adopted it for their own use cases. A few notable examples are as follows: Ericsson used MC^2 as a proof-of-concept to securely train models on data belonging to several network operators for applications in telecommunicataions [PJ20]; Scotiabank has been spearheading an effort with other Canadian banks using MC^2 towards anti-money laundering; and, Ant Financial deployed MC^2 internally in production for credit loan risk modeling.

1.5 Dissertation Roadmap

In Chapter 2 we provide an overview of the approaches for secure computation on encrypted data. We discuss the design challenges that come with the different approaches, and develop a strategy for building systems that overcome these challenges. In later sections, we employ and extend these strategies to develop systems that are both secure and efficient.

The next two chapters focus on tackling query execution in databases via cryptographic approaches for secure computation. Chapter 3 presents the design of Arx, and Chapter 4 presents Senate. Subsequently, we turn to designing systems that have stricter performance requirements, with the help of trusted execution environments. In Chapter 5 we present SafeBricks, in Chapter 6 we present Visor, and in Chapter 7 we describe Secure XGBoost. Finally, we conclude in Chapter 8 and discuss future directions.

Chapter 2

Building Secure and Practical Data Systems

Broadly, there are two major techniques for secure computation. One approach is to rely on the use of specialized hardware to create *trusted execution environments* (or "enclaves") within which sensitive data can be securely loaded and processed. The other approach is to use cryptographic protocols that enable parties to directly process encrypted data, without ever decrypting it. In this chapter, we provide an overview of both techniques and discuss the challenges that come with building systems using the techniques. We then outline high-level strategies for designing systems that are both secure and efficient.

2.1   Trusted Execution Environments

Trusted execution environments (TEEs), or enclaves,[1] are a recent advancement in computer processor technology designed to protect an application's code and data from all other software in the system. This approach is based on the assumption that users are willing to trust some specialized hardware that is deployed in an otherwise untrusted environment, such as a cloud datacenter. The hardware in turn gives the users the ability to create a trusted execution environment (or secure "enclave") within which they can put their most sensitive pieces of code and data. The enclaves are isolated from the rest of the platform, including privileged software such as the operating system and the hypervisor. As a result, the enclave contents remain protected from server administrators as well as attackers with privileged access.

To bootstrap the enclaves, the hardware provides a mechanism called *remote attestation* which allows cloud tenants to verify that they are communicating with their own code hosted within a secure enclave. In more detail, this procedure allows a remote client system to cryptographically verify that specific software has been securely loaded into an enclave [AGJS13]. At a high level, the processor computes a hash measurement of the enclave once it is initialized with the protected software, and stores the measurement in a special register. When a client requests remote attestation, the enclave generates a report signed by the processor that contains a hash measurement of the enclave. It then returns the report to the client which can then verify the hash and the signature. As

---

[1]In this dissertation, we will use the terms TEEs and enclaves interchangeably.

part of the attestation, the enclave can also bootstrap a secure channel with the client by generating a public key and returning it with the signed report.

Attacks and vulnerabilities.   Unfortunately, existing enclave implementations, such as Intel SGX, are known to be vulnerable to a host of side-channel attacks. Such attacks commonly exploit micro-architectural side-channels software-based channels, or application-specific leakage such as network and memory accesses.

A large subset of known side-channel attacks on enclaves fundamentally rely on the exploitation of *data-dependent memory access patterns—i.e.*, the sequence of memory addresses accessed during the application's execution. Examples of such attacks include cache attacks [GESM17, BMD+17, SWG+17, MIE17, HCP17], attacks based on branch prediction [LSG+17], paging-based attacks [XCP15, BWK+17], or memory bus snooping [LJF+20]. The impact of access pattern leakage can be dire; for instance, Xu *et al.* [XCP15] showed that by simply observing the page access patterns of image decoders, an attacker can reconstruct entire images, even though the image itself remains protected within the enclave for the duration of the application's execution. In Section 6.2.3, we also illustrate the impact of access pattern leakage on video analytics pipelines, and show that by observing just the memory access patterns of commonly used video processing modules, an attacker can infer the exact shapes and positions of all moving objects in the video. Memory access pattern leakage is typically considered to be outside the threat model of existing enclaves. For example, Intel considers its prevention to be a matter for the developer of the enclave application, *e.g.*, via side-channel resistant program development techniques [Intc, Intb].

Transient execution attacks form another large class of attacks on enclaves (*e.g.*, [BMW+18, SLM+19, CCX+19, VBMS+20, RMR+21, vSMO+19, vSMK+20]). These attacks exploit microarchitectural optimizations in modern CPUs that predict and occasionally reorder the instruction stream to boost performance (*e.g.*, speculative execution of instructions). However, transient instructions can potentially leave behind secret-dependent traces in the microarchitectural state of the CPU, allowing them to be recovered by an attacker. Unlike attacks based on access pattern leakage, transient execution attacks typically rely solely on the behavior of the CPU, and do not exploit software vulnerabilities or application behavior. In fact, researchers have demonstrated how these attacks can be used to extract the enclave platform's attestation keys. As such, these vulnerabilities are usually patched promptly by enclave vendors via microcode updates when they come to light.

Enclaves are also vulnerable to attacks that exploit timing analysis or power consumption [MOG+20, TSS17], DoS attacks [JLLK17, GLS+17b], and rollback attacks [PLD+11].

2.2   Cryptographic Approaches

We summarize cryptographic protocols that are relevant to the techniques and systems developed in this dissertation.

Homomorphic encryption.   Homomorphic encryption schemes [Gen09] allow a data owner to encrypt their secret data d using a public key, and hand the encrypted data $\mathsf{Enc}(d)$ over to an untrusted party for running computations on it. Using the public key, the untrusted party can compute a function f directly on $\mathsf{Enc}(d)$ to obtain an encrypted output $\mathsf{Enc}(f(d))$. The data owner

can decrypt the output to get the final result $f(d)$. The protocol ensures that the untrusted party learns no information about the secret data or the result, other than their lengths.

Homomorphic encryption protocols can either be *fully* homomorphic, or *partially* homomorphic. Fully homomorphic protocols can compute arbitrary functions f, while partially homomorphic protocols are restricted in the choice of f; for example, the partially homomorphic Paillier scheme [Pai99] can only compute additions of ciphertexts.

Secure multi-party computation.   Secure multi-party computation [Yao82, GMW87, BGW88] (MPC) is a cryptographic technique that allows m parties, each having secret data d_i, to jointly compute a function f on their aggregate data, and to share the result $f(d_1, \ldots, d_m)$ amongst themselves, *without* learning each other's data beyond what the function's result reveals. In essence, MPC protocols enable the parties to cryptographically *emulate* a setting where all the secret data is provided to a trusted third party that applies the function f on the data, and outputs the results.

General-purpose protocols for MPC typically fall into one of two categories: arithmetic MPC [BGW88, GMW87], and Boolean MPC [Yao82, BMR90]. In the former, the function f is represented as a circuit of addition and multiplication gates over finite field elements, while in the latter f is represented as a circuit of XOR and AND gates that process Boolean values.

Garbled circuits.   Garbled circuits [Yao86, BHR12, BMR90, GMW87] are a commonly used cryptographic primitive in MPC constructions. Formally, an m-party garbling scheme is a set of algorithms (Garble, Encode, Eval, Decode) that enables the secure evaluation of a (typically Boolean) circuit C. Using the garbling scheme, the parties can invoke the algorithm Garble on the circuit C to obtain a *garbled* version of the circuit $\mathsf{G}(C)$, along with some secret encoding information e, and decoding information d. Given input x, the parties can run $\mathsf{Encode}(e, x)$ to produce a *garbled input* X. Then, the parties run $\mathsf{Eval}(\mathsf{G}(C), X)$ on the garbled input to obtain a garbled output Y. Finally, the parties decode the garbled output $\mathsf{Decode}(d, Y)$ to obtain the actual output y. The correctness of the scheme ensures that $y = C(x)$. In addition, the security of garbled circuits guarantees that the parties learn *nothing* about x other than the output $C(x)$ (along with size information).

2.3   Challenges and Design Strategy

The techniques described in the previous sections are valuable tools for designing end-to-end systems that can compute on data securely, without revealing information to any untrusted parties. However, a significant challenge lies in ensuring that the system is able to simultaneously meet the security and performance requirements of the application. These requirements dictate the design decisions and tradeoffs involved in building the system. For example, a video analytics application must be able to keep up with the incoming frame rate of the video stream to a reasonable degree in order to be useful. On the other hand, real-time performance may be less important for collaborative query execution and analytics. In such cases, the security properties afforded by cryptographic approaches might potentially outweigh performance concerns.

Cryptographic execution.   While cryptographic protocols provide precise and provable security guarantees, they typically incur high overhead on the order of several orders of magnitude compared

to plaintext computation. The overhead impacts both performance as well as resource consumption (*e.g.*, memory and network bandwidth). This often makes the black-box usage of general-purpose schemes such as MPC or homomorphic encryption infeasible for implementing an end-to-end system, especially for complex and latency-sensitive workloads.

Instead, we improve performance by developing novel protocols that are informed by the nature of the required computation. By tailoring the protocols to the application at hand, we are able to significantly outperform their general-purpose counterparts, as illustrated by our systems Arx and Senate (Chapter 3 and Chapter 4, respectively). In devising the protocols, we employ the following high-level strategy:

- First, we break the computation down into smaller modules, and design a secure solution for the modules by employing the most suitable cryptographic protocol for that task. For example, in Arx, we modularize a database into indices for separate operations (*e.g.*, keyword search, range queries, joins, and aggregates), and devise efficient protocols for each index separately. In Senate, we decompose queries into a tree of sub-operations, and cryptographically execute each sub-operation separately.

- Second, we cryptographically stitch the individual modules together into the larger system. This is challenging, because one needs to ensure that the overall security guarantees of the system continue to be met. This requirement in turn dictates the suitability of the protocols used for the different modules. As an example, both Arx and Senate devise protocols to *solder* different garbled circuits together into a larger whole.

- Third, we develop protocols and strategies to reduce the computation that occurs on encrypted data as much as possible (*e.g.*, by strategically offloading lightweight but crucial pieces of computation to the client). This needs to be done with care—we need to ensure that the overall security guarantees remain unaffected, and that the client is not unduly burdened. This strategy helps both Arx and Senate improve performance by up to an order of magnitude.

Trusted execution environments. For systems with stricter performance requirements, we turn to the use of enclaves. Unlike cryptographic approaches, enclaves can harness the raw power of the CPU on plaintext computation, as the encrypted data is decrypted by the hardware once it is loaded into the processor package. However, designing systems based on enclaves are accompanied by their own set of design challenges.

- First, minimizing the amount of code that runs within the enclave, *i.e.*, the trusted computing base (or TCB), requires careful partitioning of the target application, and choosing a boundary that reduces the code without compromising security. At the same time, partitioning the application is likely to result in transitions between enclave and non-enclave code. These transitions are expensive, introducing a high run-time overhead due to the cost of saving and restoring the state of the secure environment. Consequently, there is a tension between TCB size and the overall performance of the application: the lesser code the enclave contains, the more transitions it is likely to make to non-enclave code.

- Second, enclaves can be limited by the amount of memory at their disposal. For example, the current generation of Intel SGX enclaves currently only have 93.5 MB enclave memory (though this is likely to increase in the future). System designers need to be cognizant of any memory constraints to prevent them from becoming a performance bottleneck.

- Third, as discussed above, enclaves are vulnerable to side-channel attacks, such as those based on access pattern leakage, the prevention of which is delegated to the application developer. An effective strategy for removing access pattern leakage is to ensure that the enclave application's *execution* is *data-oblivious*, a property that guarantees that the access patterns of the application remain independent of secret data. However, general-purpose approaches for executing the application obliviously (*e.g.*, [RLT15, AJX$^+$19]) can slow it down by several orders of magnitude. Instead, as we show, implementing the enclave application to be algorithmically free of access pattern leakage can significantly boost performance.

While researchers have proposed generic solutions for securely executing arbitrary applications in enclaves [TPV17, BPH14], these solutions are unable to account for the above requirements as a result of their generality, resulting in both security and performance downsides. We illustrate how we overcome these challenges in the design of SafeBricks (Chapter 5), Visor (Chapter 6), and Secure XGBoost (Chapter 7). We carefully partition the target applications to optimize the amount of code that runs within the enclave, while simultaneously balancing the frequency of enclave transitions. We ease the enclaves' memory burden by streaming data through the enclaves to enable memory reuse. Finally, we carefully devise new algorithms for the enclave code so as to make it free of memory access pattern by design.

Chapter 3

Database Queries on Encrypted Data

This chapter presents Arx, a practical and functionally rich database system that *always* keeps the data encrypted with semantically secure encryption schemes, enabling clients to outsource database management to untrusted servers. Arx combines novel cryptographic protocols with insights in query planning and execution.

3.1 Introduction

In recent years, encrypted databases [PRZB11, AEK$^+$13, TKMZ13, PBC$^+$16] (EDBs) have emerged as a promising direction towards achieving both confidentiality and functionality for processing sensitive data—the database queries are run directly on encrypted data. CryptDB [PRZB11] demonstrated that such an approach can be practical and can support a rich set of queries; it then spurred a rich line of work including Cipherbase [AEK$^+$13] and Monomi [TKMZ13]. The demand for such systems is demonstrated by the adoption in industry such as in Microsoft's SQL Server [Micf], Google's Encrypted Big Query [Gooa], and SAP's SEEED [GHH$^+$14] amongst others [Sky, Cipa, KGM$^+$14, iQr]. Most of these services are *NoSQL databases* of various kinds showing that a certain class of encrypted computation suffices for many applications.

Unfortunately, this area faces a challenging privacy-efficiency tradeoff, with no known practical system that does not leak information. The leakage is of two types: leakage from data and leakage from queries.

Leakage from data is leakage from an encrypted database, *e.g.*, *relations among data items*. In order to execute queries efficiently, the EDBs above use a set of encryption schemes some of which are *property-preserving* by design (denoted PPE schemes), *e.g.*, order-preserving encryption (OPE) [BCLO09, BCO11, PLZ13] or deterministic encryption (DET). OPE and DET are designed to reveal the *order* and the *equality relation* between data items, respectively, to enable fast order and equality operations. However, while these PPE schemes confer protection in some specific settings, a series of recent attacks [DDC16, GSB$^+$16, NKW15] have shown that given certain auxiliary information, an attacker can extract significant sensitive information from the order and equality relations revealed by these schemes. These works demonstrate *offline* attacks in which the attacker

obtains a PPE-encrypted database and analyzes it offline. For example, such attackers include hackers stealing a snapshot of the database, or government subpoenas.

Leakage from queries refers to what an (*online*) attacker can see during query execution. This includes all observable state in memory, along with which parts of the database are touched (called access patterns), including *which (encrypted) rows* are returned and *how many*, which could be exploited in certain settings [KKNO16, CGPR15, LMP18, GLMP19]. Unfortunately, hiding the leakage due to queries is very expensive as it requires oblivious protocols (*e.g.*, ORAM [SvDS+13]) to hide access patterns, along with aggressive padding [Nav15] to hide the result size. For instance, Naveed [Nav15] shows that in some cases it is more efficient to stream the database to the client and answer queries locally than to run such a system on a server.

A natural question is then: how can we protect a database from offline attackers as well as make progress against online attackers, while *still providing rich functionality and good performance?*

This chapter presents Arx, a practical and functionally rich database system that takes an important step in this direction by *always* keeping the data encrypted with semantically secure encryption schemes. Semantic security implies that no information about the data is leaked (other than its size and layout), preventing the aforementioned offline attacks on a stolen database. This model is particularly suitable for protecting data against subpoenas, in which case there is only leakage from data, and no leakage from queries.

For an online attacker, Arx incurs pay-as-you-go information leakage: the attacker no longer learns the frequency count or order relations for *every* value in the database, but *only for data involved in the queries it can observe*. In the worst case (*e.g.*, if the attacker observes many queries over time), this leakage could add up to the leakage of a PPE-based EDB, but in practice it may be significantly more secure for short-lived online attackers. As prior work points out [LW16, BLR+14, CLWW16], this model fits the "well-intentioned cloud provider" which uses effective intrusion-detection systems to prevent attackers from observing and logging queries over time, but fears "steal-and-run" attacks. For example, Microsoft's Always Encrypted [Micf] advocates this model.

Unfortunately, there is little work on such EDBs, with most work focusing on PPE-based EDBs. The closest to our goal is the line of work by Cash *et al.* [CJJ+14, CJJ+13] and Faber *et al.* [FJK+15], which builds on searchable encryption. As a result, these schemes are significantly limited in functionality—they do not support common queries such as order-by-limit, aggregates over ranges, or joins—and are also inefficient for write operations (*e.g.*, updates, deletes). Furthermore, for certain *online* attackers, these systems have some extra leakage not present in PPEs, as we elaborate in Section 3.12. To replace PPE-based EDBs, we need a solution that is always at least as secure as PPE-based EDBs.

Overall, by exclusively using semantically secure encryption, Arx prevents the offline attacks above [DDC16, GSB+16, NKW15] from which PPE-based EDBs suffer. For online attackers, Arx is *always* either more or as secure as PPE-based EDBs.

3.1.1 Techniques and contributions

A simple attempt to protect against offline attacks could be to keep the data encrypted at rest in the database and only decrypt it when it is in use. However, such an approach directly leaks the secret

key even to a short-lived attacker who succeeds in taking a well-timed snapshot of memory. Worse, the key itself would be directly vulnerable to a subpoena, akin to not encrypting the database at all. A better alternative might be to consider a hybrid design that uses a PPE-based EDB, but employs a second layer of encryption for data at rest on the disk. However, an attacker who similarly obtains the decryption key gets access to the PPE-encrypted data, rendering the second layer of encryption useless and leaking more information than our goal in Arx.

Instead, Arx introduces two new database indices, ArxRange and ArxEq that encrypt the data with semantic security; queries on these indices reveal only a limited per-query access pattern. ArxRange is for range and order-by-limit queries, and ArxEq is for equality queries. While ArxRange can be used for equality queries as well, ArxEq is substantially faster.

To enable range queries, ArxRange builds a tree over the relevant keywords, and stores at each node in the tree a *garbled circuit* for comparing the query against the keyword in the node [Yao86, GMW87]. Our tree is history-independent [AS89] to reduce structural leakage. The main challenge with ArxRange is to avoid interaction (*e.g.*, as needed in BlindSeer [PKV$^+$14]) at every node on a tree path. To address this challenge, Arx draws inspiration from the theoretical literature on Garbled RAM [GLO15]. Arx chains the garbled circuits on a tree in such a way that, when traversing the tree, a garbled circuit produces input labels for the child circuit to be traversed next. Thereby, the whole tree can be traversed in a single round of interaction. For security, each such index node may only be used once, so ArxRange essentially *destroys itself for the sake of security*. Nevertheless, only a logarithmic number of nodes are destroyed per query, and Arx provides an efficient repair procedure.

ArxEq builds a regular database index over encrypted values by embedding a counter into repeated values. This ensures that the encryption of two equal values is different and the server does not learn frequency information. To search for a value *v*, the client provides a small token to the server, which the server expands into many search tokens for all occurrences of *v*. ArxEq provides forward privacy [Bos16], preventing old tokens from being used to search new data.

Building on top of ArxRange, Arx speeds up aggregations by transforming them into tree lookups via ArxAgg.

Because of the new indices, index and query planning become challenging in Arx. The application's administrator specifies a set of regular indices, thereby expecting a certain asymptotic performance. However, regular indices do not directly map to Arx's indices because Arx's indices pose new constraints. The main constraints are: Arx cannot use the same index for both $=$ and $\geq$ operations, an equality index on (a, b) cannot be used to compute equality on *a* alone, and range queries requires an ArxRange index. With this in mind, we designed an index planning algorithm that guarantees the expected asymptotic performance while building few additional indices.

Finally, we designed Arx's architecture so that it is amenable to adoption. Two lessons [Pop14] greatly facilitated the adoption of the CryptDB system: do not change the DB server and do not change applications. Arx's architecture, presented in Figure 3.1, accomplishes these goals. The difference over the CryptDB architecture [PRZB11] is that it has a server-side proxy, a frontend for the DB server. The server proxy converts encrypted processing into regular queries to the DB, allowing the DB server to remain unchanged.

We implement and evaluate Arx on top of MongoDB, a popular NoSQL database. We show that
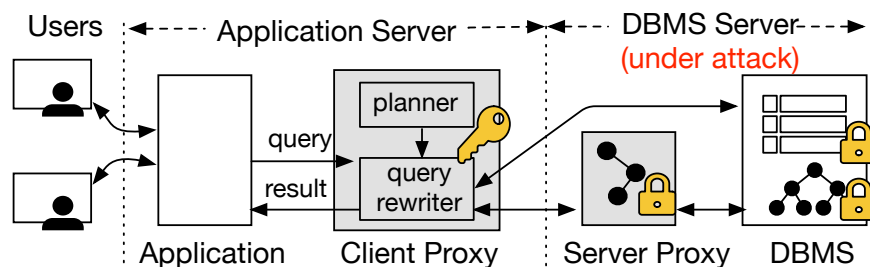
Figure 3.1: Arx's architecture: Shaded boxes depict components introduced by Arx. Locks indicate that sensitive data at the component is encrypted.

Arx supports a wide range of real applications such as ShareLaTeX [Sha], the Chino health data platform [Chi], NodeBB forum [Nod], and Leanote [Lea] amongst others. In particular, Chino is a cloud-based platform that serves the European medical project UNCAP [UNC]. Chino provides a MongoDB-like interface to medical web applications (running on hospital premises) but currently operates on *plaintext* data. The project's leaders confirmed that Arx's model fits Chino's setup perfectly. Finally, we also show that Arx's overheads are modest: it impacts the performance of ShareLaTeX by 11% and the YCSB [CST$^+$11] benchmark by 3–9%.

3.2 Overview

In the rest of this chapter, we use MongoDB/NoSQL terminology such as collections (for RDBMS tables), documents (for rows), and fields (for columns), but we use SQL format for queries because we find MongoDB's JS format harder to read. While we implement Arx for MongoDB, its design applies to other databases as well.

3.2.1 Architecture

Arx considers the model of an application that stores sensitive data at a database (DB) server. The DB server can be hosted on a private or public cloud. Figure 3.1 shows Arx's architecture. The application and the database system remain unmodified. Instead, Arx introduces two components between the application and the DB server: a trusted client proxy and an untrusted server proxy. The client proxy exports the same API as the DB server to the application so the application does not need to be modified. The server proxy interacts with the DB server by invoking its unmodified API (*e.g.*, issuing queries); in other words, the server proxy behaves as a regular client of the DB server. Unlike CryptDB, Arx cannot use user-defined functions instead of the server proxy because the proxy must interact with the DB server multiple times per client query.

The client proxy stores the master key. It rewrites queries, encrypts data, and forwards the rewritten queries to the server proxy for execution along with helper cryptographic tokens. It forwards all queries without any sensitive fields directly to the DB server. The client proxy is *lightweight*: it does not store the DB and does much less work than the server. The client proxy

stores metadata (schema information), a small amount of state, and optionally a cache. The server runs the expensive part of DB queries, filtering and aggregating many documents into a small result set.

In most cases, the client proxy processes only the results of queries (*e.g.*, to decrypt them). However, in some corner cases, it performs some post-processing; as a result, our implementation needs to duplicate some parts of the typesystem and expression evaluation logic of the server database.

3.2.2 Threat Model

Arx targets attackers to the database server. Hence, our threat model assumes that the attacker does not control or observe the data or execution on the client-side, including not issuing queries through the client proxy, and may only access the server-side which consists of Arx's server proxy and the database servers.

Arx considers *passive* (honest-but-curious) server attackers: the attackers examine server-side data to glean sensitive information, but follow the protocol as specified, and do not modify the database or query results. The active attacker is interesting future work, that can potentially leverage complementary techniques [ZKP15, KFPC16, Mer79, LHKR10]. Further, in the Arx model, an attacker cannot inject any *new* queries as she does not have access to the client application or to the secret keys at the client proxy, but only to the server.

We consider two types of passive attackers, offline and online attackers, and provide different guarantees for each. The *offline* attacker manages to steal *one* copy of the database, consisting of (encrypted) collections and indices. It does not contain in-memory data related to the execution of current queries (which falls under the online attacker). The *online* attacker is a generic passive attacker: it can log and observe any information available at the server (*i.e.*, all changes to the database, all in-memory state, and all queries) at any point in time for any amount of time.

3.2.3 Security guarantees

Arx has different guarantees for the two attackers.

Offline attacker. Arx's most visible contribution over PPE-based EDBs is for the offline attacker. Such an attacker corresponds to a wide range of real-world instances including hackers who extract a dump of the database, or insiders who managed to steal a copy of the database.

For this attacker, Arx provides strong security guarantees revealing nothing about the data beyond the schema and size information (the number of collections, documents, items per field, size of items, which fields have indices and for what operations). Padding is a standard procedure for hiding sizes at a performance cost. The contents of the database (collections and indices) are protected with *semantically secure* encryption, and the decryption key is never sent to the server. In particular, Arx prevents the offline attacks of [DDC16, GSB$^+$16, NKW15] from which PPE-based EDBs suffer. In PPE-based EDBs, the attacker readily sees the order or the frequency of all the values in the database for PPE-encrypted fields. This is significantly less secure than the semantic secure schemes in Arx, in which the attacker does not see such relations.

Online attacker. The online attacker additionally watches *queries* and their execution. Arx hides the parameters in the queries, but not the operations performed. In particular, Arx does not hide metadata (*e.g.*, query identifiers and timestamps) or access patterns during execution (*e.g.*, which positions in the database or index are accessed/returned and how many). Prior work has shown that, if an attacker can observe such information from many queries and if certain assumptions and conditions hold, the attacker can reconstruct data [KKNO16, CGPR15, LMP18, GLMP19]. Since each query in Arx reveals only a limited amount of metadata, the sooner an attacker is detected (*e.g.*, the fewer queries they observe), the less information they are able to glean.

For this attacker, Arx aims to always be more or as secure as PPE-based EDBs. Indeed, for all operations, Arx's leakage is always upper-bounded by the leakage in PPE-based EDBs. This is non trivial: for example, a prior EDB aiming for semantic security [FJK$^+$15] is not always more secure than PPE-based EDBs, as we explain in Section 3.12.

Security definition. To quantify the leakage to online attackers, we provide a security definition for Arx and its protocols that is parameterized by a *leakage profile* $\mathcal{L}$, which is a function of the database and the sequence of the queries issued by the client. Our security definition is fairly standard, and similar to prior work [FJK$^+$15, CJJ$^+$13].

We say that a DB system (or a query execution protocol) is $\mathcal{L}$-semantically secure if, for any PPT adversary $\mathcal{A}$, the entirety of $\mathcal{A}$'s view of the execution of the queries is efficiently *simulatable* given only $\mathcal{L}$. $\mathcal{A}$ invokes the interface exposed by the client to submit any sequence of queries Q. $\mathcal{A}$ then observes the execution of the queries from the perspective of the server, *i.e.*, it can observe all the state at the server, as well as the full transcript of communication between the client and server. Formally, $\mathcal{A}$'s task is to distinguish between a real world execution of the queries (**Real**) between the client and server, and an ideal world execution (**Ideal**) where the transcript is generated by a PPT simulator S that is only given access to the leakage function $\mathcal{L}$.

Definition 1. *Let $\mathcal{L}$ be a leakage function. We say that a protocol Π is $\mathcal{L}$-semantically-secure if for all adversaries $\mathcal{A}$ and for all sequences of queries Q, there exists a PPT simulator S such that:*
$$\Pr[\mathbf{Real}_{\mathcal{A}}^{\Pi}(\lambda, Q) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A}, S, \mathcal{L}}^{\Pi}(\lambda, Q) = 1] \leq \mathsf{negl}(\lambda)$$
where λ is the security parameter and $\mathsf{negl}(\lambda)$ a negligible function in λ.

We formalize the leakage profile of Arx and its protocols in Section 3.9, and provide proofs of security with respect to Definition 1 for non-adaptive adversaries (who select the query sequence beforehand) in an extended report [PBP16]. Informally, the leakage for ArxEq includes the list of queries that search for the same keyword; for ArxRange, the leakage includes the ranks of the bounds in the range query.

3.2.4 Admin API

We describe the API exposed by Arx to application admins. The admin can take an existing application and enhance it with Arx annotations. Arx's planner, located at the client proxy, uses this API to decide the data encryption plan, the list of Arx indices to build, and a query execution plan for each query pattern.

Following the example of Microsoft's SQL Server [Micf] and Google's Encrypted BigQuery [Gooa], Arx requires the admin to declare what operations will run on the database fields. By default, Arx considers *all* the fields in the database to be sensitive, unless specified otherwise. To use Arx, the admin specifies the following information during system setup:

1. (Optional) Annotated schema: fields that are unique, fields that are nonsensitive (if any), and field sizes;

2. The operations that run on sensitive fields;

3. The fields that should be indexed.

For the first, the admin uses the API: *collection = { field₁: info₁, ..., fieldₙ: infoₙ }*, to annotate the fields in a collection. This annotation is optional, but it benefits the performance of Arx if provided. *info* should specify "unique" if the values in the field are unique, *e.g.*, SSN. Arx automatically infers primary keys to be unique. *info* may also specify a maximum length for the field, which helps Arx choose a more effective encryption scheme.

Arx encrypts all the fields in the DB by default. However, the admin may explicitly override this behavior by specifying *info* as "nonsensitive" for a particular field. This option should only be used if (1) the admin thinks this field is not sensitive and desires to reduce encryption overhead, or (2) Arx does not support the computation on this field but the admin still wants to use Arx for the rest of the fields. However, we caution that though some fields may not be sensitive themselves, they may leak auxiliary information about other fields in the database. Hence, the admin should select such fields with care.

Second, Arx needs to know the query patterns that will run on the database. Concretely, Arx needs to know what operations run on which fields, though not the constants that will be queried— *e.g.*, for the query `select * from T where age = 10`, Arx needs to know there will be an equality check on `age`. The admin can either specify these operations directly, or provide a trace from a run of the application and Arx will automatically identify them.

Third, Arx needs to know the list of regular indices built by the application. Arx needs this information in order to provide the same asymptotic performance guarantees as an unencrypted database. Note that this requirement poses no extra work on the part of the admin, and is the same as required by a regular database.

3.2.5 Functionality

We now describe the classes of read and write queries that Arx can execute over encrypted data. As we show in Section 3.10, this functionality suffices for a wide range of applications.

Read queries. Arx supports queries of the form:

```
select [agg doc] fields from collection
        where clause [orderby fields] [limit ℓ]
```

doc denotes a document and [agg *doc*] aggregations over documents, which take the form $\sum Func(doc)$. $\sum$ can be any associative operator and *Func* an arbitrary, efficiently-computable function. Examples include sum, count, sum of squares, min, and max. More aggregations can be computed with minimal postprocessing at the client proxy by combining a few aggregations, such as average or standard deviation. The predicate *clause* is $\left[\wedge_i \mathsf{op}(f_i) \right]$ where $\mathsf{op}(f_i)$ denotes equality/range operations over a field f_i such as $=, \geq$ and $<$.

In addition to these queries, Arx supports a common form of joins—namely, foreign-key joins—which we describe in Section 3.7.

Write queries. Arx supports standard write queries such as inserts, deletes, and updates.

Constraints. Not all range/order queries are supported by Arx. First, queries may not contain range operations over more than one encrypted field—*i.e.*, $(5 \geq f_1 \geq 3) \wedge (f_2 \leq 10)$ is not supported unless f_2 is unencrypted. Second, if the query contains a limit along with range operations over an encrypted field, then it may contain an order-by operation over the encrypted field *alone*.

3.3 Encryption Building Blocks

Besides its indices, Arx relies on three semantically-secure encryption schemes. These schemes already exist in the literature, so we do not elaborate on them.

BASE is standard probabilistic encryption, *e.g.*, AES-CTR.

EQ enables equality checks using a searchable encryption scheme similar to existing work [CJJ$^+$14, SLPR15]. The EQ scheme we use is as follows. $\mathsf{EQEnc}_k(v) = (\mathsf{IV}, \mathsf{AES}_{\mathsf{KDF}_k(v)}(\mathsf{IV}))$, where IV is a random value and KDF is a key derivation algorithm based on AES. To search for a word w, $\mathsf{EQToken}_k(w)$ computes the token as $\mathsf{tok} = \mathsf{KDF}_k(w)$. To identify if the token matches an encryption, the server proxy combines tok with IV and checks to see if it equals the ciphertext: $\mathsf{EQMatch}((\mathsf{IV}, x), \mathsf{tok}) = (\mathsf{AES}_{\mathsf{tok}}(\mathsf{IV}) \overset{?}{=} x)$. Note that one cannot build an index on this encryption directly because it is randomized. Hence, Arx uses this scheme only for non-indexed fields (*i.e.*, for linear scans). When the developer desires an index on this field, Arx uses our new ArxEq index.

EQunique is a special case of EQ. In many applications, some fields have unique values, *e.g.*, primary keys, SSN. In this case, Arx makes an optimization: instead of implementing EQ with the scheme above, it uses deterministic encryption. Deterministic encryption does *not* leak frequency when values are unique. Such a scheme is very fast: the server can simply use the equality operator as if the data were unencrypted. Databases can also build indices on the field as before, so this case is an optimization for ArxEq too.

AGG enables addition using the Paillier scheme [Pai99].

3.4 ArxRange & Order-based Queries

We now present our index enabling range queries and order-by-limit operations.
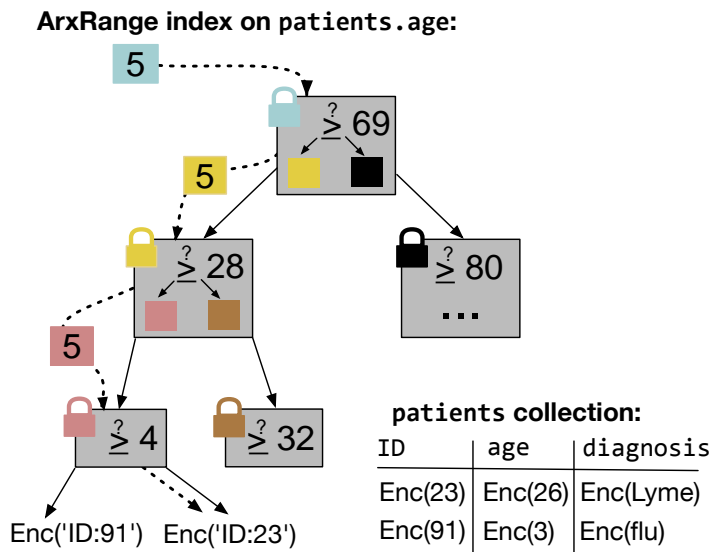
Figure 3.2: ArxRange example. Enc is encryption with BASE.

3.4.1 Strawman

We begin by presenting a helpful but inefficient strawman, that corresponds to the protocols in mOPE [PLZ13] and the startup ZeroDB [EW16]. For simplicity, consider the index to be a binary tree (instead of a regular B+ tree). To obtain the desired security, each node in the tree is encrypted using a standard encryption scheme. Because such encryption is not functional, the server needs the help of the client to traverse the index. To locate a value a in the index, the server and the client interact: the server provides the root node to the client, the client decrypts it into a value v, compares v to a, and tells the server whether to go left or right. The server then provides the relevant child to the client, and the procedure repeats until it reaches a leaf. As a result, each level in the tree requires a roundtrip, making the process inefficient.

3.4.2 Non-interactive index traversal

ArxRange enables the server to traverse the tree by itself. Say the server receives $\mathsf{BASE}_k(a)$ and must locate the leaf node corresponding to a. To achieve this goal, the server must be able to compare $\mathsf{BASE}_k(a)$ with the encrypted value at a node, say $\mathsf{BASE}_k(v)$. Inspired from the theoretical literature on garbled RAM [GLO15, GMP15], we store **a garbled circuit at each tree node** that performs the comparison, while hiding a and v from the attacker.

As described in Chapter 2, a garbling scheme is a set of algorithms (Garble, Encode, Eval) [Yao86, GMW87]. Using a garbling scheme, the client can invoke the algorithm Garble on a boolean circuit f to obtain a *garbled* version F of the circuit, along with some secret encoding information e. Given an input a, the client can run $\mathsf{Encode}(e, a)$ to produce an encoding e_a corresponding to the input. Then, the server can run $\mathsf{Eval}(F, e_a)$ and obtain the output $y = f(a)$. The security of garbled circuits

guarantees that the server learns *nothing* about a or the data hardcoded in f other than the output $f(a)$ (and the size of a and f). This guarantee holds as long as the garbled circuit is used *only once*. That is, if the client provides two encodings e_a and e_b using the same encoding information e to the server, the security guarantees no longer hold. Hence, our client provides at most one input encoding for each garbled circuit.

To allow the server to traverse the index without interaction, each node in the index must *re-encode* the input for the next node in the path, because the encoding e_a of the input to a node differs from the encoding for its children. We therefore **chain the garbled circuits** so that each circuit outputs an encoding compatible with the relevant child node.

Let N be a node in the index with value v, and let L and R be the left and right nodes. Let e^N, e^L, and e^R be the encoding information for these nodes. The garbled circuit at N is a garbling of a boolean circuit that compares the input with the hardcoded value v and additionally outputs the re-encoded input labels for the next circuit:

> **if** $a < v$ **then**
> $\quad$ $e'_a \leftarrow \mathsf{Encode}(e^L, a)$; output e'_a and 'left'
> **else**
> $\quad$ $e'_a \leftarrow \mathsf{Encode}(e^R, a)$; output e'_a and 'right'

Figure 3.2 shows how **the server traverses the index without interaction**. The number at each node indicates the value v hardcoded in the relevant garbled circuit. Now consider the query: `select * from patients where age` < 5. The client provides an encoding of 5, $\mathsf{Encode}(5)$ encrypted with the key for the root garbled circuit. The server runs this garbled circuit on the encoding and obtains "left" as well as an encoding of 5 for the left garbled circuit. The server then runs the left circuit on the new encoding, and proceeds similarly until it reaches the desired leaf node. Note that since each node encodes the $<$ operation, in order to perform $\leq$ operations the client needs to first transform the query into an equivalent query with the $<$ operation; *e.g.*, `age` ≤ 5 is transformed to `age` < 6 instead.

Repairing the index. A part of our index gets destroyed during the traversal because each garbled circuit may be used at most once. To repair the index, the client needs to supply new garbled circuits to replace the circuits consumed. Fortunately, only a logarithmic number of garbled circuits get consumed. Suppose a node N and its left child L get consumed. For each such node N, the client needs two pieces of information from the server: the value v encoded in N, and the encoding information for the right child R. The server therefore sends an encryption of v (*i.e.*, $\mathsf{BASE}(v)$, stored separated in the index), and the ID of the circuit at R. The ID of each circuit is a unique, random value that is used by the client proxy (together with the secret key) to generate the encodings for the circuit; *i.e.*, the ID of the circuit at R was used to compute e^R. Sending ID instead of e^R saves bandwidth because the encoding information is not small (1KB for a 32-bit comparison).

3.4.3 The database index

We need to take two more steps to obtain an index with the desired security.

First, the shape of the index should not leak information about the order in which the data was inserted. Hence, we use a history-independent treap [AS89, NT01] instead of a regular search tree. This data structure has the property that its shape is independent of the insertion or deletion order.

Second, we store at each node in the tree the encrypted primary key of the document containing the value. This enables locating the documents of interest. Note that the index *does not leak the order of values in the database* even though the leaves are ordered: the mapping between a leaf and a document is encrypted, and the index can be simulated from size information. If the primary key were not encrypted, the server would learn such an order.

Query execution. Consider the query select * from patients where $1 <$ age ≤ 5. Each node in the index has two garbled circuits to allow concurrent search for the lower and upper bounds. The client proxy provides tokens for values 1 and 5 to the server, which then locates the leftmost and rightmost leaves in the interval $(1, 5]$ and fetches the encrypted primary keys from all nodes in between. The server sends the encrypted keys to the client proxy which decrypts them, shuffles them, and then selects the documents mapped to these primary keys from the server. The shuffling *hides from the server the order of the documents in the range*.

For order-by-limit ℓ queries, the server simply returns the leftmost or rightmost ℓ nodes. Order-by operations without a limit are not performed using ArxRange. Since they do not have a limit, they do not do any filtering, so the client proxy can simply sort the result set itself.

Updating the index. For inserts and deletes, the server traverses the index to the appropriate position, performs the operation, and rebalances the index if required. For updates, the server first performs a delete followed by an insert. As a result of the rebalancing, all nodes that have at least one different child node are also marked as consumed (in addition to those consumed during traversal), and are sent for repair to the client proxy; however, the total number of consumed nodes is always upper bounded by the height of the index.

Some update or delete queries may first perform a filter on a field using a different index, but also requiring deletes from an ArxRange index as a result. To support this case, we maintain an encrypted *backward* pointer from the document to the corresponding node in the tree. The backward pointers enable the identification of these nodes *without* having to traverse the ArxRange index. Decryption of these pointers requires a single round of interaction with the client proxy.

Additionally, for *monotonic inserts*—a common case where inserts are made in increasing or decreasing order—a cheap optimization is for the client proxy to remember the position in the tree of the last value, so that most values can be inserted directly without requiring traversal and repair.

Concurrency. ArxRange provides limited concurrency because each index node needs to be repaired before it can be used again. To enable a degree of concurrency, the client proxy stores the top few levels of the tree. As a result, the index at the server essentially becomes a forest of trees and accesses across different trees can be performed in parallel. At the same time, the storage at the client proxy is very small because trees grow exponentially in size with the number of levels. For example, for less than 40KB of storage on the client proxy (which corresponds to about 12 levels of the tree because the tree is not entirely full), there will be about 1024 nodes in the first level of the tree, so up to 1024 queries can proceed in parallel. Queries to the same subtree, however, are still

sequential. This technique improves performance without impacting the security guarantees of the index.

3.4.4 Optimizations

We employ several techniques to further improve the performance of ArxRange: (1) we chain garbled circuits together using *transition tables* instead of computing the encoding function inside the circuit; (2) we incorporate recent advances in garbling in order to make our circuits short and fast; and (3) we remove index repair from the critical path of a query, and return the query results to the client before starting repair.

Optimizing garbled circuit chaining. For performance we do not compute the encoding function inside the garbled circuit. Instead, we chain the garbled circuits together by augmenting each garbled circuit with a *transition table*. The transition table aids in translating an input label I_i for the current circuit to an input label for the correct child circuit corresponding to the same bit value. Note that the server should not be able to infer the underlying bit value that the label corresponds to but nevertheless should be able to translate it to the correct label for the next circuit.

The garbled circuit at each node first performs the comparison $a < v$, and outputs a key K_0 or K_1 based on the result of the comparison. This key is the label of the output wire in the instantiation of the scheme.

For each bit i of the input, the transition table stores four ciphertexts. Let $I^0[i], I^1[i]$ denote the i-th input labels in the encoding e_a for the current circuit; let $O_0^0[i], O_0^1[i]$ be the corresponding labels for the left child, and $O_1^0[i], O_1^1[i]$ for the right child. The table stores the following four ciphertexts:

$$E(K_0, I^0[i], O_0^0[i]),$$
$$E(K_0, I^1[i], O_0^1[i]),$$
$$E(K_1, I^0[i], O_1^0[i]),$$
$$E(K_1, I^1[i], O_1^1[i]).$$

Here E denotes a double-key-cipher implemented as $E(A, B, X) = H(A||B) \oplus X$, where H is a random oracle. Hence, without having both A and B, it is impossible to learn any information about X. We note that there are many other instantiations of such double-key-ciphers in the literature, with different security guarantees under different assumptions but for simplicity we just resort to a random oracle in this construction.

The values in the transition table are not stored in a fixed order. Instead, we employ the point-and-permute technique [BHR12], which means that if the least significant bit of $I^0[i]$ is 0, the table entries are stored in the order as written and otherwise switched. This way the evaluator knows which ciphertext is the correct one without learning what bit value corresponds to the label.

Garbled circuit design. One of the main drawbacks of garbled circuits is that converting even a simple program to a circuit often results in large circuits, and hence bad performance. We put considerable effort into making our garbled circuits short and fast. First, we used the short circuit for comparison from [KSS09], which represents comparison of n-bit numbers in n gates. Second, we employ transition tables between two garbled circuits, to avoid embedding the encoding information for a child circuit inside the garbled circuit. Since the encoding information is large, this

optimization reduces the size of the garbled circuit by a factor of 128. Third, we use the half-gates technique [ZRE15] to further halve the size of the garbled circuit. Fourth, since all garbled circuits have the same topology but different ciphertexts, we decouple the topology from the ciphertext it contains. The server hardcodes the topology and the client transmits only ciphertexts.

3.5 ArxEq & Equality Queries

The ArxEq index enables equality queries and builds on insights from the searchable encryption literature [BHJP14], as explained in Section 3.12. We aim for ArxEq to be *forward private*, a property shown to increase security significantly in this context [Bos16]: the server cannot use an old search token on newly inserted data. We begin by presenting a base protocol that we improve in stages.

3.5.1 Base protocol

Consider an index on the field age. ArxEq will encrypt the value in age (as follows) and it will then tell the DB server to build a *regular index* on age.

The case when the fields are unique (*e.g.*, primary key, IDs, SSNs) is simple and fast: ArxEq encrypts the fields with EQunique and the regular index suffices. The rest of the discussion applies to non-unique fields.

The client proxy stores a map, called ctr, mapping each distinct value v of age that exists in the database to a counter indicating the number of times v appears in the database. For age, this map has about 100 entries.

Encrypt and insert. Suppose the application inserts a document where the field age has value v. The client proxy first increments $\text{ctr}[v]$. Then, it encrypts v into:

$$\text{Enc}(v) = H(\text{EQunique}(v), \text{ctr}[v]) \tag{3.1}$$

where H is a cryptographic hash (modeled as a random oracle). This encryption provides semantic security because $\text{EQunique}(v)$ is a deterministic encryption scheme which becomes randomized when combined with a unique salt per value v: $\text{ctr}[v]$. This encryption is not decryptable, but as discussed in Section 3.8.2, Arx encrypts v with BASE as well. The document is then inserted into the database.

Search token. When the application sends the query select $*$ where age $= 80$, the client proxy computes a search token using which the server proxy can search for all occurrences of the value 80. The search token for a value v is the list of encryptions from Equation (3.1) for every counter from 1 to $\text{ctr}[v]$: $H(\text{EQunique}(v), 1), \ldots, H(\text{EQunique}(v), \text{ctr}[v])$.

Search. The server proxy uses the search token to reconstruct the query's where clause as: age $= H(\text{EQunique}(v), 1)$ or ... or age $= H(\text{EQunique}(v), \text{ctr}[v])$ (with the clauses in a random order). The DB server uses the regular index on age for each clause in this query and returns the results. If the number of values exceeds the maximum query size allowed by the backend database, then Arx's server proxy split the disjunction into multiple queries (at the cost of additional index lookups).

Note that the scheme provides forward privacy: the server cannot use an old search token to learn if newly inserted values are equal to v as they would have a higher counter.

3.5.2 Reducing the work of the client proxy

The protocol so far requires the client proxy to generate as many tokens as there are equality matches on the field age. If a query filters on additional fields, the client proxy does more work than the size of the query result, which we want to avoid whenever possible. We now show how the client proxy can work in time $(\log \mathsf{ctr}[v])$ instead of $\mathsf{ctr}[v]$.

Instead of encrypting a value v as in Equation (3.1), the client proxy hashes according to the tree in Figure 3.3. It starts with $\mathsf{EQunique}_k(v)$ at the root of a binary tree. A left child node contains the hash of the parent concatenated with 0, and a right child contains the hash of the parent with 1. The leaves of the tree correspond to counters $0, 1, 2, 3, \ldots, \mathsf{ctr}[v]$.

The client proxy does not materialize this entire tree. Given a counter value ct, the proxy can compute the leaf corresponding to ct, simply by using the binary representation of ct to compute the corresponding hashes.
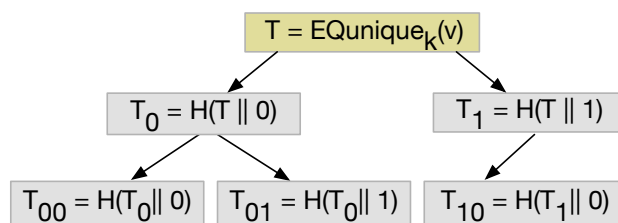
Figure 3.3: Search token tree.

New search token. To search for a value v with counter $\mathsf{ctr}[v]$, the client proxy computes the *covering set* for leaf nodes $0, \ldots, \mathsf{ctr}[v] - 1$. The covering set is the set of internal tree nodes whose subtrees cover exactly the leaf nodes $0, \ldots, \mathsf{ctr}[v] - 1$. E.g., in Figure 3.3, $\mathsf{ctr}[v] = 3$ and the covering set of the three leaves is $\{T_0, T_{10}\}$. The nodes in the covering set constitute the search token. The covering set can be easily deduced from the binary representation of $\mathsf{ctr}[v] - 1$.

Search. The server proxy expands the covering set into the leaf nodes, and proceeds as before.

3.5.3 Updates

We have already discussed inserts. For deletes, Arx simply deletes the document. An update is a delete followed by an insert. As a result, encrypted values for some counters will not return matches during search. This does not affect accuracy, but as more counters go missing, it affects throughput because the DB server wastes cycles looking for values with no matches. It also provides a small security leakage because a future search leaks how many items were deleted. As a result, ArxEq runs a cleanup procedure after each deletion. As a performance optimization, one can run a cleanup

procedure when a search query for a value v indicates more than a threshold of missing counters, relaxing security slightly.

Cleanup. The server proxy tells the client proxy how many matches were found for a search, say ct. The client proxy updates $\mathsf{ctr}[v]$ with ct, chooses a new key k' for v, and generates new tokens as in Figure 3.3: T'_{00}, ..., T'_{ct} using k'. It gives these tokens to the server, which replaces the fields found matching with these.

3.5.4 Counter map

To alleviate the burden of storing the counter map at the client proxy, it is possible to store it encrypted at the server instead while still providing strong guarantees against offline attackers. However, we recommend storing it at the client proxy for increased security against the online attacker. We now discuss both design points and accompanying tradeoffs.

Counter map at server. The counter map can be stored encrypted at the server. An entry of the sort $v \to \mathsf{ct}$ becomes $\mathsf{EQunique}_{k_1^*}(v) \to \mathsf{EQunique}_{k_2^*}(\mathsf{ct})$, where k_1^* and k_2^* are two keys derived from the master key, used for the counter map. When encrypting a value in a document or searching for a value v, the client proxy first fetches the encrypted counter from the server by providing $\mathsf{EQunique}_{k_1^*}(v)$ to the server. Then, the algorithm proceeds the same as above.

To avoid leaking the number of distinct fields, Arx pads the counter map to the number of documents in the relevant collection. This scheme satisfies Arx's security goal in Section 3.2.3: a stolen database remains encrypted with semantic security and leaks nothing except size information.

Counter map at client. However, we recommend keeping the counter map at the client proxy for higher security against an online attacker. If the counter map is stored at the server, then with every newly inserted value, an online attacker can see which entry of the counter map is accessed and which document is inserted in the database. Storing the counter map at the client hides such correlations entirely.

Though the size of the counter map grows with the number of different values a field can take, in many cases, the storage overhead is small—*e.g.*, for low-cardinality fields such as gender, age, and letter grades. Moreover, in the extreme case when all values are unique (*i.e.*, the maximum possible size for a counter map), ArxEq defaults to the regular index built over EQunique encryptions, which doesn't need a counter map at all. The case when there are many distinct values with few repetitions is less ideal, and we implement an optimization for this case: to decrease the size of the counter map, Arx groups multiple entries into one entry by storing their prefixes. As a tradeoff, the client proxy has to filter out some results.

3.6 ArxAgg & Aggregation Queries

We now explain Arx's aggregation over the encrypted indices. It is based on AES and is faster than homomorphic encryption schemes like Paillier [Pai99]. Many aggregations happen over a range query, such as computing the average days in hospital for people in a certain age group. Arx computes the average by computing sum and count at the server, and then dividing them at the client

proxy. Hence, let's focus on the query: `select sum(daysAdmitted) from patients where` $70 \leq$ `age` ≤ 80.

The idea behind aggregations in Arx is inspired from literature on authenticated data structures [LHKR10]. This work targets integrity guarantees (not confidentiality), but interestingly, we use it for computations on encrypted data. Consider the ArxRange index in Figure 3.2 built on age. At every node *N* in the tree, we add the *partial aggregate* corresponding to the subtree of *N*. For the query above, *N* contains a partial sum of `daysAdmitted` corresponding to the leaves under *N*. The root node thus contains the sum of all values. This value is stored encrypted with BASE.

To compute the sum over an arbitrary range such as $[70, 80]$, the server first locates the edges of the range as before, and then identifies a *perfect covering set*. Note that the covering set is *logarithmic* in the size of the index. For each node in this set, the server returns the encrypted aggregates of all its children and the encrypted value of the node itself to the client proxy, which decrypts them and sums them up.

In the case of (i) inserting/deleting a document, or (ii) modifying a field having an aggregate, the partial sums on the path from *N* to the root need to be updated, where *N* is the node corresponding to the changed document. In the second case, the client also needs to repair the path in the tree, so the partial sum update happens essentially for free.

This strategy supports any aggregation function of the form $\sum F(doc)$ where *F* is an arbitrary function whose input is a document, as explained in Section 3.2.5. For aggregates over fields with an ArxEq index, we have a similar strategy to the aggregates over a range, but we do not describe it here due to space constraints. For all other cases, we use AGG . However, the number of such cases is reduced significantly.

3.7 ArxJoin & Join Queries

We now describe how Arx supports a common class of join operations, namely, foreign-key joins. Arx extends ArxEq or ArxRange for this purpose. This assumes that the join contains:

```
select [...] from C1 join C2
      on C1.fkey = C2.ID
      where clause(C1) [and eq(C2)]
```

where C1 and C2 are the two collections being joined, `fkey` is the foreign key in C1 pointing to the primary key ID in C2, and *clause* is a predicate that can be evaluated using an ArxEq or ArxRange index. The query may additionally filter the joined documents in C2 using equality operations, denoted by *eq*(C2).

3.7.1 ArxEq-based joins

Consider an example with collection C2 having a primary key ID, and collection C1 having a field age with ArxEq, and `diagnosis` which is a foreign key pointing to C2.ID.

The primary key in the secondary collection `C2.ID` is encrypted with EQunique as before. Consider inserting a document with age 10 and diagnosis 'flu' in `C1`, and let's discuss how the client proxy encrypts this pair. Since foreign keys are not unique, `C1.diagnosis` is encrypted with BASE. Additionally, to perform the join, the client proxy computes an *encrypted pointer* for `C1.diagnosis`. When decrypted, this pointer will point to the appropriate encrypted `C2.ID`. Instead of using one key for ArxEq, the client proxy now uses two keys k_1 and k_2. It generates a token for each key as before: t_1 and t_2. The client proxy includes t_1 in the document as before, and uses t_2 to encrypt the diagnosis 'flu' as in: $J = \text{BASE}_{t_2}(\text{EQunique}(\text{'flu'}))$. J will help with the join. Hence, upon insert, the pair (10, 'flu') becomes $(\text{BASE}(10), t_1, \text{BASE}(\text{'flu'}), J)$. Note that the client does not add t_2 to the document: this prevents an attacker from decrypting the join pointer and performing joins that were not requested.

Now consider the join query: `select [...] from C1 join C2 on C1.diagnosis = C2.ID where C1.age = 10`. To execute this query, the server proxy computes t_1 and t_2 for the age of 10, as usual with ArxEq. It locates the documents of interest using t_1, and then uses t_2 to decrypt J and obtain EQunique('flu'). This value is a primary key in `C2`, and the server simply does a lookup in `C2`.

The `where` clause of the query may additionally filter documents in `C2` using an equality predicate, *e.g.*, `where age = 10 and C2.symptom = 'fever'`. To filter the joined documents by `symptom`, Arx employs the EQ protocol for equality checks as described in Section 3.3. Note that this additional filtering cannot make use of an index; hence, it is restricted to equality predicates and may not contain range operations.

3.7.2 ArxRange-based joins

Arx employs a different strategy in case the `where` clause of the join query requires an ArxRange index for execution, *e.g.*, `where C1.age > 10`. In such a scenario, ArxJoin's tokens for `C1.age` cannot be computed as described above.

Instead, the foreign key values encrypted with BASE are directly added to the nodes of the ArxRange index over `C1.age`, which already contain the encrypted primary keys of documents in `C1` (as described in Section 3.4.3). While traversing the index in order to resolve the `where` clause, the server fetches the encrypted foreign keys as well from the nodes of interest, and sends them to the client proxy for decryption as with regular ArxRange. The client decrypts the encrypted foreign keys, re-encrypts them with EQunique, shuffles them, and returns them to the server. The server then uses these values to locate the corresponding documents in `C2`, and performs the join. Note that this strategy does not bring any extra round trips between the proxies.

Updates. The semantics of updates remain unchanged in the presence of ArxJoin. Updates to the foreign key `C1.fkey` simply update the underlying index, ArxEq or ArxRange. Updates to `C2.ID` are also straightforward, and do not affect the pointers in `C1`. This is because `ID` is a primary key in `C2` and its values are unique.

3.8   Arx's Planner

Arx's planner takes as input a set of query patterns, Arx-specific annotations, and a list of regular indices (per Section 3.2.4), and produces a data encryption plan, a list of Arx-style indices, and a query plan for each pattern.

3.8.1   Index planning

Before deciding what index to build, note that ArxRange and ArxEq support *compound* indices, which are indices on multiple fields. For example, an index on (`diagnosis`, `age`) enables a quick search for `diagnosis` = `'flu'` and `age` ≥ `10`. Arx enables these by simply treating the two fields as one field alone. For example, when inserting a document with `diagnosis= 'flu'`, `age = 10`, Arx merges the fields into one field `'flu'` || `00010`, prefixing each value appropriately to maintain the equality and order relations, and then builds a regular Arx index.

When deciding what indices to build, we aim to provide the *same asymptotic performance* as the application admin expects: if she specified an index over certain fields, then the time to execute queries on those fields should be logarithmic and not require a linear scan. At the same time, we would like to build few indices to avoid the overhead of maintaining and storing them. Deciding what indices to build automatically is challenging because (1) there is no direct mapping from regular indices to Arx's indices, and (2) Arx's indices introduce various constraints, such as:

- A regular index serves for both range and equality operations. This is not true in Arx, where we have two different indices for each operation. We choose not to use an ArxRange index for equality operations because of its higher cost and different security.

- Unlike a regular index, a compound ArxEq index on (a, b) cannot be used to compute equality on a alone because ArxEq performs a complete match.

- A range or order-by-limit on a sensitive field can be computed only via an ArxRange index, so it can no longer be computed after applying a separate index.

All these are further complicated by the fact that the application admin can explicitly specify certain fields to be nonsensitive (as described in Section 3.2.4), and simultaneously declare compound indices on a mixture of fields, both sensitive and not. Similarly, queries can have both sensitive as well as nonsensitive fields in a `where` clause.

As a consequence of our performance goal and these constraints, interestingly, there are cases when Arx builds an ArxRange index on a composition of a nonsensitive and a sensitive field. Consider, for example, that the admin built an index on a, a nonsensitive field, and wants to perform a query containing where $a =$ and $s \geq$, where s is sensitive. The admin expects the DB to filter documents by a rapidly based on the index, and then, to filter the result by "$s \geq$".

If we follow the straightforward solution of building an ArxRange index on s alone, the resulting asymptotics are different. The DB will filter by s and then, it will scan the results and filter them by a, rendering the index on a useless. The reason the admin specified an index on a might be that

performance is better if the server filters on "$a=$" first; hence, the new query plan could significantly affect the performance of this query especially if the ArxRange index returns a large number of matches. To deliver the expected performance, Arx builds a composite ArxRange index on (a,s).

Note that this is beneficial for security too because the server will not learn which documents match one filter but not the other filter: the server learns only which documents matched the entire where clause in an all-or-nothing way.

Despite all these constraints, our index planning algorithm is quite simple. It runs in two stages: per-query processing and global analysis. Only the where clauses (including order-by-limit operations) matter here. The first stage of the planner treats sensitive and nonsensitive fields equally. For clarity, we use two query patterns as examples. Their where clauses are: W_1: "$a=$ and $b=$", W_2: "$x=$ and $y\geq$ and $z=$". The indices specified by the admin are on x and (a,b).

Stage 1: Per-query processing. For each where clause W_i, extract the set of filters S_i that can use the indices in a regular database. Example: For W_1, $S_1 = \{(a=,b=)\}$ and for W_2, $S_2 = \{(x=)\}$.

Then, if W_i contains a sensitive field with a range or order-by-limit operation, append a "$\geq$" filter on this field to each member of S_i, if the member does not already contain this. Based on the constraints in Section 3.2.5, a where clause cannot have more than one such field. Example: For W_1, $S_1 = \{(a=,b=)\}$, and for W_2, $S_2 = \{(x=,y\geq)\}$.

Stage 2: Global analysis. Union all sets $S = \cup_i S_i$. Remove any member $A \in S$ if there exists a member $B \in S$ such that an index on B implies an index on A. The concrete conditions for this implication depend on whether the fields involved are sensitive or not, as we now exemplify. Example: If a and b are nonsensitive, and S contains both $(a=,b=)$ and $(a=,b\geq)$, then $(a=,b=)$ is removed. If all of a, b and c are sensitive and S contains both $(a=,b=,c\geq)$ and $(a=,b\geq)$, then $(a=,b\geq)$ is removed. For W_1 and W_2 above, if b and y are sensitive (a,x,z can be either way), the indices Arx builds are: ArxEq (a,b) and ArxRange (x,y).

One can see why our planner maintains the asymptotic performance of the admin's index specification: each expression that was sped up by an index remains sped up. In Section 3.10, we show that the number of extra indices Arx builds is modest and does not blow up in practice.

3.8.2 Data layout

Next, laying out the encryption plan is straightforward:

- All values of a sensitive field are encrypted with the same key, but this key is different from field to field.

- For every aggregation in a query, decide if the where clause in this query can be supported entirely by using ArxRange or ArxEq. Concretely, the where clause should not filter by additional fields not present in the index. If so, update the metadata of the respective index to follow our aggregation strategy per Section 3.6. If not, encrypt the respective fields with AGG if the aggregate requires the computation of a sum.

Protocol (e)	**Operation**	**Leakage** ($\mathcal{L}_\mathsf{e}(\mathsf{DB},\mathsf{q})$)
EQ	where $field = w$	$\mathsf{sp}(w), \mathsf{Hist}(w)$
(Section 3.3)	insert	$\mathsf{sp}(w)$
	delete	−
ArxEq	where $field = w$	$\mathsf{sp}(w), \mathsf{Hist}(w)$
(Section 3.5)	cleanup (Section 3.5.3)	$\mathsf{sp}(w), \mathsf{Hist}(w)$
	insert, delete	−
ArxRange	where $a \leq field \leq b$	$\mathsf{rk}(a-1), \mathsf{rk}(b)$
(Section 3.4)	orderby limit ℓ	ℓ
	insert, delete v	$\mathsf{rk}(v)$
ArxJoin	the same information as ArxEq or ArxRange, depend-	
(Section 3.7)	ing on which ArxJoin was built on, as well as leakage	
	as in EQ for the foreign key, where each match iden-	
	tifies a primary key.	

Table 3.1: Query leakage in Arx's protocols.

- For every query pattern, if the where clause W_i checks equality on a field f that is not part of every element of S_i, encrypt f with EQ (since at least one query plan will need to filter this field by equality without an index).

- For every sensitive field projected by at least one query, additionally encrypt it with BASE. The reason is that EQ and our indices are not decryptable.

3.9 Security Analysis

We now formalize the security guarantees of Arx. We first develop a formal model of a database system, and then provide leakage definitions with respect to offline and online attackers. Proofs of security follow in an extended report [PBP16].

Notation. We denote the set of all binary strings of length n as $\{0,1\}^n$. We write $[a_i]_{i=1}^n$ to denote the list of values $[a_1, \ldots, a_n]$. If S is a list or a set, then $|S|$ denotes its size.

3.9.1 Preliminaries

A *database system* is a pair of stateful random access machines (Client, Server). The server Server stores a *database* DB, and the client Client can through interaction with Server compute *queries* out of a set of supported queries, which may modify the database.

Database. A *database* $\mathsf{DB} = \{\mathsf{T}_1, \ldots, \mathsf{T}_n\}$ is a set of *collections*. Each collection $\mathsf{T}_i = (\mathsf{F}_i, \mathsf{Ind}_i, [(\mathsf{id}_j, \mathsf{D}_j)]_j))$ comprises a set of *fields* $\mathsf{F}_i = \{\mathsf{f}_1, \ldots, \mathsf{f}_{m_i}\}$ of size m_i; a set of *indices* Ind_i; and a list of identifier-document pairs, where id_j is the *identifier* for document D_j. A *document*

$D_j = [w_1, \ldots, w_{m_i}]$ is a list of keywords where w_i is indexed by field f_i (denoted $w_i = D_j[f_i]$). Here, $w_i \in \{0,1\}^{\|f_i\|} \cup \{\phi\}$, where $\|f_i\|$ denotes the size of the keywords in the field's domain. Also, we write $\|T\|$ to denote the number of documents in collection T.

Given a collection T, we write $T(w)$ to denote the set of identifiers of documents that contain w, *i.e.*, $T(w) = \{id \mid \exists(id, D) \in T \text{ s.t. } w \in D\}$.

Indices. Given a collection T_i and a field f, an index $I \in Ind_i$ is a search tree built over the $D_j[f]$, for all $D_j \in T_i$. We represent the search tree as a tuple (V, E) of nodes and edges, where each node contains a function f that enables tree traversal. We define the shape of an index $shape(I) = E$ to be the set of edges. Since a field may contain multiple indices (*i.e.*, both ArxEq and ArxRange), we write $I(f)$ to refer to all the indices maintained on a field f.

Schema. Let $\mathbb{E} = \{BASE, EQ, EQunique, ArxEq, ArxRange\}$ denote the set of protocols supported by Arx. We define the *schema* $\mathbb{S}$ of a collection T_i to be its name, its set of fields, the size of each field, protocols maintained per field, and the shapes of all indices:
$$\mathbb{S}(T_i) = (i, \{f_j, \|f_j\|, plan(f_j), shape(I) \; \forall I \in I(f_j)\}_{j=1}^m).$$
Here, $plan(f_j) = \{e \in \mathbb{E}\}$ is the set of protocols maintained on field f_j. The schema of a database DB is then given by $\mathbb{S}(DB) = \bigcup_i \mathbb{S}(T_i)$.

Queries. A *predicate* $pred = (f, op)$ is a tuple comprising a field f and an operation op over the field, where $op \in \{<, \geq, =, < \text{ and } \geq, \texttt{orderby limit}\}$. A *query* $q = (ts, T, qtype, pred, params)$ is a 5-tuple that comprises a timestamp ts, the name of a collection T, a query type $qtype \in \{\texttt{read}, \texttt{insert}, \texttt{delete}\}$, a predicate $pred$, and query parameters $params$ corresponding to the predicate. We model updates as a $\texttt{delete}$ followed by an $\texttt{insert}$.

Note that our definition of a query consists only of a single predicate for simplicity of exposition. We model queries with multiple predicates as a list of single-predicate queries. To insert a document D, we model the operations as a list of $\texttt{insert}$ queries, one per field of the document; in these $\texttt{insert}$ queries, $q.pred.op = \bot$, $q.pred.f$ is the corresponding field, and $q.params$ is the value to be inserted.

As an example, for the query $\texttt{select * from patients where } 1 \leq \texttt{ age } < 5$, we have $T = \texttt{patients}$, $qtype = \texttt{read}$, $pred = (age, <)$, $params = (1, 5)$, and $D = \phi$.

We write $DB(q) = ([id_i]_i, e)$ to denote the set of identifiers of documents that satisfy q along with the protocol $e \in \mathbb{E}$ used to execute q. For inserts, deletes, and updates, $[id_i]_i$ indicates the list of documents inserted, deleted, or updated.

Admin API. During system setup, for each collection T_i, the admin supplies a *predicate set*:
$$\mathbb{P}(T_i) = \{[(f_j, op_j)]_j \mid f_j \in F_i\}$$
which is the set of query predicates that will be issued by Client over the collection (as described in Section 3.2.4). The *global predicate set* is then given by $\mathbb{P}(DB) = \bigcup_i \mathbb{P}(T_i)$.

3.9.2 Leakage definitions

We define the leakage profile of Arx, $\mathcal{L} = \{\mathcal{L}_{off}, \mathcal{L}_{on}\}$: first for the database itself (offline attacker), then for the execution of each query (online attacker).

Definition 2 (Offline leakage of a database)**.** *The leakage of a database* DB *is:*
$$\mathcal{L}_{\mathsf{off}}(\mathsf{DB}) = (\mathbb{S}(\mathsf{DB}), \mathbb{P}(\mathsf{DB}), \{\forall \mathsf{T}_i \, \|\mathsf{T}_i\|\}),$$
where $\mathbb{S}(\mathsf{DB})$ *is the schema of the database, and* $\mathbb{P}(\mathsf{DB})$ *is the global predicate set of the database.*

Before defining the leakage of queries, we define the rank of an element x in a list $L = [a_1, \ldots, a_n]$ as $\mathsf{rk}(L, x) = |\{a_i \mid a_i \leq x\}|$, and we write $\mathsf{rk}(x)$ if L is clear from context.

Our online leakage function $\mathcal{L}_{\mathsf{on}}$ is stateful, and maintains the *query history* of the database $\mathbb{Q}(\mathsf{DB}) = [\mathsf{q}_i]_i$ as a list of every query issued by Client. We denote the query history of a collection T as $\mathbb{Q}(\mathsf{T}) = \{\mathsf{q} \mid \mathsf{q} \in \mathbb{Q}(\mathsf{DB}) \text{ and } \mathsf{q}.\mathsf{T} = \mathsf{T}\}$.

Given collection T with a field f that has the EQ or ArxEq protocol maintained on it, we define the *search pattern* of a keyword w (following Bost [Bos16]) as:
$$\mathsf{sp}(w, \mathsf{T}, \mathsf{f}) = \{\mathsf{q}.\mathsf{ts} \mid \exists \, \mathsf{q} \in \mathbb{Q}(\mathsf{T}) \text{ s.t. } \mathsf{q}.\mathsf{pred} \text{ is } (\mathsf{f}, =), \text{ and } w \in \mathsf{q}.\mathsf{params}\},$$
and we write $\mathsf{sp}(w)$ if T and f are clear from context. Essentially, sp leaks which equality queries relate to the same keyword. Similarly, for collection T with a field f containing the EQ or ArxEq protocol, we define the *write history* of keyword w as:
$$\mathsf{WHist}(w, \mathsf{T}, \mathsf{f}) = \{(\mathsf{q}.\mathsf{ts}, \mathsf{q}.\mathsf{qtype}, \mathsf{id}) \mid \exists \, \mathsf{q} \in \mathbb{Q}(\mathsf{T}) \text{ s.t. } \mathsf{id} \in \mathsf{DB}(\mathsf{q}), \mathsf{q}.\mathsf{qtype} \in \{\texttt{insert}, \texttt{delete}\},$$
and $\mathsf{D}[\mathsf{f}] = w\}$,
where and D is the document corresponding to id. We write $\mathsf{WHist}(w)$ if T and f are clear from context. Essentially, WHist leaks the list of all write operations on keyword w.

Finally, let T^0 be the state of collection T in the initial database, before any queries are issued. Then, we define the *history* of keyword w as $\mathsf{Hist}(w, \mathsf{T}) = (\mathsf{T}^0(w), \mathsf{WHist}(w, \mathsf{T}))$, and we write $\mathsf{Hist}(w)$ if T is clear from context.

Definition 3 (Online leakage of queries)**.** *Let* $([\mathsf{id}_i]_i, [\mathsf{e}_j]_j) \leftarrow \mathsf{DB}(\mathsf{q})$. *Then, the leakage of a query* q *over database* DB *is:*
$$\mathcal{L}_{\mathsf{on}}(\mathsf{DB}, \mathsf{q}) = ((\mathsf{q}.\mathsf{ts}, \mathsf{q}.\mathsf{T}, \mathsf{q}.\mathsf{qtype}, \mathsf{q}.\mathsf{pred}), [\mathsf{id}_i]_i, \mathcal{L}_{\mathsf{e}}(\mathsf{DB}, \mathsf{q}))$$
where $\mathcal{L}_{\mathsf{e}}(\mathsf{DB}, \mathsf{q})$ *is additional leakage due to the protocol* e *used to execute the query, as detailed in Table 3.1.*

We note that the leakage of ArxEq, as captured in Table 3.1, is similar to that of Sophos [Bos16] and Diana [BMO17].

3.10   Evaluation

We now show that Arx supports real applications with a modest overhead.

Implementation.   While the design of Arx is decoupled from any particular DBMS, we implemented our prototype for MongoDB 3.0. Arx's implementation consists of ∼11.5K lines of Java, and ∼1800 lines of C/C++ code. We used the Netty I/O framework [Net17] to implement Arx's proxies. We also disable query logs and query caches to reduce the chance that an offline attacker gets information an online attacker would see, as discussed in Grubbs *et al.* [GRS17].

Testbed.   To evaluate the performance of Arx, we used the following setup. Arx's server proxy was collocated with MongoDB 3.0.11 on 4 cores of a machine with 2.3GHz Intel E5-2670 Haswell-EP processors and 256GB of RAM. Arx's client proxy was deployed on 4 cores of an identical machine.

Application	**Examples of fields**	**N/S**	**No. of indices**		**Total indices**	
			ArxEq	**ArxRange**	**Vanilla**	**Arx**
ShareLaTeX [Sha]	document lines, edits	1	12	4	12	16
Uncap (medical) [UNC]	heart rate, tests	–	0	2	2	2
NodeBB (forum) [Nod]	posts, comments	2	13	4	12	17
Pencilblue (CMS) [Pen]	articles, comments	3	46	27	70	73
Leanote (notes) [Lea]	notes, books, tags	5	64	28	69	92
Budget manager [Bud]	expenditure, ledgers	–	5	0	5	5
Redux (chat) [Red]	messages, groups	–	3	0	3	3

Table 3.2: Examples of applications supported by Arx: examples of fields in these applications; the number of queries not supported by Arx (N/S); how many Arx-specific indices the application requires; and the total number of indices the database builds in the vanilla application and with Arx. Since ArxAgg is built on top of ArxEq and ArxRange, we do not count it separately.

A separate machine with 48 cores was used to run the clients. In throughput experiments, we ran the clients on all 48 cores of the machine to measure the server at maximum capacity. All three machines were connected over a 1GbE network; to simulate real-world deployments, we added a latency of 10ms (RTT) to each server using the tc utility.

3.10.1 Functionality

To understand if Arx supports real applications, we evaluate Arx on seven existing applications built on top of MongoDB. We manually inspected the source code of each application to obtain the list of *unique* queries issued by them, and cross-verified the list against query traces produced during an exhaustive run of the application. All these applications contain sensitive user information, and Arx encrypts all fields in these applications by default.

Table 3.2 summarizes our results. With regard to unsupported queries across the applications, 4 of the 11 were due to timestamps; Arx can support these queries in case the timestamps are nonsensitive and explicitly specified as such by the application admin. The limitation was the number of range/order operations Arx allows in the query, as explained in Section 3.2.5. For NodeBB, the two unsupported queries performed text searches, and for Leanote, the five queries were evaluating regular expressions, both of which Arx cannot support. Even so, these are *only a small fraction* of the total queries issued, which are tens to hundreds in number. In general, the table shows that Arx can support almost all the queries issued by real applications. In cases where an application contains queries that are not supported by Arx, the application admin should consider whether the application needs the query in that form and if she can adjust it (*e.g.*, by removing a filter that is not necessary or that can be executed in the application). The admin could also consider if the unsupported data field is nonsensitive and mark it as such, but this should be done with care. The table also shows that though Arx's planner increases the number of indices by 20%, this number does not blow up. The main reason is that the number of fields with order queries that are not

Scheme	Enc.	Dec.	Token	Operation
BASE	0.327	0.13	–	–
EQ	4.998	–	2.353	Match: 2.368
EQunique	0.012	0.047	–	Equality: ~0
AGG	16,254	15,116	–	Sum: 8

Table 3.3: Microbenchmarks of cryptographic schemes used by Arx in μs

Height	Token	Cover	Expansion
8	3.7	9.5	131.9
10	4.6	14.5	542.3
12	5.5	20.5	2164.9

Table 3.4: Microbenchmarks of ArxEq operations in μs.

indexed by the application is small.

3.10.2 Cryptographic schemes microbenchmarks

The cryptographic schemes used by Arx are efficient, as shown in Table 3.3. The reported results are the median of a million iterations.

ArxEq microbenchmarks. The ArxEq protocol encrypts a value v as $(\mathsf{BASE}(v), t)$, where t is a token for the value computed using the search token tree as described in Section 3.5. The time to compute t is directly proportional to the height of the tree, involving a hash computation at each level. We evaluate the time taken to compute t for different tree heights, and report the results as the median of a 100K iterations in Table 3.4. The results show that ArxEq encryption is efficient.

   To search for v, the client proxy computes the covering set of all tokens and sends it to the server. The computation depends on the number of existing tokens for v, which ranges from 1 to 2^h where h is the height of the tree. We compute the cover for a randomly selected number of tokens, and report the median time over 100K iterations. The server proxy searches for v by expanding the covering set into all possible tokens. Table 3.4 shows that the operations are efficient, and that the client proxy does little work in comparison to the server.

ArxRange microbenchmarks. Our garbled circuits are implemented in AES, which takes advantage of existing hardware implementations. For a 32-bit value, the garbled circuit is 3088 bytes long, the time to garble is 19.8K cycles and the time to evaluate is 7.8K cycles. For a 128-bit value, the circuit is 12.3KB in size, the time to garble is 70.1K cycles (0.03ms) and the time to evaluate is 29.1K cycles.

ArxJoin microbenchmarks. ArxJoin builds on top of ArxEq and ArxRange. As a result, the performance of joins is closely tied to the performance of the underlying index. In this section, we report only on the additional performance overhead ArxJoin brings.

For joins based on ArxEq, the client proxy computes two sets of covers instead of one, thereby incurring an additional latency of $14.5\mu s$ for a token tree of height 10 (Table 3.4). The server proxy expands the additional set, and uses it to decrypt the foreign key pointers. Therefore, the latency at the server increases by (i) the cost of expanding the second covering set (Table 3.4), and (ii) the cost of decrypting the foreign key pointers in the filtered documents (Table 3.3). As a result, a query that joins 10,000 documents increases the latency of ArxEq by $542.3\mu s + 10,000 \times 0.13\mu s = 2.38$ms at the server.

Joins over ArxRange are executed by adding the encrypted foreign key pointers to the index nodes, which are sent to the client proxy for decryption. Such joins increase the latency of ArxRange by the time taken by the client proxy to decrypt the foreign key pointers. As a result, a join over 10,000 documents takes $10,000 \times 0.13\mu s = 1.3$ms longer.

3.10.3   Performance of ArxEq

We evaluate the overall performance of ArxEq (without the optimization for unique values) using relevant queries issued by ShareLaTeX. These queries filter documents by one field using ArxEq. We loaded the database with 100K documents representative of a ShareLaTeX workload.
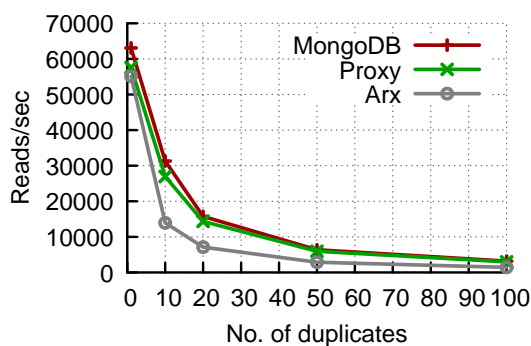
Figure 3.4: ArxEq read throughput with increasing no. of duplicates.
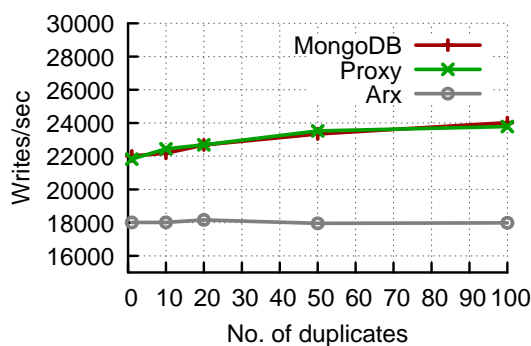
Figure 3.5: ArxEq write throughput with increasing no. of duplicates.

Figure 3.4 compares the read throughput of ArxEq with a regular MongoDB index, when varying the number of duplicates per value of the indexed field. The ArxEq scheme expands a query from a single equality clause into a disjunction of equalities over all possible tokens. The number of tokens corresponding to a value increases with the number of duplicates. The DB server essentially looks up each token in the index. In contrast, a regular index maps duplicates to a single reference and can fetch them all in a scan. Both indices need to fetch the documents for each primary key identified as a matching, which constitutes a significant part of the execution time. Overall, ArxEq incurs a penalty of 55% in the worst case, of which 8% is due to Arx's proxy. When all fields are unique, the added latency due to ArxEq is small—1.13ms versus 0.94ms for MongoDB. As the number of duplicates increases, the latency of both MongoDB and Arx increase as well—at 100 duplicates, Arx's latency is 42.1ms, while that of MongoDB is 18.8ms.
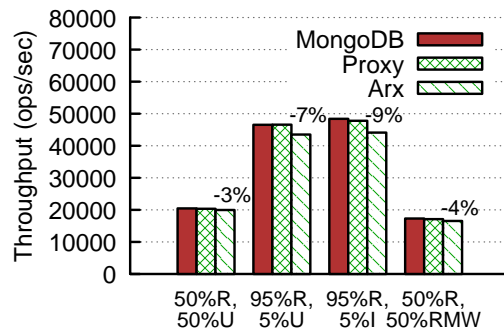
Figure 3.6: YCSB throughput for different workloads.

Dup.	Read latency (ms)			Write latency (ms)		
	Mongo	Proxy	Arx	Mongo	Proxy	Arx
1	0.94	1.04	1.13	2.69	2.72	3.30
10	1.91	2.23	4.29	2.69	2.66	3.34
20	3.81	4.19	8.49	2.62	2.65	3.28
50	9.40	10.09	20.86	2.55	2.53	3.33
100	18.80	20.23	42.10	2.50	2.51	3.35

Table 3.5: ArxEq latency of reads and writes with increasing no. of duplicates.

Figure 3.5 compares the write throughput of ArxEq with increasing number of duplicates. The write performance of a regular B+Tree index slowly improves with increased duplication, as a result of a corresponding decrease in the height of the tree. In contrast, writes to an ArxEq index are independent of the number of duplicates by virtue of security: each value looks different. Further, since each individual insert requires the computation of a single token (a constant-time operation), the write throughput of ArxEq remains stable in this experiment. As a result, the net overhead grows from 18% (when fields are unique) to 25% when there are 100 duplicates per value. Latency follows a similar trend, as shown in Table 3.5, and remains stable for ArxEq at ∼3.3ms. For a regular MongoDB index, the latency slowly improves from 2.7ms to 2.5ms as the number of duplicates grows to 100.

YCSB Benchmark. Since Arx is a NoSQL database, we also evaluate its overhead using the YCSB benchmark [CST+11]. YCSB conforms to ArxEq's optimized case when fields are unique. In this experiment, we loaded the database with 1M documents. Arx considers all fields to be sensitive by default, including the primary key. Hence, the primary key has an ArxEq index and the rest of the fields are encrypted with BASE. Figure 3.6 shows the average performance of Arx versus vanilla MongoDB, across four different workloads with varying proportions of reads and writes, as specified. "R" refers to proportion of reads, "U" to updates, "I" to inserts, and "RMW" to read-modify-write. The reduction in throughput is higher for read-heavy workloads as a result of the added latency due to sequential decryption of the result sets. Overall, the overhead of Arx is 3%-9%
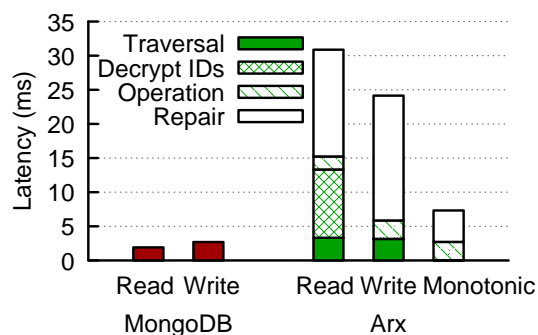
Figure 3.7: ArxRange latency of reads and writes.
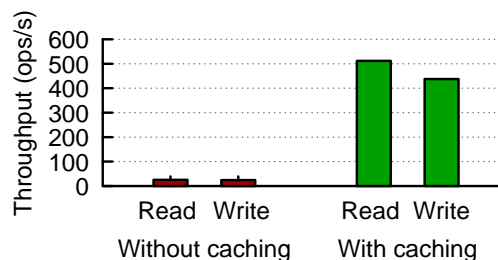
Figure 3.8: ArxRange throughput, with and without caching.

across workloads, showing that indexing primary keys is fast with Arx. Increase in latency due to Arx is also unremarkable—for example, average read latency increases from 2.31ms to 2.43ms under peak throughput, while average update latency increases from 2.36ms to 2.47ms in the 50% read-50% update workload.

3.10.4   Performance of ArxRange

We now evaluate the latency introduced by ArxRange. We pre-inserted 1M values into the index, and assumed a length of 128 bits for the index keys, which is sufficient for composite keys. We cached the top 1000 nodes of the index at the client proxy, which amounted to a mere 88KB of memory. We subsequently evaluated the performance of read and write operations on the index. Figure 3.7 illustrates the latency of each operation, divided into three parts: (1) the time taken to traverse the index, (2) the time taken to decrypt the retrieved document IDs (for reads)—this incurs a network roundtrip as described in Section 3.4.3; (3) the time taken to retrieve the corresponding results (for reads) or insert the document (for writes), and (4) the time taken to repair the index. The generation of fresh garbled circuits in order to repair the index contributes the most towards latency.

Overall, range queries cost more than writes because the former require a network roundtrip in order to decrypt the retrieved IDs before fetching the corresponding documents. The cost of traversing a path in the index is ~3ms. We note that the strawman in Section 3.4.1 incurs a roundtrip overhead for each node in the path, while our protocol incurs only a single roundtrip cost for decrypting the IDs in the leaves of the index. Figure 3.7 also highlights the improvement when the index can be optimized for monotonic inserts, which was common in the applications we evaluated. We also note that though the overall latency of ArxRange is high, the results of a range query can be returned to the client *before* performing the repair operation (see Section 3.4.4). Thus, in low load scenarios, the effective latency of a range query drops to ~15ms.

We next measure the throughput of ArxRange in Figure 3.8. Without client-side caching of nodes, the throughput of the index is very limited, since each operation requires the circuit at the root of the index to be replenished, forcing the operations to be sequential. However, when the top

few levels of the tree are cached at the client, multiple queries to different parts of the index can proceed in parallel, and the throughput increases by more than an order of magnitude (at which point the client proxy in our testbed gets saturated).

3.10.5   Performance of ArxAgg

The cost of computing an aggregate over a range in Arx is essentially equal to the cost of computing the range query. This is because traversing the index for a range query automatically computes the covering set. As a result, with 1M values in the index, aggregating over a range takes ~3 ms in Arx, equal to the cost of traversing the index.

3.10.6   End-to-end evaluation on ShareLaTeX
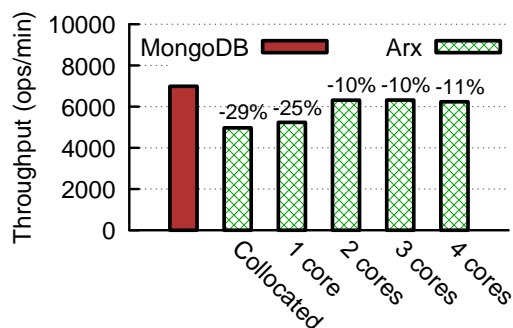
Figure 3.9: ShareLaTeX performance with Arx's client proxy on varying cores
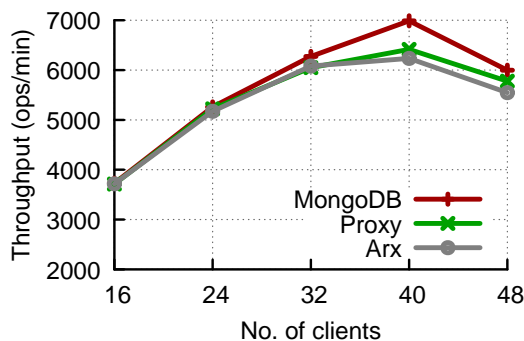
Figure 3.10: ShareLaTeX performance with increasing no. of client threads

We now evaluate the end-to-end overhead of Arx using ShareLaTeX [Sha], a popular web application for real-time collaboration on LaTeX projects, that uses MongoDB for persistent storage. We chose ShareLaTeX because it uses both of Arx's indices, it has sensitive data (documents, chats) and is a popular application. ShareLaTeX maintains multiple collections in MongoDB corresponding to users, projects, documents, chat messages, etc. We considered all the fields in the application to be sensitive, which is the default in Arx. The application was run on four cores of the client server.

Before every experiment, we pre-loaded the database with 100K projects, 200K users, and other collections with 100K records each. Subsequently, using Selenium (a tool for automating browsers [Sel]), multiple users launch browsers in parallel and collaborate on projects in pairs—(i) editing documents, and (ii) exchanging messages via chat. We ran the user processes on a separate machine with 48 cores. Figure 3.9 shows the throughput of ShareLaTeX in a vanilla deployment with regular MongoDB, compared to its performance with Arx in various configurations. The client proxy is either collocated with the ShareLaTeX application sharing the same four cores, or deployed on extra and separate cores. The application's throughput declines by 29% when the client proxy

and ShareLaTeX are collocated; however, when two separate cores are allocated to Arx's client proxy, the reduction in throughput stabilizes at a reasonable 10%.

Figure 3.10 compares the performance of Arx with increasing load at the application server, when four separate cores are allocated to Arx's client proxy. It also shows the performance of MongoDB with the Netty [Net17] proxy without the Arx hooks. Note that each client thread issues many requests as fast as it can, bringing a load equivalent to many real users. At peak throughput with 40 clients and 100% CPU load at the application, the reduction in performance due to Arx is 11%; 8% is due to Arx's proxy, and the remaining 3% due to its encryption and indexing schemes.

Finally, the latency introduced by Arx is modest compared to the latency of the application. In conditions of low stress with 16 clients, performance remains bottlenecked at the application, and the latency added by Arx is small in comparison, increasing from an average of 268ms per operation to 280ms. At peak throughput, the latency of vanilla ShareLaTeX is 355ms, which grows by 15% to 408ms with Arx, having marginal impact on user experience.

In sum, Arx brings a modest overhead to the overall web application. There are two main reasons for this. First, web applications have a significant overhead themselves at the web server, which masks the latency of Arx's protocols. Second, even though ArxRange is not cheap, it's one out of a set of multiple operations Arx runs, with the others being faster and overall more common in applications.

3.10.7 Storage

Arx increases the total amount of data stored in the database because: (1) ciphertexts are larger than plaintexts for certain encryption schemes, and (2) additional fields are added to documents in order to enable certain operations, *e.g.*, equality checks using EQ, or tokens for ArxEq indexing. Further, ArxRange indices are larger than regular B+Trees, because each node in the index tree stores garbled circuits. Vanilla ShareLaTeX with 100K documents per collection occupied 0.56GB in MongoDB, with an extra 48.7 MB for indices. With Arx, the data storage increased by $1.9\times$ to 1.05GB. The application required three compound ArxRange indices, which together occupied 8.4GB of memory at the server proxy while indices maintained by the database occupied 56.5MB. This resulted in a net increase of $16\times$ at the DB server. We note, however, there remains substantial scope for optimizing index size in our implementation.

Finally, the application required two ArxEq indices for which counter maps were maintained at the client proxy, which in turn occupied 8.6MB of memory, illustrating that ArxEq imposes modest storage overhead at the application server. Moreover, the values inserted into the counter maps were distinct; in case of duplicates, the memory requirements would be proportionately lower.

3.10.8 Comparison with CryptDB

Arx supports fewer queries than CryptDB, but we find their functionalities are nevertheless comparable. For example, CryptDB supports all the queries in the TPC-C benchmark [TPCa], while Arx supports 30 out of 31 queries. Arx also enables a rich class of applications as shown above, though

it does not support group-by operations (for security issues), arbitrary conjunctions of range filters, and more generic joins, supported by CryptDB.

As regards performance, on one hand, CryptDB's order and equality queries via PPE schemes are faster than Arx's—with a reported overhead of ∼1ms [PRZB11], as opposed to a few milliseconds in Arx—but also significantly less secure. On the other hand, Arx's aggregate over a range is an order of magnitude faster for the same security, because CryptDB uses Paillier [Pai99] to compute aggregates which requires a homomorphic multiplication per value in the range. For a range of 10,000 values, aggregates take 80ms in CryptDB compared to ∼3ms in Arx. Overall, Arx is a heavier solution due to the significant extra security, but remains at par with CryptDB in terms of the overall impact on applications: both systems report an overhead on the order of 10%, and an added latency on the order of milliseconds per operation.

3.11   Limitations and Future Work

ArxRange extensions.   Our current ArxRange index is a binary tree. An interesting extension is to implement the index for data structures with higher fanout such as B-trees, *e.g.*, by (i) storing at each node in the tree multiple garbled circuits; and (ii) using a history-independent B-treap data structure [Gol09], instead of a binary treap.

History-independence.   One needs to be careful that when logically implementing a history-independent data structure (as in ArxRange), the physical implementation of it is history-independent as well. For example, in our treap data structure, we ultimately require file system support for implementing *secure deletion* [BS13, RBC13]. This is because, when a node is logically deleted, the file system needs to ensure that instead of merely unlinking the data structure in memory, all copies of the data (caches, in-memory and disk) are in fact physically removed so as to become irrecoverable to an attacker. Implementing secure deletion is complementary to our work.

Transactions.   Arx currently does not support transactional semantics. While our techniques can be extended to transactional systems as well, it has significant practical challenges. For instance, our ArxRange index requires updates to multiple nodes in the tree per query along with interaction with the client, making support for transactions complicated. However, doing so is interesting future work.

3.12   Related Work

We compare Arx with state-of-the-art EDBs, and discuss protocols related to its building blocks, ArxEq and ArxRange. Early approaches [HILM02] used heuristics instead of encryption schemes with provable security. We do not discuss PPE-based EDBs [PRZB11, AEK⁺13, TKMZ13, PBC⁺16] further as we have already compared Arx against them extensively in Section 3.1 and Section 3.2.3. Seabed [PBC⁺16] hides frequency in some cases, but still uses PPE.

EDBs using semantically-secure encryption.   This category is the most relevant to Arx, but unfortunately, there is little work done in this space. First, the line of work in [CJJ⁺14, FJK⁺15]

is based on searchable encryption, but is too restricted in functionality. It does not support joins, order-by-limit queries (commonly used for pagination, more common than range queries in TPC-C [TPCa]), or aggregates over a range (because the range identifies a superset of the relevant documents for security, yielding an incorrect aggregate). Regarding security, while being significantly more secure than PPE-based EDBs for offline attackers, for online attackers they could leak more than PPE-based EDBs because their range queries leak the number of values matching sub-ranges as well as some prefix matching information—leakage that is not implied by order. Arx addresses all these aspects. Other recent works [KM16, KLL+16] also support equality-based queries but do not support range, order-by-limit, or aggregates over range queries; the former doesn't support inserts or updates either.

Second, BlindSeer [PKV+14] is another EDB providing semantic security. BlindSeer provides stronger security than Arx and even hides the client query from the server through two-party computation. Its primary drawbacks with respect to Arx are performance and functionality. BlindSeer requires a large number of interactions between the client and server. For example, for a range query, the client and server need to interact for every data item in the range (and a few times more) because tree traversal is interactive. If the range contains many values, this query is slow. In Arx, there is no interaction in this case. BlindSeer also does not handle inserts easily, nor does it support deletes, updates, aggregates over ranges or order-by-limit queries.

Finally, Obladi [CBC+18] targets much stronger guarantees than Arx by combining ACID semantics with ORAM, but consequently, is also orders of magnitude slower.

Work related to ArxEq. ArxEq falls in the general category of searchable-encryption schemes and builds on insights from this literature. While there are many schemes proposed in this space [SWP00, CGKO06, KPR12, OKKM13, CJJ+14, FJK+15, BHJP14, LCS+14, Kur14, NPG14, SPS14, HAJ+14, Bos16], none of them meets the following desired security and performance from a database index. Besides semantic security, when inserting a value, the access pattern should not leak what other values it equals, and an old search token should not allow searching on newly inserted data (forward privacy), both crucial in reducing leakage [Bos16]. Second, inserts, updates and deletes should be efficient and should not cause reads to become slow. ArxEq meets all these goals. Perhaps the closest prior work to ArxEq is [CJJ+14]. This scheme uses revocation lists for delete operations, which adds significant complexity and overhead, as well as leaks more than our goal in Arx: it lacks forward privacy and the revocation lists leak various metadata. Sophos [Bos16] also provides forward privacy, but uses expensive public key cryptography instead of symmetric key. Diana [BMO17] is similar to ArxEq.

Work related to ArxRange. There has been a significant amount of work on OPE schemes in both industry and research communities [ÖSC03, AKSX04, YKK+11, AEAEM09, LPL+09, KAK10, XYH12, LW12, LW13, PLZ13, BCLO09, BCO11, AWW12, Cipb, Ras11]. OPE schemes are efficient but have significant leakage [NKW15]. Order-revealing encryption (ORE) provides semantic security [BLR+14, CLWW16, LW16]. The most relevant of these is the construction by Lewi and Wu [LW16] which is more efficient than ArxRange because it does not need replenishment, but also less secure because it leaks the position where two plaintexts differ. Thus, it is not strictly more secure than OPE.

3.13   Summary

In summary, Arx is a practical and functionally rich database system that encrypts data only with semantically secure schemes. As a result, Arx provides significantly stronger security than PPE-based EDBs, while ensuring that the performance of the system remains satisfactory. Our evaluation shows that Arx supports real applications such as ShareLaTeX with a modest overhead.

Chapter 4

Collaborative SQL Analytics on Encrypted Data

This chapter presents Senate [PKY⁺21], a platform that enables multiple parties to jointly run SQL analytics on their collective data, without sharing their unencrypted data with each other. At the heart of Senate lies a new MPC protocol that decomposes the cryptographic computation into smaller units, some of which can be executed by subsets of parties and in parallel, while still providing strong security guarantees against malicious attackers.

4.1   Introduction

A large number of services today collect valuable sensitive user data. These services could benefit from pooling their data together and jointly performing query analytics on the aggregate data. For instance, such collaboration can enable better medical studies [BEE⁺17, KBV13]; identification of criminal activities (*e.g.*, fraud) [SvHA⁺19]; more robust financial services [SvHA⁺19, BFLV12, AKL12, PNH17]; and more relevant online advertising [IKN⁺17]. However, many of these institutions cannot share their data with each other due to privacy concerns, regulations, or business competition.

As described in Chapter 2, secure multi-party computation [Yao82, GMW87, BGW88] (MPC) promises to enable such scenarios by allowing *m* parties, each having secret data d_i, to compute a function f on their aggregate data, and to share the result $f(d_1, \ldots, d_m)$ amongst themselves, *without* learning each other's data beyond what the function's result reveals. To summarize at a high level, MPC protocols work by having each party encrypt its data, and then perform joint computations on encrypted data leading to the desired result.

Despite the pervasiveness of data analytics workloads, there are very few works that consider secure collaborative analytics. While closely related works such as SMCQL [BEE⁺17] and Conclave [VSG⁺19] make useful first steps in the direction of secure collaborative analytics, their main limitation is their weak security guarantee: *semi-honest security*. Namely, these works assume that each party, even if compromised, follows the protocol faithfully. If any party deviates from the
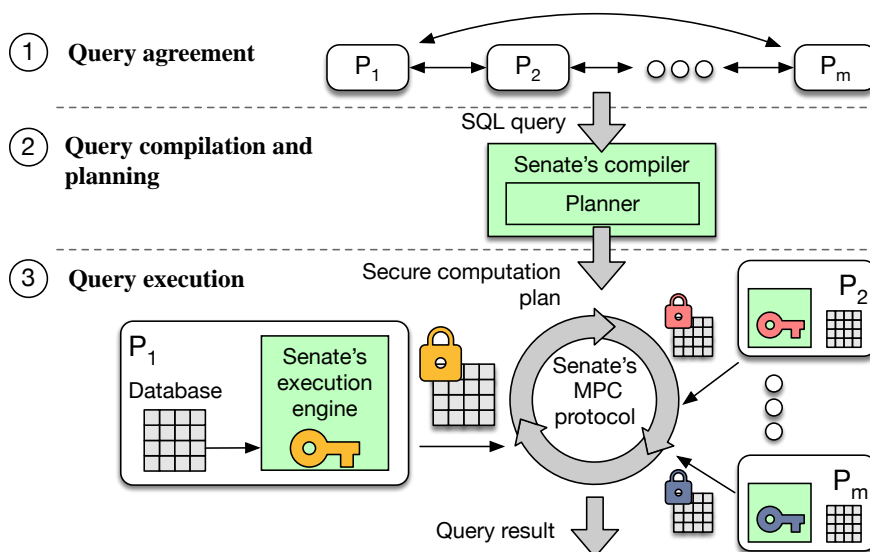
Figure 4.1: Overview of Senate's workflow.

protocol, it can, in principle, extract information about the sensitive data of other parties. This is an unrealistic assumption in many scenarios for two reasons. First, each party running the protocol at their site has full control over what they are actually running. For example, it requires a bank to place the confidentiality of its sensitive business data in the hands of its competitors. If the competitors secretly deviate from the protocol, they could learn information about the bank's data without its knowledge. Second, in many real-world attacks [Per14], attackers are able to install software on the server or obtain control of a server [dir19], thus allowing them to alter the server's behavior.

4.1.1 Senate overview

We present Senate, a platform for secure collaborative analytics with the strong guarantee of *malicious security*. In Senate, even if $m - 1$ out of m parties fully misbehave and collude, an honest party is guaranteed that nothing leaks about their data other than the result of the agreed upon query. Our techniques come from a synergy of new cryptographic design and insights in query rewriting and planning. A high level overview of Senate's workflow (as shown in Figure 4.1) is as follows:

Agreement stage. The *m* parties agree on a shared schema for their data, and on a query for which they are willing to share the computation result. This happens before invoking Senate and may involve humans.

Compilation and planning stage. Senate's compiler takes the query and certain size information (described in Section 4.2) as input and outputs a cryptographic execution plan. It runs at each party, deterministically producing the same plan. In particular, the compiler employs our *consistent and verifiable query splitting* technique in order to minimize the amount of joint computation performed
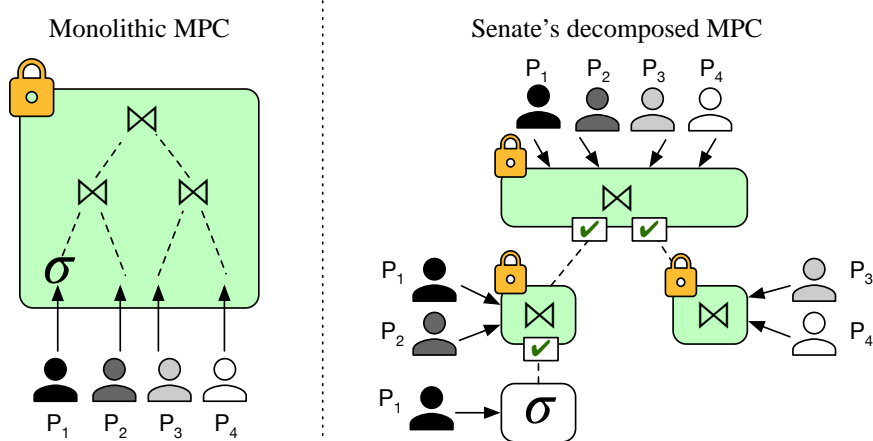
Figure 4.2: Query execution in the baseline (monolithic MPC) vs. Senate (decomposed MPC). σ represent a filtering operation, and $\bowtie$ is a join. Green boxes with locks denote MPC operations; white boxes denote plaintext computation. ✓ represents additional verification operations added by Senate.

by the parties. Then, the compiler plans the execution of the joint computation using our *circuit decomposition* technique, which can produce a significantly more efficient execution plan.

Execution stage. An execution engine at each party runs the cryptographic execution plan by coordinating with the other parties and routing encrypted intermediate outputs based on the plan. This is done using our efficient *MPC decomposition protocol*, which outputs the query result to all the parties.

4.1.2 Senate's techniques

Designing a maliciously-secure collaborative analytics system is challenging due to the significant overheads of such strong security. Consider simply using a state-of-the-art *m*-party maliciously-secure MPC tool such as AGMPC [EMP] which implements the protocol of Wang *et al.* [WRK17]; we refer to this as the *baseline*. When executing a SQL query with this baseline, the query gets transformed into a single, large Boolean circuit (*i.e.*, a circuit of AND, OR, XOR gates) taking as input the data of the *m* parties. The challenge then is that the *m* parties need to execute a *monolithic* cryptographic computation *together* to evaluate this circuit.

Minimizing joint computation. Prior work [BEE+17, VSG+19] in the semi-honest setting shows that one can significantly improve performance by splitting a query into local computation (the part of the query that touches only one party's data) and the rest of the computation. The former can be executed locally at the party on plaintext, and the latter in MPC; *e.g.*, if a query filters by "disease = flu", the parties need to input only the records matching the filter into MPC as opposed to the entire dataset. In the semi-honest setting, the parties are trusted to perform such local computation

faithfully. Unfortunately, this technique no longer works with malicious parties because a malicious party *M* can perform the local computation:

- *incorrectly*. For example, *M* can input records with "`disease = HIV`" into MPC. This can reveal information about another party's "`HIV`" records, *e.g.*, via a later join operation, when this party might have expected the join to occur only over rows with the value "`flu`".

- *inconsistently*. For example, if one part of a query selects patients with "`age = 25`" and another with "`age` $\in [20, 30]$", the first filter's outputs should be included within the second's. However, *M* might provide inconsistent sets of records as the outputs of the two filters.

Senate's *verifiable and consistent query splitting* technique allows Senate to take advantage of local computation via a different criteria than in the semi-honest case. Given a query, Senate's compiler splits the query into a special type of local computation—one that does not introduce inconsistencies—and a joint computation, which it annotates with verification of the local computation, in such a way that the verification is faster to execute than the actual computation. For example, Figure 4.2 shows a 4-party query in which party P_1's inputs are first filtered (denoted σ). Unlike the baseline execution, Senate enables P_1 to evaluate the filter locally on plaintext, and the secure computation proceeds from there on the smaller filtered results; these results are then jointly verified.

Decomposing MPC. In order to decompose the joint computation (instead of evaluating a single, large circuit using MPC) one needs to open up the cryptographic black box. Consider a 4-way join operation ($\bowtie$) among tables of 4 parties, as shown in Figure 4.2. With the baseline, all 4 parties have to execute the whole circuit. However, if privacy were not a concern, P_1 and P_2 could join their tables without involving the other parties, P_3 and P_4 do the same *in parallel*, and then everyone performs the final join on the smaller intermediate results. This is not possible with existing state-of-the-art protocols for MPC, which execute the computation in a *monolithic* fashion.

To enable such decomposition, we design a new cryptographic protocol we call *secure MPC decomposition* (Section 4.4), which may be of broader interest beyond Senate. In the example above, our protocol enables parties P_1 and P_2 to evaluate their join obtaining an encrypted intermediate output, and then to *securely reshare* this output with parties P_3 and P_4 as they all complete the final join. The decomposed circuits include verifications of prior steps needed for malicious security. We also develop more efficient Boolean circuits for expressing common SQL operators such as joins, aggregates and sorting (Section 4.6), using a small set of Boolean circuit primitives which we call *m*-SI, *m*-SU and *m*-Sort (Section 4.5).

Efficiently planning query execution. Finally, we develop a new *query planner*, which leverages Senate's MPC decomposition protocol (Section 4.7.1). Unsurprisingly, the circuit representation of a complex query can be decomposed in many different ways. However, the rules governing the cost of each execution plan differ significantly from regular computation. Hence, we develop a *cost model* for our protocol which estimates the cost given a circuit configuration (Section 4.7.2). Senate's query planner selects the most efficient plan based on the cost model.

4.1.3   Evaluation summary

We implemented Senate and evaluate it in Section 4.8. Our decomposition and planning mechanisms result in a performance improvement of up to $10\times$ compared to the monolithic circuit baseline, with up to $11\times$ less resource consumption (memory / network communication), on a set of representative queries. Senate's query splitting technique for local computation can *further* increase performance by as much as $10\times$, bringing the net improvement to up to $100\times$. Furthermore, to stress test Senate on more complex query structures, we also evaluate its performance on the TPC-H analytics benchmark [TPCb]; we find that Senate's improvements range from $3\times$ to $145\times$.

Though MPC protocols have improved steadily, they still have notable overhead. Given that such collaborative analytics do not have to run in real time, we believe that Senate can already be used for simpler workloads and / or relatively small databases, but is not yet ready for big data analytics. However, we expect faster MPC protocols to continue to appear. The systems techniques in Senate will apply independently of the protocol, and the cryptographic decomposition will likely have a similar counterpart.

4.2   Senate's API

Senate exposes a SQL interface to the parties. To reason about which party supplies which table in a collaborative setting, we augment the query language with the simple notation $R|P$ to indicate that table R comes from party P. Hence, $R|P_1 \cup R|P_2$ indicates that each party holds a *horizontal* partition of table R. One can obtain a *vertical* partitioning, for example, by joining two tables from different parties $R_1|P_1$ and $R_2|P_2$. Here, we use the $\cup$ operator to denote a simple concatenation of the tables, instead of a set union (which removes duplicates).

In principle, Senate can support arbitrary queries because it builds on a generic MPC tool. The performance improvement of our techniques, though, is more relevant to joins, aggregates, and filters. We now give three use cases and queries, each from a different domain, which we use as running examples.

Query 1. Medical study [BEE$^+$17]. Clostridium difficile (cdiff) is an infection that is often antibiotic-resistant. As part of a clinical research study, medical institutions $P_1 \dots P_m$ wish to collectively compute the most common diseases contracted by patients with cdiff. However, they cannot share their databases with each other to run this query due to privacy regulations.

```
SELECT diag, COUNT(*) AS cnt
FROM diagnoses|P1 ∪...∪ diagnoses|Pm
     WHERE has_cdiff = 'True'
     GROUP BY diag ORDER BY cnt LIMIT 10;
```

Query 2. Prevent password reuse [WR19]. Many users unfortunately reuse passwords across different sites. If one of these sites is hacked, the attacker could compromise the account of these users at other sites. As studied in [WR19], sites wish to identify which users reuse passwords across the sites, and can arrange for the salted hashes of the passwords to match if the underlying passwords are the same (and thus be compared to identify reuse using the query below). However,

these sites do not wish to share what other users they have or the hashed passwords of these other users (because they can be reversed).

```
SELECT user_id
FROM passwords|P₁ ∪...∪ passwords|Pₘ
        GROUP BY CONCAT(user_id, password)
        HAVING COUNT(*) > 1;
```

Query 3. Credit scoring agencies do not want to share their databases with each other [VSG+19] due to business competition, yet they want to identify records where they have a significant discrepancy in a particular financial year. For example, an individual could have a low score with one agency, but a higher score with another; the individual could take advantage of the higher score to obtain a loan they are not entitled to.

```
SELECT c1.ssn
FROM credit_scores|P₁ AS c1
...
JOIN credit_scores|Pₘ AS cm ON c1.ssn = cm.ssn
WHERE GREATEST(c1.credit, ..., cm.credit) -
  LEAST(c1.credit, ..., cm.credit) > threshold
  AND c1.year = 2019 ... AND cm.year = 2019;
```

4.2.1 Sizing information

Given a query, Senate's compiler first splits the query into local and joint computation. Each party then specifies to the compiler an upper bound on the number of records it will provide as input to the joint computation, following which the compiler maps the joint computation to circuits. These upper bounds are useful because we do not want to leak the size of the parties' inputs, but also want to improve performance by not defaulting to the worst case, *e.g.*, the maximum number of rows in each table. For example, for Query 1, Senate transforms the query so that the parties group their records locally by the column diag and compute local counts per group. In this case, Senate asks for the upper bound on the number of diagnoses per party. In many cases, deducing such upper bounds is not necessarily hard: *e.g.*, it is simple for Query 1 because there is a fixed number of known diseases [Cen17]. Further, meaningful upper bounds significantly improve performance.

4.3 Threat Model and Security Guarantees

Senate adopts a strong threat model in which a malicious adversary can corrupt $m - 1$ out of m parties. The corrupted parties may arbitrarily deviate from the protocol and collude with each other. As long as one party is honest, the only information the compromised parties learn about the honest party is the final global query result (in addition to the upper bounds on data size provided to the compiler by the parties, and the query itself).

More formally, we define an ideal functionality $\mathcal{F}_{\mathsf{MPC\text{-}tree}}$ (Functionality 2, Section 4.4.3) for securely executing functions represented as a tree of circuits, while placing some restrictions on

the structure of the tree. We then develop a protocol that realizes this functionality and prove the security of our protocol (per Theorem 2, Section 4.4.3) according to the definition of security for (standalone) maliciously secure MPC [Gol04b], as captured formally by the following definition:

Definition 4. *Let $\mathcal{F}$ be an m-party functionality, and let Π be an m-party protocol that computes $\mathcal{F}$. Protocol Π is said to* securely compute $\mathcal{F}$ *in the presence of static malicious adversaries if for every non-uniform PPT adversary A for the real model, there exists a non-uniform PPT adversary $\mathcal{S}$ for the ideal model, such that for every $I \subset [m]$*

$$\{\text{IDEAL}_{\mathcal{F},I,\mathcal{S}(z)}(\bar{x})\}_{\bar{x},z} \overset{\text{c}}{\equiv} \{\text{REAL}_{\Pi,I,\mathcal{A}(z)}(\bar{x})\}_{\bar{x},z}$$

where $\bar{x} = (x_1,\ldots,x_m)$ and $x_i \in \{0,1\}^*$.

Here, $\text{IDEAL}_{\mathcal{F},I,\mathcal{S}(z)}(\bar{x})$ denotes the joint output of the honest parties and $\mathcal{S}$ from the ideal world execution of $\mathcal{F}$; and $\text{REAL}_{\Pi,I,\mathcal{A}(z)}(\bar{x})$ denotes the joint output of the honest parties and $\mathcal{A}$ from the real world execution of Π [Gol04b].

As with malicious MPC, we cannot control what data a party chooses to input. The parties can, if they wish, augment the query to run tests on the input data (*e.g.*, interval checks). Senate also does not intend to maintain consistency of the datasets input by a party across *different* queries as the dataset could have changed in the meantime. If this is desired, Senate could in principle support this by writing multiple queries as part of a single bigger query, at the expense of performance.

Note that the query result might leak information about the underlying datasets, and the parties should choose carefully what query results they are willing to share with each other. Alternatively, it may be possible to integrate techniques such as differential privacy [DR14, JNS18] with Senate's MPC computation, to avoid leaking information about any underlying data sample; we discuss this aspect in more detail in Section 4.9.

4.4   Senate's MPC Decomposition Protocol

In this section we present Senate's *secure MPC decomposition* protocol, the key enabler of our compiler's planning algorithm. Our protocol may be of independent interest, and we present the cryptography in a self-contained way.

Suppose that m parties, $P_1,\ldots,P_m$, wish to securely compute a function f, represented by a circuit C, on their private inputs x_i. This can be done easily given a state-of-the-art MPC protocol by having all the parties collectively evaluate the entire circuit using the protocol. However, the key idea in Senate is that if f can be "nicely" decomposed into multiple sub-circuits, we can achieve a protocol with a significantly better concrete efficiency, by having only a *subset* of the parties participate in the secure evaluation of each sub-circuit.

For example, consider a function $f(x_1,\ldots,x_m)$ that can be evaluated by separately computing $y_1 = h_1(x_1,\ldots,x_i)$ on the inputs of parties $P_1 \ldots P_i$, and $y_2 = h_2(x_{i+1},\ldots,x_m)$ on the inputs of parties $P_{i+1} \ldots P_m$, followed by $\tilde{f}(y_1,y_2)$. That is,

$$f(x_1,\ldots,x_m) = \tilde{f}\big(h_1(x_1,\ldots,x_i),h_2(x_{i+1},\ldots,x_m)\big).$$

Such a decomposition of f allows parties $P_1,\ldots,P_i$ to securely evaluate h_1 on their inputs (using an MPC protocol) and obtain output y_1. In parallel, parties $P_{i+1},\ldots,P_m$ securely evaluate h_2 to

get y_2. Finally, all parties securely evaluate $\tilde{f}$ on y_1, y_2 and obtain the final output y. We observe that such a decomposition may lead to a more efficient protocol for computing f, since the overall communication and computation complexity of state-of-the-art concretely efficient MPC protocols (*e.g.*, [WRK17, KPR18]) is at least quadratic in the number of involved parties. Furthermore, sub-circuits involving disjoint sets of parties can be evaluated in parallel.

 Although appealing, this idea has some caveats:

1. In a usual ("monolithic") secure evaluation of f, the intermediate values y_1, y_2 remain secret, whereas the decomposition above reveals them to the parties as a result of an intermediate MPC protocol.

2. Suppose that h_1 is a *non-easily-invertible* function (*e.g.*, pre-image resistant hash function). If all of $P_1, \ldots, P_i$ collude, they can pick an arbitrary "output" y_1, even without evaluating h_1, and input it to $\tilde{f}$. Since h_1 is non-invertible, it is infeasible to find a pre-image of y_1; thus, such behavior is not equivalent to the adversary's ability to provide an input of its choice (as allowed in the malicious setting). In addition, such functions introduce problems in the proof's simulation as a PPT simulator cannot extract the corrupted parties' inputs with high probability. This attack, however, would not have been possible if f had been computed entirely by all of $P_1, \ldots, P_m$ in a *monolithic* MPC.

3. If one party is involved in multiple sub-circuits and is required to provide the same input to all of them, then we have to make sure that its inputs are consistent.

In this section we show how to deal with the above problems, by building upon the MPC protocol of Wang *et al.* [WRK17].

 First, we show how to securely transfer the output of one garbled circuit as input to a subsequent garbled circuit, an action called *soldering* (Section 4.4.2). Our soldering is inspired by previous soldering techniques proposed in the MPC literature [NO09, FJN+13, FJNT15, KNR+17, AHMR15, GLOS15, HY16, GGMP16, LO17, KY18, PBP19, BPP16]. Here, we make the following contributions. To the best of our knowledge, Senate is the first work to design a soldering technique for the state-of-the-art protocol of Wang *et al.* [WRK17]. More importantly, whereas previous uses of soldering were limited to cases in which the *same set of parties* participate in both circuits, we show how to solder circuits when the first set of parties *is a subset of* the set of parties involved in the second circuit. This property is crucial for the performance of the individual sub-circuits in our overall protocol, as most of them can now be evaluated by non-overlapping subsets of parties, in parallel.

 Second, as observed above, the decomposition of a function for MPC cannot be arbitrary. We therefore formalize the class of decompositions that are *admissible* for MPC (Section 4.4.3). Informally, we require that every sub-computation evaluated by less than m parties must be efficiently invertible. This fits the ability of a malicious party to choose its input before providing it to the computation.

 Furthermore, we define the admissible circuit structures to be *trees* rather than directed acyclic graphs. That is, the function's decomposition may only take the form of a tree of sub-computations, and not an arbitrary graph. This is because if a node provides input to more than one parent node

and all the parties at the node are corrupted, they may collude to provide inconsistent inputs to the different parents. We therefore circumvent this input consistency problem by restricting valid decompositions to trees alone. Even so, as we show in later sections, this model fits SQL queries particularly well, since many SQL queries can be naturally expressed as a tree of operations.

4.4.1 Background

We start by briefly introducing the cryptographic tools that our MPC protocol builds upon. In particular, we build upon the maliciously-secure garbled circuit protocol of Wang *et al.* [WRK17] (hereafter referred to as the WRK protocol).

Information-theoretic MACs (IT-MACs). IT-MACs [NNOB12] enable a party P_j to authenticate a bit held by another party P_i. Suppose P_i holds a bit $x \in \{0,1\}$, and P_j holds a key $\Delta_j \in \{0,1\}^\kappa$ (where κ is the security parameter). Δ_j is called a *global key* and P_j can use it to authenticate multiple bits across parties. Now, for P_j to be able to authenticate x, P_j is given a random *local key* $K_j[x] \in \{0,1\}^\kappa$ and P_i is given the corresponding MAC tag $M_j[x]$ such that:
$$M_j[x] = K_j[x] \oplus x\Delta_j.$$
P_j does not know the bit x or the MAC, and P_i does not know the keys; thus, P_i can later reveal x and its MAC to P_j to prove it did not tamper with x. In this manner, P_i's bit x can be authenticated to more than one party—each party j holds a global key Δ_j and local key for x, $K_j[x]$. P_i holds all the corresponding MAC tags $\{M_j[x]\}_{j \neq i}$. We write $[x]^i$ to denote such a bit where x is known to P_i, and is authenticated to *all* other parties. Concretely, $[x]^i$ means that P_i holds $(x, \{M_j[x]\}_{j \neq i})$, and every other party $P_j \neq P_i$ holds $K_j[x]$ and Δ_j.

Note that $[x]^i$ is XOR-homomorphic: given two authenticated bits $[x]^i$ and $[y]^i$, it is possible to compute the authenticated bit $[z]^i$ where $z = x \oplus y$ by simply having each party compute the XOR of the MAC / keys locally.

Authenticated secret shares. In the above construction, x is known to a single party and authenticated to the rest. Now suppose that x is *shared* amongst all parties such that no subset of parties knows x. In this case, each P_i holds x^i such that $x = \oplus_i x^i$. To authenticate x, we can use IT-MACs on each share x^i and distribute the authenticated shares $[x^i]^i$. We write $\langle x \rangle_\Delta$ to denote the collection of authenticated shares $\{[x^i]^i\}_i$ under the global keys $\Delta = \{\Delta_i\}_i$. We omit the subscript in $\langle x \rangle_\Delta$ if the global keys are clear from context. One can show that $\langle x \rangle$ is XOR-homomorphic, *i.e.*, given $\langle x \rangle$ and $\langle y \rangle$ the parties can locally compute $\langle z \rangle$ where $z = x \oplus y$.

Garbled circuits and the WRK protocol. As described in Chapter 2, garbled circuits [Yao86, BHR12, BMR90] are a commonly used cryptographic primitive in MPC constructions. Formally, an m-party garbling scheme is a pair of algorithms (Garble, Eval) that allows a secure evaluation of a (typically Boolean) circuit C. To do so, the parties first invoke Garble with C, and obtain a garbled circuit $\mathsf{G}(C)$ and some extra information (each party may obtain its own secret extra information). Then, given the input x_i to party P_i, the parties invoke Eval with $\{x_i\}_i$ and obtain the evaluation output y. (This is a simplification of a garbling scheme in many ways, but this abstraction suffices to understand the WRK protocol below.) Typically, constructions utilizing a garbling scheme are in

the *offline-online* model, in which they may invoke Garble offline when they agree on the circuit C, and only later they learn their inputs $\{x_i\}_i$ to the computation.

The WRK protocol [WRK17] is the state-of-the-art garbled circuit protocol that is maliciously-secure even when $m-1$ out of m parties are corrupted. WRK follows the same abstraction described above, with its own format for a garbled circuit; thus, we denote its garbling scheme by $(\mathsf{WRK \cdot Garble}, \mathsf{WRK \cdot Eval})$. Our construction does not modify the inner workings of the protocol; therefore, we describe only its input and output layers, but elide internal details for simplicity.

$\mathsf{WRK \cdot Garble}$: Given a Boolean circuit C, the protocol outputs a garbled circuit $\mathsf{G}(C)$. The garbling scheme authenticates the circuit by maintaining IT-MACs on all input/output wires,[1] as follows. Each party P_i obtains a global key Δ_i for the circuit. In addition, each wire w in the circuit is associated with a random "masking" bit λ_w which is output to the parties as $\langle \lambda_w \rangle_\Delta$.

$\mathsf{WRK \cdot Eval}$: The protocol is given a garbled circuit $\mathsf{G}(C)$. Then, for a party P_i who wishes to input b_w to input wire w, we have the parties input $\hat{b}_w = b_w \oplus \lambda_w$ instead; in addition, instead of receiving the real output bit b_v the parties receive a masked bit $\hat{b}_v = b_v \oplus \lambda_v$. Note that λ_w and λ_v should be kept secret from the parties (except from the party who inputs b_w or receives b_v, respectively). The procedures by which parties privately translate masked values to real values and vice versa are simple and not part of the core functionality, as we describe below.

Using the above abstractions, the overall WRK protocol is simple and can be described as follows:

1. *Offline.* The parties invoke $\mathsf{WRK \cdot Garble}$ on C and obtain $\mathsf{G}(C)$ and $\langle \lambda_w \rangle$ for every input/output wire w.

2. *Online.*

   a) *Input.* If an input wire w is associated with party P_i, who has the input bit b_w, then the parties reconstruct λ_w to P_i. Then, P_i broadcasts the bit $\hat{b}_w = b_w \oplus \lambda_w$.

   b) *Evaluation.* The parties invoke $\mathsf{WRK \cdot Eval}$ on $\mathsf{G}(C)$ and the bit $\hat{b}_w$ for every input wire w. They obtain a bit $\hat{b}_v = b_v \oplus \lambda_v$ for every output wire v.

   c) *Output.* To reveal bit b_v of an output wire v, the parties publicly reconstruct λ_v and compute $b_v = \hat{b}_v \oplus \lambda_v$.

4.4.2 Soldering wires of WRK garbled circuits

The primary technique in Senate is to securely transfer the *actual value* that passes through an output wire of one circuit, without revealing that value, to the input wire of another circuit. This action is called *soldering* [NO09]. We observe that the WRK protocol enjoys the right properties that enable soldering of its wires *almost for free*. In addition, we show how to extend the soldering notion even to cases where the set of parties who are engaged in the 'next' circuit is a superset of the set of parties engaged in the current one. This was not known until now. We believe this extension is of independent interest and may have more applications beyond Senate.

---

[1]In fact, it does so for all the wires in the circuit; we omit this detail as we focus on the input / output interface.

Specifically, we wish to securely transfer the (hidden) output $b_v = \hat{b}_v \oplus \lambda_v$ on output wire v of $\mathsf{G}(C_1)$ to the input wire u of $\mathsf{G}(C_2)$. 'Securely' means that $b_v = b_u$ should hold while keeping both b_u and b_v secret from the parties. To achieve this, the parties need to securely compute the masked value of the input to the next circuit, as expected by the WRK protocol:

$$\hat{b}_u = \lambda_u \oplus b_u = \lambda_u \oplus b_v = \lambda_u \oplus \lambda_v \oplus \hat{b}_v$$

and input it to $\mathsf{WRK} \cdot \mathsf{Eval}$ for the next circuit.

Note that the parties already hold the three terms on the right hand side of the above equation—$\mathsf{WRK} \cdot \mathsf{Eval}$ outputs $\hat{b}_v$ to the parties as a masked output when evaluating $\mathsf{G}(C_1)$, and the parties hold $\langle \lambda_v \rangle$ and $\langle \lambda_u \rangle$ as output from $\mathsf{WRK} \cdot \mathsf{Garble}$ on C_1 and C_2 respectively. Thus, one attempt to obtain $\hat{b}_u$ might be to have the parties compute the shares of $\langle \lambda_u \oplus \lambda_v \oplus \hat{b}_v \rangle$ using XOR-homomorphism, and then publicly reconstruct it. However, this operation is *not defined* unless the global key that each party uses in the constituent terms is the *same*. Since we do not modify the construction of $\mathsf{WRK} \cdot \mathsf{Garble}$ and $\mathsf{WRK} \cdot \mathsf{Eval}$, the global keys in the two circuits (and hence in $\langle \lambda_v \rangle$ and $\langle \lambda_u \rangle$) are different with high probability.

We overcome this limitation using the functionality $\mathcal{F}_{\mathsf{Solder}}$:

FUNCTIONALITY 1. $\mathcal{F}_{\mathsf{Solder}}(v, u)$ – *Soldering*

Inputs. Parties in set $\mathcal{P}_1$ agree on $\hat{b}_v$ and have $\langle \lambda_v \rangle_\Delta$ authenticated under global keys $\{\Delta_i\}_{i \in \mathcal{P}_1}$. Parties in set $\mathcal{P}_2$ (where $\mathcal{P}_1 \subseteq \mathcal{P}_2$) have $\langle \lambda_u \rangle_{\tilde{\Delta}}$ authenticated under global keys $\{\tilde{\Delta}_i\}_{i \in \mathcal{P}_2}$.
Outputs. Compute $\hat{b}_u = \lambda_u \oplus \lambda_v \oplus \hat{b}_v$. Then,

- Output $\delta_i = \Delta_i \oplus \tilde{\Delta}_i$ for all $P_i \in \mathcal{P}_1$ to parties in $\mathcal{P}_1$.

- Output $\lambda_v^i \oplus \lambda_u^i$ for all $P_i \in \mathcal{P}_1$ to parties in $\mathcal{P}_1$.

- Output λ_u^i for all $P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1$ to everyone.

- If $\langle \lambda_v \rangle_\Delta$ and $\langle \lambda_u \rangle_{\tilde{\Delta}}$ are valid then output $\hat{b}_u$ to parties in $\mathcal{P}_2$.

- Otherwise, output $\hat{b}_u$ to the adversary and $\perp$ to the honest parties.

Before proceeding, note that $\mathcal{F}_{\mathsf{Solder}}$ satisfies our needs: $\mathcal{P}_1$ and $\mathcal{P}_2$ are engaged in evaluating garbled circuits $\mathsf{G}(C_1)$ and $\mathsf{G}(C_2)$ respectively. v is an output wire of $\mathsf{G}(C_1)$, and u is an input wire of $\mathsf{G}(C_2)$. The parties in $\mathcal{P}_2$ want to transfer the actual value that passes through v, namely b_v, to $\mathsf{G}(C_2)$. That is, they want the actual value that would pass through u to be b_v as well. However, they do not know b_v, but only the masked value $\hat{b}_v$. Thus, by using $\mathcal{F}_{\mathsf{Solder}}$, they can obtain exactly what they need in order to begin evaluating $\mathsf{G}(C_2)$ with $b_u = b_v$.

Along with the soldered result $\hat{b}_u$, functionality $\mathcal{F}_{\mathsf{Solder}}$ also reveals additional information to the parties—specifically, the values of δ_i (for all $P_i \in \mathcal{P}_1$); $\lambda_v^i \oplus \lambda_u^i$ (for all $P_i \in \mathcal{P}_1$); and λ_u^i (for all $P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1$). We model this extra leakage in the functionality as this information is revealed by our protocol that instantiates $\mathcal{F}_{\mathsf{Solder}}$. However, we will show that this does not affect the security of our overall MPC protocol.

Instantiating $\mathcal{F}_{\mathsf{Solder}}$. We start by defining a procedure for XOR-ing authenticated shares under *different* global keys, which we denote $\boxplus$. That is, $\langle x \rangle_\Delta \boxplus \langle y \rangle_{\tilde{\Delta}}$ outputs $\langle x \oplus y \rangle_{\tilde{\Delta}}$.

We observe that it is possible to implement $\boxplus$ in a very simple manner: every party P_i only needs to broadcast the difference of the two global keys: $\delta_i = \Delta_i \oplus \tilde{\Delta}_i$. Using this, the parties can switch the underlying global keys of $\langle x \rangle$ from Δ_i to $\tilde{\Delta}_i$ by having each party P_i compute new authentications of x^i, denoted $M'_j[x^i]$, as follows. For every $j \neq i$, P_i computes

$$M'_j[x^i] = M_j[x^i] \oplus x^i \delta_j$$
$$= K_j[x^i] \oplus x^i \Delta_j \oplus x^i \delta_j = K_j[x^i] \oplus x^i \tilde{\Delta}_j$$

So now, x is shared and authenticated under the new global keys $\{\tilde{\Delta}_i\}_i$. Given this procedure, we can realize $\mathcal{F}_{\mathsf{Solder}}$ as follows: the parties first compute $\langle b_v \rangle_\Delta = \langle \lambda_v \rangle_\Delta \oplus \hat{b}_v$; [2] the parties then compute $\langle \hat{b}_u \rangle_{\tilde{\Delta}} = \langle b_v \rangle_\Delta \boxplus \langle \lambda_u \rangle_{\tilde{\Delta}}$, and reconstruct $\hat{b}_u$ by combining their shares.

Note that the description above (implicitly) assumes that $\mathcal{P}_1 = \mathcal{P}_2$; however, if $\mathcal{P}_1 \subset \mathcal{P}_2$ then the $\boxplus$ protocol does not make sense because parties in $\mathcal{P}_2$ that are not in $\mathcal{P}_1$ do not have a global key Δ_i corresponding to $\langle x \rangle_\Delta$. Forcing them to participate in the $\boxplus$ protocol with $\Delta_i = 0$ would result in a complete breach of security as it would reveal $\delta_i = \Delta_i \oplus \tilde{\Delta}_i = \tilde{\Delta}_i$, which must remain secret! We resolve this problem in the protocol Π_{Solder} (Protocol 1) which extends $\boxplus$ to the case where $\mathcal{P}_1 \subset \mathcal{P}_2$.

Theorem 1. *Protocol Π_{Solder} securely computes functionality $\mathcal{F}_{\mathsf{Solder}}$ (per Definition 4) in the presence of a static adversary that corrupts an arbitrary number of parties.*

We defer the proof to an extended report [PKY$^+$20].

4.4.3 Secure computation of circuit trees

Given a SQL query, Senate decomposes the query into a tree of circuits, where each *non-root* node (circuit) in the tree involves only a subset of the parties. We now describe how the soldering technique can be used to evaluate trees of circuits, while preserving the security of the overall computation. To this end, we first formalize the class of circuit trees that represent valid decompositions with respect to our protocol; then, we concretely describe our protocol for executing such trees.

We start with some preliminary definitions and notation. A *circuit tree T* is a tree whose internal nodes are circuits, and the leaves are the tree's input wires (which are also input wires to some circuit in the tree). Each node that provides input to an internal node C in the tree is a child of C. Since T is a tree, this implies that all of a child's output wires may only be fed as input to a single parent node in the tree.

We denote a circuit C's and a tree T's input wires by $\mathcal{I}(C)$ and $\mathcal{I}(T)$ respectively. Each wire $w \in \mathcal{I}(T)$ is associated with one party P_i, in which case we write $\mathsf{parties}(w) = P_i$. Let $G_1, \ldots, G_k$ be C's children, we define $\mathsf{parties}(C) = \cup_{i=1}^{k} \mathsf{parties}(G_i)$. Note that we assume, without loss of generality, that the root circuit $C \in T$ has $\mathsf{parties}(C) = \{P_1, \ldots, P_m\}$ (*i.e.*, it involves inputs from all parties). Our goal is to achieve secure computation for circuit trees; however, as discussed earlier, our construction does not support arbitrary trees. We now describe formally what can be achieved.

---

[2]XOR homomorphism works also when one literal is a constant, rather than an authenticated sharing.

PROTOCOL 1. Π_{Solder} – *Soldering*

Denote by $\langle \lambda_u^{\mathcal{P}_1} \rangle_{\tilde{\Delta}}$ the authenticated secret shares of λ_u held by parties in $\mathcal{P}_1$ only. That is $\lambda_u^{\mathcal{P}_1} = \bigoplus_{i:P_i \in \mathcal{P}_1} \lambda_u^i$.

1. The parties in $\mathcal{P}_1$ reconstruct $\langle \hat{b}_u^{\mathcal{P}_1} \rangle_{\tilde{\Delta}} = (\hat{b}_v \oplus \langle \lambda_v \rangle_{\Delta}) \boxplus \langle \lambda_u^{\mathcal{P}_1} \rangle_{\tilde{\Delta}}$.

   Specifically, each party $P_i \in \mathcal{P}_1$ broadcasts: a) the bit $\hat{b}_u^i = \lambda_v^i \oplus \lambda_u^i$, and b) the difference $\delta_i = \Delta_i \oplus \tilde{\Delta}_i$. After receiving $\hat{b}_u^j$ and δ_j from every $P_j \in \mathcal{P}_1$, it computes

$$
\begin{aligned}
\hat{b}_u^{\mathcal{P}_1} &= \hat{b}_v \oplus \bigoplus_{i:P_i \in \mathcal{P}_1} \hat{b}_u^i, \\
M_j[\hat{b}_u^i] &= M_j[\lambda_v^i \oplus \lambda_u^i] \\
&= M_j[\lambda_v^i] \oplus M_j[\lambda_u^i] \oplus \lambda_v^i \cdot \delta_j \\
&= (K_j[\lambda_v^i] \oplus \lambda_v^i \cdot \Delta_j) \oplus (K_j[\lambda_u^i] \oplus \lambda_u^i \cdot \tilde{\Delta}_j) \oplus (\lambda_v^i \cdot \delta_j) \\
&= K_j[\lambda_v^i] \oplus K_j[\lambda_u^i] \oplus \lambda_v^i \cdot (\Delta_j \oplus \delta_j) \oplus \lambda_u^i \cdot \tilde{\Delta}_j \\
&= K_j[\lambda_v^i] \oplus K_j[\lambda_u^i] \oplus (\lambda_v^i \oplus \lambda_u^i) \cdot \tilde{\Delta}_j \quad \text{and} \\
K_i[\hat{b}_u^j] &= K_i[\lambda_v^j] \oplus K_i[\lambda_u^j]
\end{aligned}
$$

   for every $j \in \mathcal{P}_1$ and broadcasts $M_j[\hat{b}_u^i]$.

2. Parties $P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1$ broadcast λ_u^i and $M_j[\lambda_u^i]$ for all $j \in \mathcal{P}_2$.

3. Parties $P_i \in \mathcal{P}_1$ verify that $K_i[\hat{b}_u^j] \oplus \hat{b}_u^j \cdot \tilde{\Delta}_i = M_i[\hat{b}_u^j]$ for all $j \in \mathcal{P}_1$.

4. Parties $P_i \in \mathcal{P}_2$ verify that $K_i[\lambda_u^j] \oplus \lambda_u^j \cdot \tilde{\Delta}_i = M_i[\lambda_u^j]$ for all $j \in \mathcal{P}_2 \setminus \mathcal{P}_1$.

5. If verification fails, output $\perp$ and abort. Otherwise, output

$$
\hat{b}_u = \left(\bigoplus_{P_i \in \mathcal{P}_2} \lambda_u^i \right) \oplus b_u = \left(\bigoplus_{P_i \in \mathcal{P}_1} \lambda_u^i \right) \oplus \left(\bigoplus_{P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1} \lambda_u^i \right) \oplus b_u = \hat{b}_u^{\mathcal{P}_1} \oplus \left(\bigoplus_{P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1} \lambda_u^i \right)
$$

Definition 5. *A circuit $C : \mathcal{D} \to \mathcal{R}$ (where $\mathcal{D} \subseteq \{0,1\}^k$ is C's domain and $\mathcal{R} \subseteq \{0,1\}^\ell$ is the range) is* invertible *if there is a polynomial time algorithm $\mathcal{A}$ (in the size of the circuit $|C|$) such that given $y \in \{0,1\}^\ell$:*

$$
\mathcal{A}(y) = \begin{cases} x \text{ such that } x \in \mathcal{D} \text{ and } C(x) = y & \text{if } y \in \mathcal{R} \\ \perp & \text{if } y \notin \mathcal{R} \end{cases}
$$

Note that in the definition above, the circuit C need not be "full range", *i.e.*, its range may be a subset of $\{0,1\}^\ell$. In such cases, we require that it is "easy" to verify that a given value $y \in \{0,1\}^\ell$ is also in $\mathcal{R}$. By easy we mean that it can be verified by a polynomial-size circuit. We also denote by $\text{ver}_C(y)$ the circuit that checks whether a value $y \in \{0,1\}^\ell$ is in $\mathcal{R}$ and returns 0 or 1 accordingly. Note that given a tree of circuits, the range of an intermediate circuit depends not only on the

circuit's computation, but also on the ranges of its children because they limit the circuit's domain. Thus, these ranges need to be deduced topologically for the tree, using which the ver_C circuit is manually crafted.

Definition 6. *For $t < m$, the class of t-admissible circuit trees, denoted $\mathcal{T}(t)$, contains all circuit trees T, such that C is invertible for all $C \in T$ where $|\text{parties}(C)| \leq t$. In addition, each circuit C that is parent to circuits $G_1, \ldots, G_k$ has $\text{ver}_{G_1}, \ldots, \text{ver}_{G_k}$ embedded within it as sub-circuits, and $\text{parties}(C) = \cup_{i=1}^{k} \text{parties}(G_i)$.*

The above suggests that there *may* indeed be non-invertible circuits (*e.g.*, a preimage resistant hash) in the tree; the only restriction is that such a circuit should be evaluated by more than t parties. The definition of MPC for circuit trees follows the general definition of MPC [Gol04b], as presented below.

FUNCTIONALITY 2. $\mathcal{F}_{\text{MPC·tree}}$ *– MPC for circuit trees*

Parameters. A circuit tree T and parties $P_1, \ldots, P_m$.
Inputs. For each $w \in \mathcal{I}(T)$ where $P_i = \text{parties}(w)$, wait for an input bit b_w from P_i.
Outputs. The bit b_w for every w in T's output wires, given by evaluating T in a topological order from leaves to root.

We realize $\mathcal{F}_{\text{MPC·tree}}$ using the protocol $\Pi_{\text{MPC·tree}}$ (Protocol 2), which is our overall protocol for securely executing circuit trees. The protocol works as follows. In the offline phase the parties

---

PROTOCOL 2. $\Pi_{\text{MPC·tree}}$ - *MPC for circuit trees*

Parameters. The circuit tree T. Parties $P_1, \ldots, P_m$.
Inputs. For $w \in \mathcal{I}(T)$, $P_i = \text{parties}(w)$ has $b_w \in \{0, 1\}$.
Protocol.

1. *Offline.* For every circuit $C \in T$, $\text{parties}(C)$ run $\text{WRK} \cdot \text{Garble}(C)$ to obtain $\text{G}(C)$ along with $\langle \lambda_w \rangle$ for all input and output wires w.

2. *Online.* For each circuit C in T (topologically) do:

   a) *Input.* For every $u \in \mathcal{I}(C)$: If $u \in \mathcal{I}(T)$ and $P_i = \text{parties}(u)$ then $\text{parties}(C)$ reconstruct λ_u to P_i. Else, if u is connected to an output wire v of a child circuit C' then run $\mathcal{F}_{\text{Solder}}(v, u)$, by which $\text{parties}(C)$ obtain $\hat{b}_u$.

   b) *Evaluate.* Run $\text{WRK} \cdot \text{Eval}$ on $\text{G}(C)$ and $\hat{b}_u$ for every $u \in \mathcal{I}(C)$, by which $\text{parties}(C)$ obtain $\hat{b}_v$ for every C's output wire v. If $G_1, \ldots, G_c$ are C's children then abort if an intermediate value $\text{ver}(G_i) = 0$ for some $i \in [c]$.

   c) *Output.* If C is the root of T, reconstruct $\langle \lambda_w \rangle$ for every $w \in \mathcal{O}(C)$, by which all parties obtain $b_w = \hat{w} \oplus \lambda_w$.

---

simply garble all circuits using $\text{WRK} \cdot \text{Garble}$; each circuit is garbled independently from the others.

Then, beginning from the tree's leaf nodes, the parties evaluate the circuits using WRK · Eval, such that each circuit C is evaluated only by parties(C) (not all the parties). When a value on an output wire of some circuit C' should travel privately to the input wire of the next circuit C then parties(C) run the soldering protocol. As discussed above, parties(C') may be a subset of parties(C). Once all the nodes have been evaluated, the parties operate exactly as in the WRK protocol in order to reveal the actual value on the output wire.

We prove the security of protocol $\Pi_{\mathsf{MPC \cdot tree}}$ per the following theorem in an extended report [PKY$^+$20]. We remark that our protocol inherits the random oracle assumption from its use of the WRK protocol.

Theorem 2. *Let $t < m$ be the number of parties corrupted by a static adversary. Then, protocol $\Pi_{\mathsf{MPC \cdot tree}}$ securely computes $\mathcal{F}_{\mathsf{MPC \cdot tree}}$ (per Definition 4) for any $T \in \mathcal{T}(t)$, in the random oracle model and the $\mathcal{F}_{\mathsf{Solder}}$-hybrid model.*

We stress that intermediate values (output wires of internal nodes) are authenticated secret shares, each using fresh randomness, and thus kept secret from the adversary. In particular, the adversary's input is independent of these values.

Note that by our construction, if there is a sub-tree rooted at a circuit C such that parties(C) are all corrupted, then the adversary may skip the 'secure computation' of that sub-tree and simply provide inputs directly to C's parent. This, however, does not form a security issue because a malicious adversary may change its input anyway, and the sub-tree is invertible—hence, whatever input is given to C's parent, it can be used to extract *some* possible adversary's input to the tree's input wires (and hence to the functionality) that leads to the target output from the functionality.

In the following sections, we describe how Senate executes SQL queries by transforming them into circuit trees that can be securely executed using our protocol.

4.5 Senate's Circuit Primitives

Senate executes a query by first representing it as a tree of Boolean circuits, and then processing the circuit tree using its efficient MPC protocol. To construct the circuits, Senate uses a small set of circuit primitives which we describe in turn. In later sections, we describe how Senate composes these primitives to represent SQL operations and queries.

4.5.1 Filtering

Our first building block is a simple circuit (Filter) that takes a list of elements as input, and passes each element through a sub-circuit that compares it with a specified constant. If the check passes, it outputs the element, else it outputs a zero.

4.5.2 Multi-way set intersection

Next, we describe a circuit for computing a *multi-way* set intersection. Prior work has mainly focused on designing Boolean circuits for two-way set intersections [HEK12, BA12]; here we design optimized circuits for intersecting multiple sets. Our circuit extends the two-way SCS circuit of Huang *et al.* [HEK12]. We start by providing a brief overview of the SCS circuit, and then describe how we extend it to multiple sets.

The two-way set intersection circuit (2-SI). The sort-compare-shuffle circuit of Huang *et al.* [HEK12] takes as input two sorted lists of size n each with *unique* elements, and outputs a list of size n containing the intersection of the lists interleaved with zeros (for elements that are not in the intersection). (1) The circuit first merges the sorted lists. (2) Next, it filters intersecting elements by comparing adjacent elements in the list, producing a list of size n that contains all filtered elements interleaved with zeros. (3) Finally, it shuffles the filtered elements to hide positional information about the matches.

In Senate's use cases, set intersection results are often not the final output of an MPC computation, and are instead intermediate results upon which further computation is performed. In such cases, the shuffle operation is not performed.

A multi-way set intersection circuit (*m*-SI). Suppose we wish to compute the intersection over three sets A, B and C. A straightforward approach is to compose two 2-SI circuits together into a larger circuit (*e.g.*, as 2-SI(2-SI(A, B), C)). However, such an approach doesn't work out-of-the-box because the intermediate output $O = $ 2-SI(A, B) needs to be sorted before it can be intersected with C, as expected by the next 2-SI circuit. While one can accomplish this by sorting the output, it comes at the cost of an extra $O(n \log^2 n)$ gates.

Instead of performing a full-fledged sort, we exploit the observation that, essentially, the output O of 2-SI is the *sorted* result of $A \cap B$ interleaved with zeros. So, we transform O into a *sorted multiset* via an intermediate *monotonizer* circuit Mono that replaces each zero in O with the nearest preceding non-zero value. Concretely, given $O = (a_1 \dots a_n)$ as input, Mono outputs $M = (b_1 \dots b_n)$, such that $b_i = a_i$ if $a_i \neq 0$, else $b_i = b_{i-1}$. For example, if $O = (1, 0, 2, 3, 0, 4)$, then Mono converts it to $M = (1, 1, 2, 3, 3, 4)$.

Since M now also contains duplicates, for correctness of the overall computation, the next 2-SI that intersects M with C needs to be able to discard these duplicates. We therefore modify the next 2-SI circuit: (i) the circuit tags a bit to each element in the input lists that identifies which list the element belongs to, *i.e.*, it appends 0 to every element in the first list, and 1 to every element in the second; (ii) the comparison phase of the circuit additionally verifies that elements with equal values have different tags. These modifications ensure that duplicates in the same intermediate list aren't added to the output. We refer to this modified 2-SI circuit as 2-SI∗.

The described approach generalizes to multiple input sets in an identical manner. Note that in general, there can be many ways of constructing the binary tree of 2-SI circuits (*e.g.*, a left-deep vs. balanced tree). In Section 4.7 we describe how Senate's compiler picks the optimal design when executing queries.

4.5.3 Multi-way sort

Given m sorted input lists of size n each, a multi-way sort circuit m-Sort merges the lists into a single sorted list of size $m \times n$, using a binary tree of bitonic merge operations (implemented as the Merge circuit).

4.5.4 Multi-way set union

Our next building block is a circuit for multi-way set unions. In designing the circuit, we extend the two-way set union circuit of Blanton and Aguiar [BA12].

The two-way set union circuit (2-SU). Given two sorted input lists of size n each with unique elements, the 2-SU circuit produces a list of size $2n$ containing the set union of the inputs. Blanton and Aguiar [BA12] proposed a 2-SU circuit similar to 2-SI: (1) It first merges the input lists into a single sorted list. (2) Next, it removes duplicate elements from the list: for every two consecutive elements e_i and e_{i+1}, if $e_i \neq e_{i+1}$ it outputs e_i, else it outputs 0. (3) Finally, the circuit randomly shuffles the filtered elements to hide positional information.

A multi-way set union circuit (m-SU). It might be tempting to construct a multi-way set union circuit by composing multiple 2-SU circuits together, similar to m-SI. However, such an approach is sub-optimal: unlike the intersection case where intermediate lists remain size n, in unions the intermediate result size grows as more input lists are added. This leads to an unnecessary duplication of work in subsequent circuits. Instead, we construct a multi-way analogue of the 2-SU circuit, as follows: (1) We first merge all m input lists together into a single sorted list using an m-Sort circuit. (2) We then remove duplicate elements from the sorted list, in a manner identical to 2-SU. We refer to the de-duplication sub-circuit in m-SU as Dedup. The m-SU circuit may thus alternately be expressed as a composition of circuits: Dedup $\circ$ m-Sort.

4.5.5 Input verification

Our description of the circuits thus far (m-SI, m-SU, and m-Sort) assumes that their inputs are sorted. While this assumption is safe in the case of semi-honest adversaries, it fails in the presence of malicious adversaries who may arbitrarily deviate from the MPC protocol. For malicious security, we need to additionally *verify* within the circuits that the inputs to the circuit are indeed sorted sets. To this end, we augment the circuits with *input verifiers* Ver, that scan each input set comparing adjacent elements e_i and e_{i+1} in pairs to check if $e_{i+1} > e_i$ for all i; if so, it outputs a 1, else 0. When a given circuit is augmented with input verifiers, it additionally outputs a logical AND over the outputs of all constituent Ver circuits. This enables all parties involved in the computation to verify that the other parties did not cheat during the MPC protocol.

4.6 Decomposable Circuits for SQL Operators

Given a SQL query, Senate decomposes it into a tree of SQL operations and maps individual operations to Boolean circuits. For some operations—namely, joins, group-by, and order-by operations—the Boolean circuits can be *further decomposed* into a tree of sub-circuits, which results in greater efficiency. In this section, we show how Senate expresses individual SQL operations as circuits using the primitives described in Section 4.5, decomposing the circuits further when possible. Later in Section 4.7, we describe the overall algorithm for transforming queries into circuit trees and executing them using our MPC protocol.

Notation. We express Senate's transformation rules using traditional relational algebra [Cod70], augmented with the notion of parties to capture the collaborative setting. Let $\{P_1, \ldots, P_m\}$ be the set of parties in the collaboration. Recall that we write $R|P_i$ to denote a relation R (*i.e.*, a set of rows) held by P_i. We also repurpose $\cup$ to denote a simple concatenation of the inputs, as opposed to the set union operation. The notation for the remaining relational operators are as follows: σ filters the input; τ performs a sort; $\bowtie$ is an equijoin; and γ is group-by.

4.6.1 Joins

Consider a collaboration of m parties, where each party P_i holds a relation R_i and wishes to compute an m-way join:

$$\bowtie (R_1|P_1, \ldots, R_m|P_m)$$

Senate converts *equijoin* operations—joins conditioned on an equality relation between two columns—to set intersection circuits. Specifically, Senate maps an m-way equijoin operation to an m-SI circuit. For all other types of join operations, such as joins based on column comparisons or compound logical expressions, Senate expresses the join using a simple Boolean circuit that performs a series of operations per pairwise combination of the inputs. However, a recent study [JNS18] notes that the vast majority of joins in real-world queries (76%) are equijoins. Thus, a majority of join queries can benefit from our optimized design of set intersection circuits.

Decomposing joins across parties. If parties don't care about privacy, the simplest way to execute the join would be to perform a series of 2-way joins in the form of a tree. For example, one way to evaluate a 4-way join is to order the constituent joins as $((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4))$. To mimic this decomposition, Senate starts by designing an m-SI Boolean circuit to compute the operation (with $m = 4$). Senate then evaluates the m-SI circuit by decomposing it into its constituent sub-circuits as follows:

1. First, each party locally sorts its input sets (as required by the m-SI circuit).

2. Next, parties P_1 and P_2 jointly compute a 2-SI operation over R_1 and R_2, followed by the monotonizer Mono. In parallel, parties P_3 and P_4 compute a similar circuit over R_3 and R_4. The 2-SI circuits are augmented with Ver sub-circuits that verify that the input sets are sorted.

3. Finally, all four parties evaluate a 2-SI∗ circuit over the outputs of the previous step; as before, the circuit includes a Ver sub-circuit to check that the inputs are sorted. Note that though the evaluated circuit takes two sets as input, the circuit computation involves all four parties.

In general, multiple tree structures are possible for decomposing an m-way join. Senate's compiler (which we describe in Section 4.7) derives the best plan for the query using a cost model.

Joins over multisets.   Senate's m-SI circuit can be extended to support joins over multisets in a straightforward manner. We defer the details to Appendix A.

4.6.2   Order-by limit

In the collaborative setting, the m parties may wish to perform an order-by operation (by some column c) on the union of their results, optionally including a limit l:
$$\tau_{c,l}(\cup_i R_i | P_i)$$
Senate maps order-by operations directly to the m-Sort circuit. If the operation includes a limit l, then the circuit only outputs the wires corresponding to the first l results.

Recall from Section 4.5.3 that m-Sort is a composition of Merge sub-circuits (that perform bitonic merge operations). If the operation includes a limit l, then we make an optimization that reduces the size of the overall circuit. We note that since the circuit's output only contains wires corresponding to the first l elements of the sorted result, any gates that do not impact the first l elements can be discarded from the circuit. Hence, if an element is outside the top l choices for any intermediate Merge, then we discard the corresponding gates.

Decomposing order-by across parties.   Since the m-Sort circuit is composed of a tree of Merge sub-circuits, it can be straightforwardly decomposed across parties by distributing the constituent Merge sub-circuits. For example, one way to construct a 4-party sort circuit is: $\mathsf{Merge}(\mathsf{Merge}(R_1, R_2), \mathsf{Merge}(R_3, R_4))$. To decompose this:

1. Each party first sorts their input locally (as expected by the m-Sort circuit).

2. Parties P_1 and P_2 compute a Merge sub-circuit; P_3 and P_4 do the same in parallel.

3. All 4 parties finally Merge the previous outputs.

Once again, multiple tree structures are possible for distributing the Merge circuits, and the Senate compiler's planning algorithm picks the best structure based on a cost model.

4.6.3   Group-by with aggregates

Suppose the parties wish to compute a group-by operation over the union of their relations (on some column c), followed by an aggregate Σ per group:
$$\gamma_{c,\Sigma}(\cup_i R_i | P_i)$$
Senate starts by mapping the operator to a $\Sigma \circ m$-SU circuit that computes the aggregate function $\Sigma = \mathsf{SUM}$. To do so, we extend the m-SU circuit with support for aggregates. Recall from Section 4.5.4 that the m-SU circuit is a composition of sub-circuits $\mathsf{Dedup} \circ m$-Sort.

Let the input to the group-by operation be a list of tuples of the form $t_i = (a_i, b_i)$, such that the a_i values represent the columns over which groups are made, and the b_i values are then aggregated per group.

1. In the *m*-Sort phase, Senate evaluates the *m*-Sort sub-circuit over the a_i values per tuple, while ignoring b_i.

2. In the Dedup phase, for every two consecutive tuples (a_i, b_i) and (a_{i+1}, b_{i+1}), the circuit outputs (a_i, b_i) if $a_i \neq a_{i+1}$, else it outputs $(0, b_i)$

3. In addition, we augment the Dedup phase to compute aggregates over the b_i values. The circuit makes another pass over the tuples (a'_i, b_i) output by Dedup while maintaining a running aggregate agg: if $a'_i = 0$ then it updates agg with b_i and outputs $(0,0)$; otherwise, it outputs (a'_i, agg).

Decomposing group-by across parties. Senate decomposes group-by operations in two ways. First, group-by operations with aggregates can typically be split into two parts: local aggregates per party, followed by a joint group-by aggregate over the union of the results. This is a standard technique in database theory. For example, suppose $\Sigma = \texttt{COUNT}$. In this case, the parties can first compute local counts per group, and then evaluate a joint sum per group over the local results. Rewriting the operation in this manner helps Senate reduce the amount of joint computation performed using a circuit, and is thus beneficial for performance.

Second, we note that the joint group-by computation can be further decomposed across parties. Specifically, the *m*-Sort phase of the overall *m*-SU circuit (as described above) can also be distributed across the parties in a manner identical to order-by (as described in Section 4.6.2).

4.6.4 Filters and Projections

Filtering is a common operation in queries (*i.e.*, the `WHERE` clause in SQL), and parties in a collaboration may wish to compute a filter on the union of their input relations:
$$\sigma_f(\cup_i R_i | P_i)$$
where f is the condition for filtering. Senate maps the operation to a Filter circuit. Filtering operations at the start of a query can be straightforwardly distributed by evaluating the filter locally at each party, before performing the union.

As regards projections, typically, these operations simply exclude some columns from the relation. Given a relation, Senate performs a projection by simply discarding the wires corresponding to the non-projected columns.

4.7 Query Execution

We now describe how Senate executes a query by decomposing it into a tree of circuits. In doing so, Senate's compiler ensures that the resulting tree satisfies the requirements of our MPC protocol (per Definition 6)—namely, that each circuit in the tree is invertible.
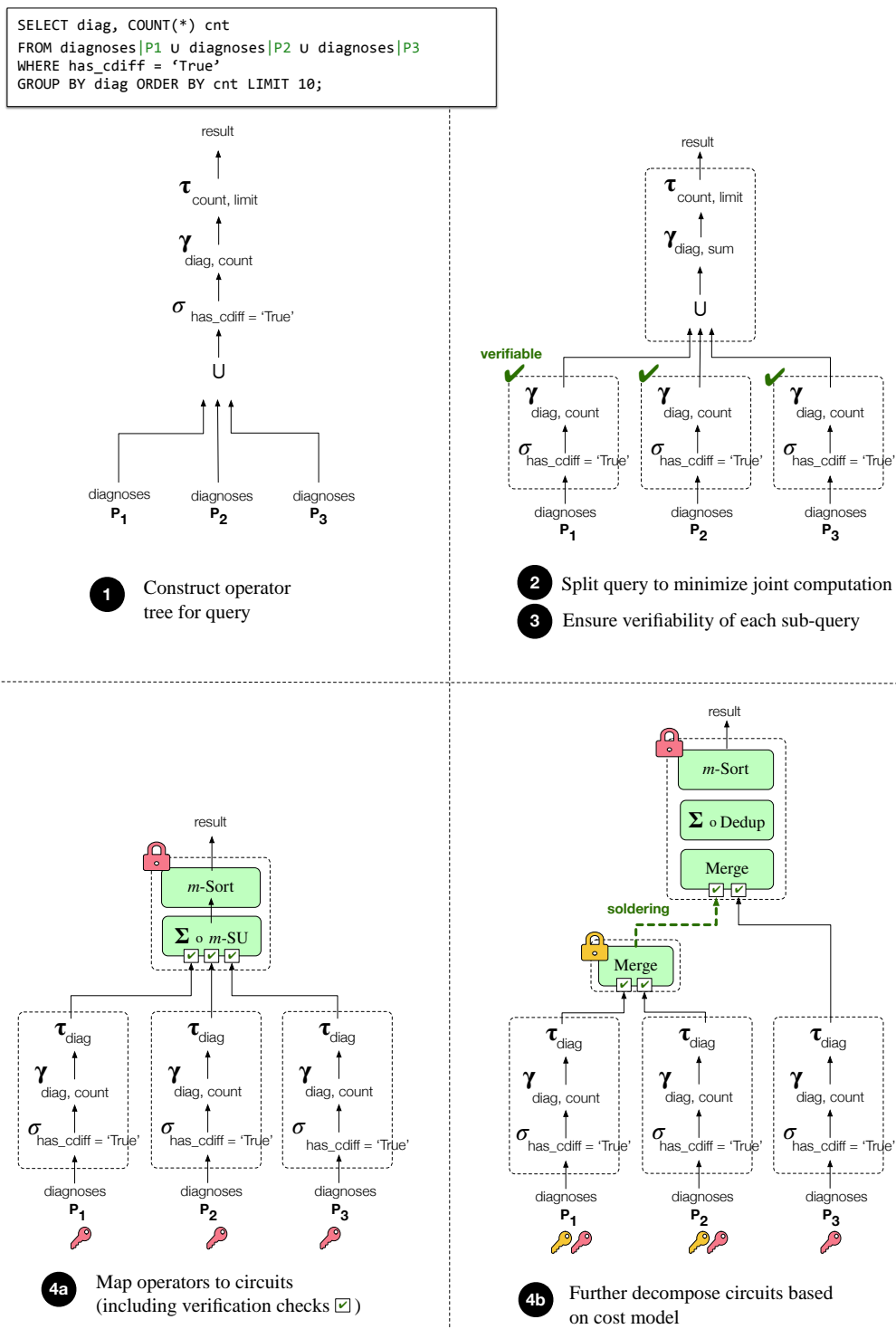
Figure 4.3: Query execution in Senate. Colored keys and locks indicate which parties are involved in which MPC circuits.

4.7.1   Query decomposition and planning

We start by describing the Senate compiler's query decomposition algorithm. Given a query, the compiler transforms the query into a circuit tree in four steps, as illustrated in Figure 4.3. We use the medical query from Section 4.1.1 as a running example.

Step ❶ : Construction of tree of operators.   Senate first represents the query as a *tree* of relational operations. The leaves of the tree are the input relations of individual parties, and the root outputs the final query result. Each non-leaf node represents an operation that will be jointly evaluated only by the parties whose data the node takes as input. Thus, the set of parties evaluating a node is always a superset of its children.

While a query can naturally be represented as a directed acyclic graph (DAG) of relational operators, Senate recasts the DAG into a tree to satisfy the *input consistency* requirements of our MPC protocol. Specifically, Senate ensures that the outputs of no intermediate node (or the input tables at the leaves) are fed to more than one parent node. This is because in such cases, if any two parents are evaluated by disjoint sets of parties, then this leads to a potential input inconsistency— that is, if all the parties at the current node collude, then there is no guarantee that they provide the same input to both parents. A tree representation resolves this problem.

Figure 4.3 illustrates the query tree for the medical query and comprises the following sequence of operator nodes—the input tables of the parties (in the leaves) are first concatenated into a single relation which is then processed jointly using a filter, a group-by aggregate, and an order-by limit operator.

Step ❷ : Query splitting.   Next, Senate logically rewrites the query tree, splitting it such that the parties perform as much computation as possible locally over their plaintext data, (*i.e.*, filters and aggregates), thereby reducing the amount of computation that need to be performed jointly using MPC. To do so, it applies traditional relational equivalence rules that (i) push down selections past joins and unions, and (ii) decomposes group-by aggregates into local aggregates followed by a joint aggregate.

For example, as shown in Figure 4.3, Senate rewrites the medical query in both these ways. Instead of performing the filtering jointly (after concatenating the parties' inputs), Senate pushes down the filter past the union and parties apply it locally. In addition, it further splits the group-by aggregate—parties first compute local counts per group, and the local counts are jointly summed up to get the overall counts.

Though such an approach has also been explored in prior work [BEE⁺17,VSG⁺19], an important difference in Senate is that while prior approaches assume a semi-honest threat model, Senate targets security against malicious adversaries who may arbitrarily deviate from the specified protocol. To protect against malicious behavior, Senate's split is different than the semi-honest split; Senate performs two actions: (i) additionally verifies that all local computations are valid; and (ii) ensures that the splitting does not introduce input consistency problems. We describe how Senate tackles these issues next.

Step ❸ : Verifying intermediate operations.   We need to take a couple of additional steps before we can execute the tree of operations securely using our MPC protocol. As Section 4.4.3 points out, to be maliciously secure, the tree of circuits needs to be "admissible" (per Definition 6), *i.e.*, each

intermediate operation in the tree must be invertible, and each intermediate node must also be able to verify that the output produced by its children is possible given the query.

Thus, in transforming a query to a circuit tree, Senate's compiler deduces the set of outputs each intermediate operation can produce, while ensuring the operation is invertible. For example, a filter of the type "WHERE 5 < age < 10" requires that in all output records, each value in column age must be between 5 and 10. Note that the values of intermediate outputs also vary based on the set of preceding operations. For more complex queries, the constraints imposed by individual operators accumulate as the query tree is executed.

Senate's compiler traverses the query tree upwards from the leaves to the root, and identifies the constraints at every level of the tree. For simplicity, we limit ourselves to the following types of constraints induced by relational operators: (i) each column in a relation can have range constraints of the type $n_1 \leq a \leq n_2$, where n_1 and n_2 are constants; (ii) the records are ordered by a single column; or (iii) the values in a column are distinct. If the cumulative constraints at an intermediate node in the tree are limited to the above, then Senate's compiler marks the node as *verifiable*. If a node produces outputs with different constraints, then the compiler marks it as *unverifiable*—for such nodes, Senate *merges* the node with its parent into a single node and proceeds as before.

If a node / leaf feeds input to more than one parent (perhaps as a result of the query rewriting in the previous step), then the compiler once again merges the node and all its parents into a single node, in order to avoid input consistency problems.

At the end of the traversal, the root node is the only potentially unverifiable node in the tree, but this does not impact security. Since all parties compute the root node jointly, the correctness of its output is guaranteed.

As an example, in Figure 4.3, the local nodes at every party locally evaluate the filter $\sigma_{\text{has\_cdiff=True}}$, which constrains the column has_cdiff to the value 'True', and satisfies condition (i) above. The subsequent group-by aggregate operation $\gamma_{\text{diag,count}}$ does not impose any constraint on either diag or count (since parties are free to provide inputs of their choice, assuming there are no constraints on the input columns). The local nodes are thus marked verifiable. All remaining operations are performed jointly by all parties at the root node, and thus do not need to be checked for verifiability.

In Appendix B, we work out in detail how Senate's compiler deduces the range constraints imposed by various relational operations (*i.e.*, what needs to be verified). Then, we show the invertibility of relational operations given these constraints. This ensures that the resulting tree is admissible, and satisfies the requirements of Senate's MPC protocol.

Step ④ : Mapping operators to circuits. The final step is to map each jointly evaluated node in the query tree to a circuit (per Section 4.6): σ maps to the Filter circuit, $\bowtie$ maps to *m*-SI, group-by aggregate maps to $\Sigma \circ m$-SU, and order-by-limit maps to *m*-Sort. In doing so, Senate's compiler uses a planning algorithm that *further decomposes* each circuit into a tree of circuits based on a cost model (described shortly).

For example, for the medical query in Figure 4.3, Senate maps the group-by aggregate operation $\gamma_{\text{diag,sum}}$ to a $\Sigma \circ m$-SU circuit. Note that *m*-SU requires its inputs to be sorted; therefore, the compiler augments the children nodes with sort operations τ_{diag}. It then further decomposes the *m*-Sort phase of *m*-SU into a tree of Merge sub-circuits, per Section 4.6.3.

This tree of circuits is finally evaluated securely using our MPC protocol. Note that at each node, only the parties that provide the node input are involved in the MPC computation.

4.7.2 Cost model for circuit decomposition

The planning algorithm models the latency cost of evaluating a circuit tree in terms of the constituent cryptographic operations. It then enumerates possible decomposition plans, assigns a cost to each plan, and picks the optimal plan for decomposing the circuit.

Recall from Section 4.4 that the cost of executing a circuit via MPC can be divided into an offline phase (for generating the circuits), and an online phase (for evaluating the circuits). Given a circuit tree T, let the root circuit be C with children C_0 and C_1. Let T_0 and T_1 refer to the subtrees rooted at nodes C_0 and C_1 respectively. Then, Senate's compiler models the total latency cost $\mathcal{C}$ of evaluating T as:

$$\mathcal{C}(T) = \max(\mathcal{C}(T_0), \mathcal{C}(T_1)) + \max(\mathcal{C}_{\text{solder}}(T_0), \mathcal{C}_{\text{solder}}(T_1))$$
$$+ \mathcal{C}_{\text{offline}}(C) + \mathcal{C}_{\text{online}}(C)$$

Essentially, since subtrees can be computed in parallel, the cost model counts the maximum of these two costs, followed by the cost of soldering the subtrees with the root node. It adds this to the cost of the offline and online phases for T's root circuit C, $\mathcal{C}_{\text{offline}}$ and $\mathcal{C}_{\text{online}}$ respectively.

We break down each cost component in terms of two unit costs by examining the MPC protocol: the unit computation cost L_s of performing a single symmetric key operation, and the unit communication cost $L_{i,j}$ (pairwise) between parties P_i and P_j. Senate profiles these unit costs during system setup. In addition, the costs also depend on the size of the circuit being computed $|C|$ (*i.e.*, the number of gates in the circuit), the size of each party's input set $|I|$, and the number of parties m computing the circuit. For simplicity, the analysis below assumes that each party has identical input set size; however, the model can be extended in a straightforward manner to accommodate varying input set sizes as well.

The soldering cost $\mathcal{C}_{\text{solder}}$ can be expressed as $(m-1)|I| \cdot \max_{i,j}(L_{i,j})$ (since it involves a single round of communication between all parties). Next, we analyze the WRK protocol to obtain the following equations:

$$\mathcal{C}_{\text{offline}}(C) = (m-1)|C| \cdot \max(L_{i,j}) + 4|C| \cdot L_s + |C| \cdot \max(L_{1,i})$$

In more detail, in the offline phase, each party (in parallel with the others) communicates with the $m-1$ other parties to create a garbled version of each gate in the circuit; each gate requires 4 symmetric key operations (one per row in the truth table representing the gate); they then send their individual garbled gates (in parallel) to the evaluator. Our analysis here is a simplification in that we ignore the cost of some function-independent preprocessing steps from the offline phase. This is because these steps are independent of the input query, and thus do not lie in the critical path of query execution.

Similarly, the cost of the online phase can be expressed as

$$\mathcal{C}_{\text{online}}(C) = (m-1)|I| \cdot \max(L_{i,j})$$
$$+ (m-1)|I| \cdot \max(L_{1,i}) + (m-1)|C| \cdot L_s$$

(a) *m*-Sl of 1*K* inputs/party.

(b) *m*-Sl with 16 parties.
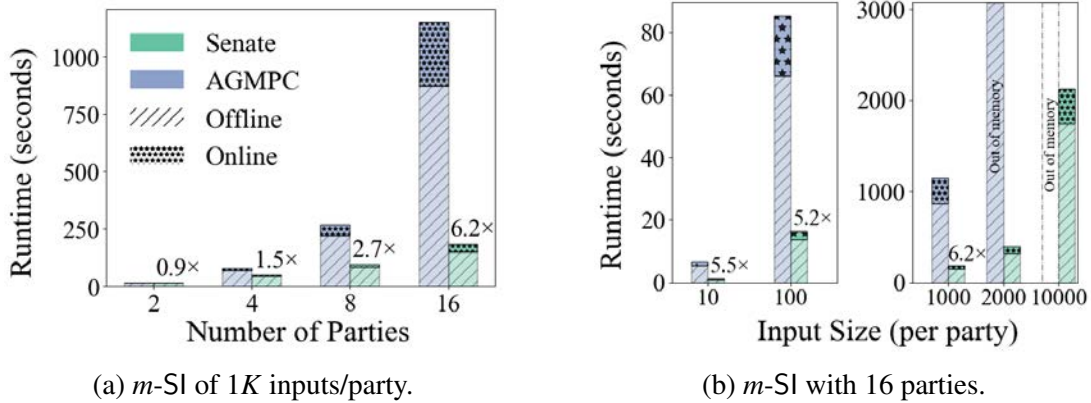
Figure 4.4: Performance of *m*-Sl in LAN.

In this phase, the garblers communicate with all other parties to compute and send their encrypted inputs to the evaluator; in addition, the evaluator communicates with each garbler to obtain encrypted versions of its own inputs. The evaluator then evaluates the gates per party. The size of the circuit $|C|$ depends on the function that the circuit evaluates (per Section 4.5), the number of inputs, and the bit length of each input.

4.8   Evaluation

In this section, we demonstrate Senate's improvements over running queries as monolithic cryptographic computations. We use vanilla AGMPC (with monolithic circuit execution) as the baseline. The highlights are as follows. On the set of representative queries from Section 4.2, we observe runtime improvements of up to $10\times$ of Senate's building blocks, with a reduction in resource consumption of up to $11\times$. These results translate into runtime improvements of up to $10\times$ for the joint computation in the benchmarked queries. Senate's query splitting technique provides a further improvement of up to $10\times$, bringing the net improvement to over $100\times$. Furthermore, on the TPC-H analytics benchmark [TPCb], Senate's improvements range from $3\times$ to $145\times$.

Implementation.   We implemented Senate on top of the AGMPC framework [EMP], a state-of-the-art implementation of the WRK protocol [WRK17] for *m*-party garbled circuits with malicious security. Our compiler works with arbitrary bit lengths for inputs; in our evaluation, we set the data field size to be integers of 32 bits, unless otherwise specified.

Experimental Setup.   We perform our experiments using r5.12xlarge Amazon EC2 instances in the Northern California region. Each instance offers 48 vCPUs and 384 GB of RAM, and was additionally provisioned with 20 GB of swap space, to account for transient spikes in memory requirements. We allocated similar instances in the Ohio, Northern Virginia and Oregon regions for wide-area network experiments.

(a) *m*-Sort of 600 inputs/party.

(b) *m*-Sort with 16 parties.
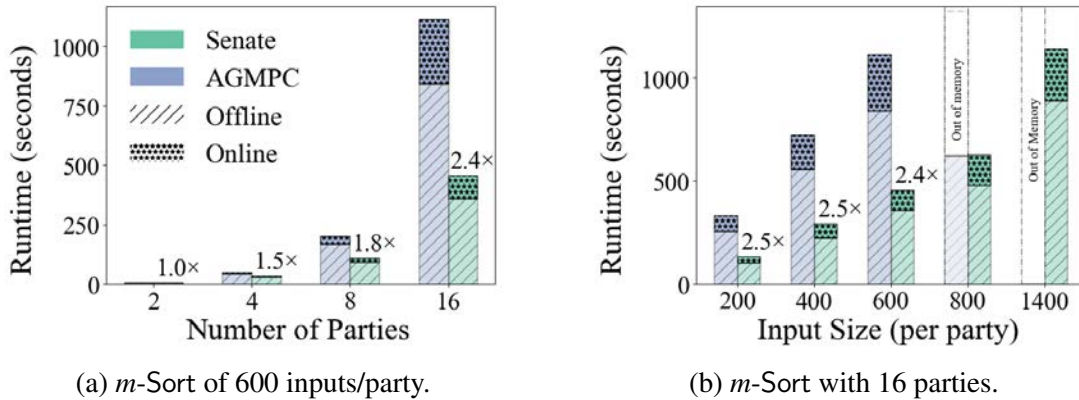
Figure 4.5: Performance of *m*-Sort in LAN.

(a) *m*-SU of 600 inputs/party.

(b) *m*-SU with 16 parties.
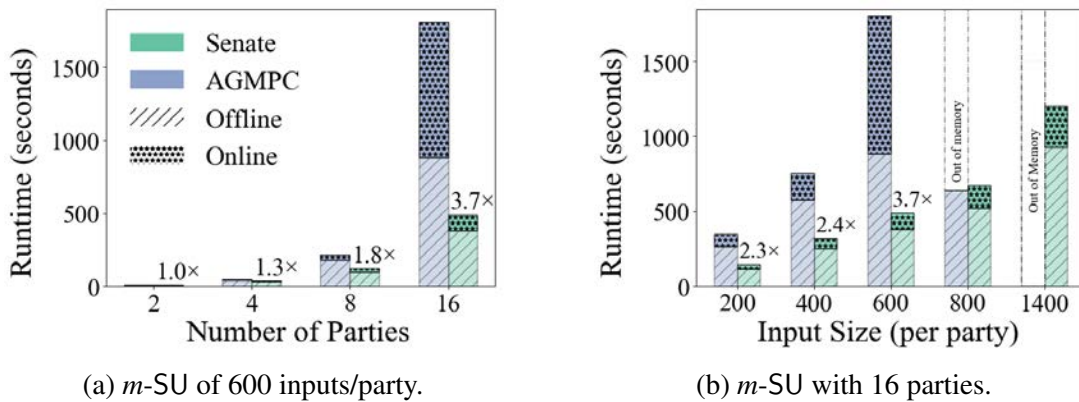
Figure 4.6: Performance of *m*-SU in LAN.

4.8.1 Senate's building blocks

We evaluate Senate's building blocks described in Section 4.5—*m*-SI, *m*-Sort, and *m*-SU. For each building block, we compare the runtimes of each phase of the computation of Senate's efficient primitives to a similar implementation of the operator as a single circuit in both LAN and WAN settings (Figures 4.4 to 4.6, and Figure 4.8). We observe substantial improvements for our operators owing to reduced number of parties evaluating each sub-circuit and the evaluation of various such circuits in parallel (per Section 4.6). We also measure the improvement in resource consumption due to Senate in Figure 4.7.

Multi-way set intersection circuit (*m*-SI). We compare the evaluation time of an *m*-SI circuit across 16 parties with varying input sizes in Figure 4.4b and observe runtime improvements ranging from $5.2\times$–$6.2\times$. This is because our decomposition enables the input size to stay constant for each sub-computation, allowing us to reduce the input set size to the final 16-party computation. Note that, while Senate can compute a set intersection of 10*K* integers, AGMPC is unable to compute it for 2*K* integers, and runs out of memory during the offline phase. Figures 4.4a and 4.8 plot the

(a) Peak memory usage

(b) Network usage
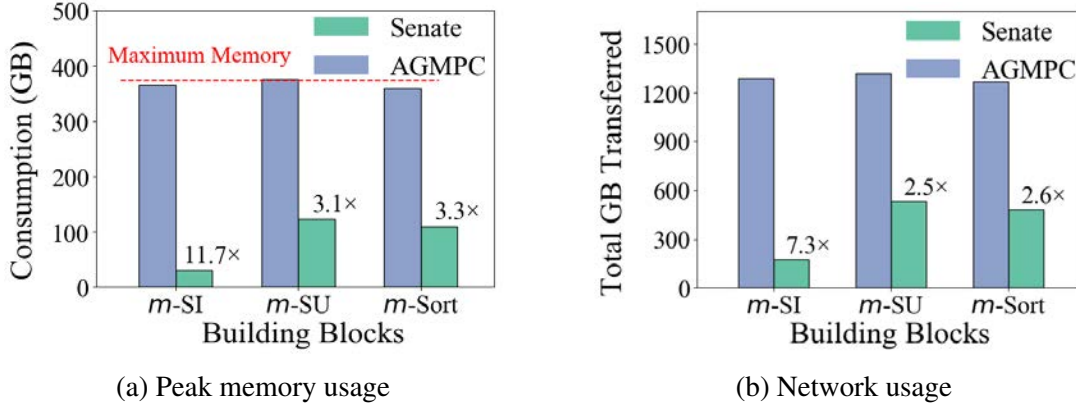
Figure 4.7: Resource consumption of building blocks (16 parties).
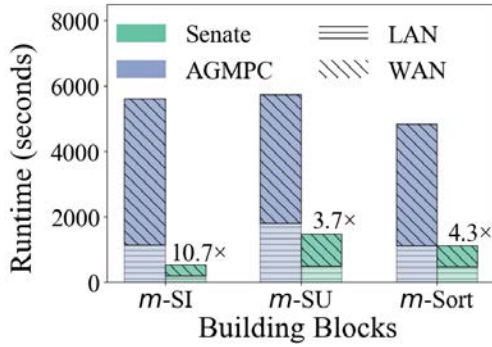
Figure 4.8: Building blocks in WAN.
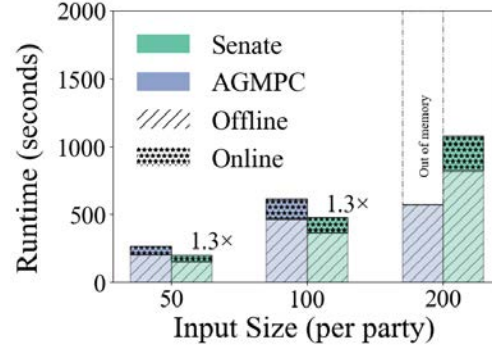
Figure 4.9: Query 1 with 16 parties.

runtime of a circuit with varying number of parties in LAN and WAN settings respectively, and observe an improvement of up to $10\times$. This can be similarly attributed to our decomposable circuits, which reduce the data transferred across all the parties, leading to significant improvements in the WAN setting.

Figures 4.7a and 4.7b plot the trend of the peak memory and total network consumption of Senate compared to AGMPC with $1K$ integers across varying number of parties.

Multi-way Sort circuit (*m*-Sort). Figures 4.5a and 4.5b illustrate the runtimes of a sorting circuit with varying number of parties and varying input sizes respectively. We observe that Senate's implementation is up to $4.3\times$ faster for 16 parties, and can scale to twice as many inputs as AGMPC. This is also corroborated by the $3.3\times$ reduction in peak memory requirement for 600 integers and $\sim 780\,\text{GB}$ reduction in the amount of data transferred, as shown in Figures 4.7a and 4.7b.

Multi-way set union circuit (*m*-SU). Figure 4.6b plots the runtime of a set union circuit with varying input sizes and 16 parties. As discussed in Section 4.5, an *m*-SU circuit can be expressed as Dedup $\circ$ *m*-Sort. Hence, we expect to trends similar to the *m*-Sort circuit. However, we observed a stark increase in runtime for the single circuit evaluation of 600 integers across 16 parties due to the exhaustion of the available memory in the system and subsequent use of swap space (see
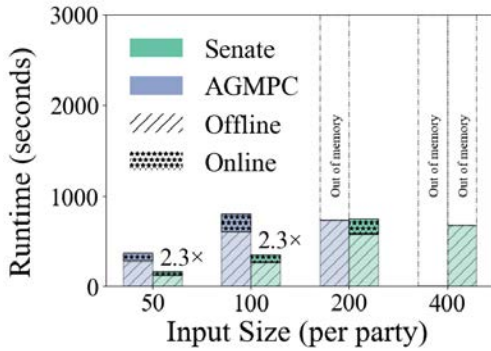
Figure 4.10: Query 2 with 16 parties.   Figure 4.11: Query 3 with 16 parties.
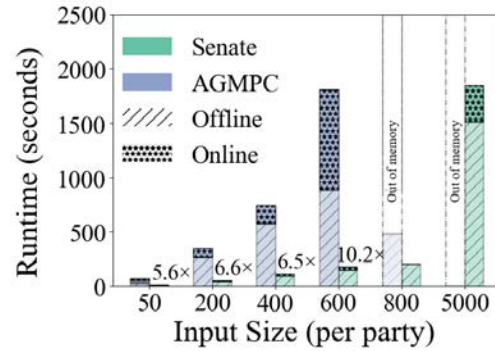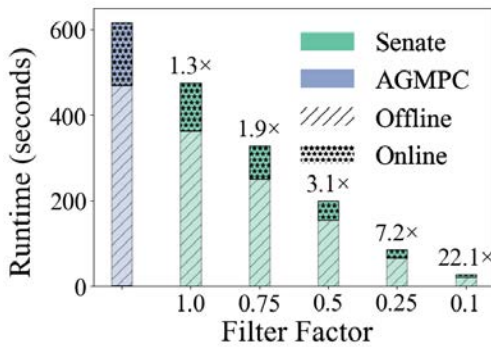
(a) Query 1 with 100 inputs/party.   (b) Query 3 with 600 inputs/party.
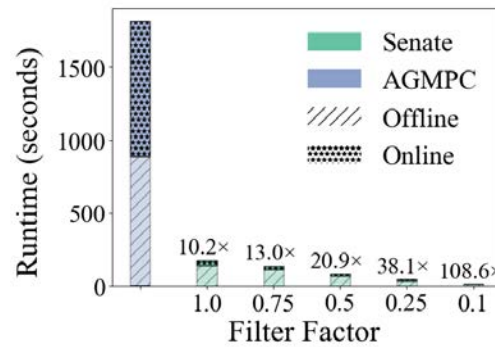
Figure 4.12: Effect of query splitting on runtime.

Figure 4.7a). We observe a similar trend in Figures 4.6a and 4.8.

4.8.2 End-to-end performance

4.8.2.1 Representative queries

We now evaluate the performance of Senate on the three representative queries discussed in Section 4.2 with a varying number of parties (Figures 4.9 to 4.11). In addition, we quantify the benefit of Senate's query splitting for different filter factors, *i.e.*, the fraction of inputs filtered as a result of any local computation (Figure 4.12). We also measure the total network usage of the queries in Figure 4.13; and Figure 4.14 plots the performance of the queries in a WAN setting.

Query 1 (Medical study).   Figure 4.9 plots the runtime of Senate and AGMPC on the medical example query with varying input sizes. Note that, the input to the circuit for a query consists of all the values in the row required to compute the final result. We observe a performance improvement of $1.3\times$ for an input size of 100 rows, and are also able to scale to higher input sizes. Figure 4.12a illustrates the benefit of Senate's consistent and verified query splitting for different filter factors. We compare the single circuit implementation of the query for 100 inputs per party,
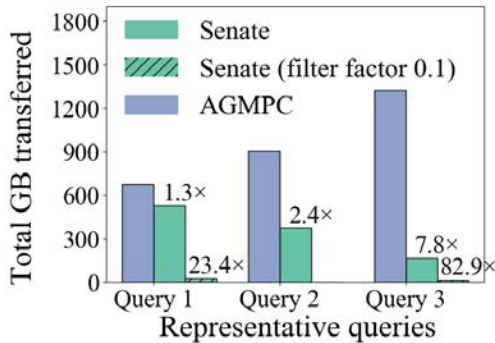
Figure 4.13: Network usage.
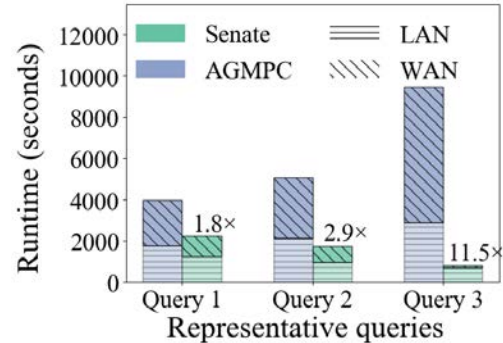
Figure 4.14: Queries in WAN.

and are able to achieve a runtime improvement of $22\times$ for a filter factor of 0.1. The improvement in network consumption follows a similar trend, reducing usage by $\sim23\times$ with a filter factor of 0.1 (Figure 4.13).

Query 2 (Prevent password reuse). Figure 4.10 plots the runtime of Senate and AGMPC with varying input sizes. Each row in this query consists of a 32 bit user identifier, and a 256 bit password hash. Since the query involves a group-by with aggregates, which is mapped to an extended m-SU (per Section 4.5), we observe a trend similar to Figure 4.6b. We remark that this query does not benefit from Senate's query splitting.

Query 3 (Credit scoring). We evaluate the third query with 16 parties and varying input sizes in Figure 4.11, and observe that Senate is $10\times$ faster than AGMPC for 600 input rows, and is able to scale to almost 10 times the input size. The introduction of a local filter into the query, with a filter factor of 0.1 reduces the runtime by $100\times$. We attribute this to our efficient m-SI implementation which optimally splits the set intersection and parallelizes its execution across parties. The reduction in network usage (Figure 4.13) is also similar.

In the WAN setting, the improvement in query performance with Senate largely mimics the LAN setting; Figure 4.14 plots the results in the absence of query splitting (*i.e.*, filter factor of 1). Overall, we find that Senate MPC decomposition protocol alone improves performance by up to an order of magnitude over the baseline. In addition, Senate's query splitting technique can further improve performance by another order of magnitude, depending on the filter factor.

4.8.2.2 TPC-H benchmark

To stress test Senate on more complex query structures, we repeat the performance experiment by evaluating Senate on the TPC-H benchmark [TPCb], an industry-standard analytics benchmark. The benchmark comprises a rich set of 22 queries on data split across 8 tables. The query structures are complex: for example, query 5 involves 5 joins across 6 tables, several filters, cross-column multiplications, aggregates over groups, and a sort. Existing benchmarks for analytical queries (including TPC-H) have no notion of collaborations of parties, so we created a multi-party version of TPC-H by assuming that each table is held by a different party.
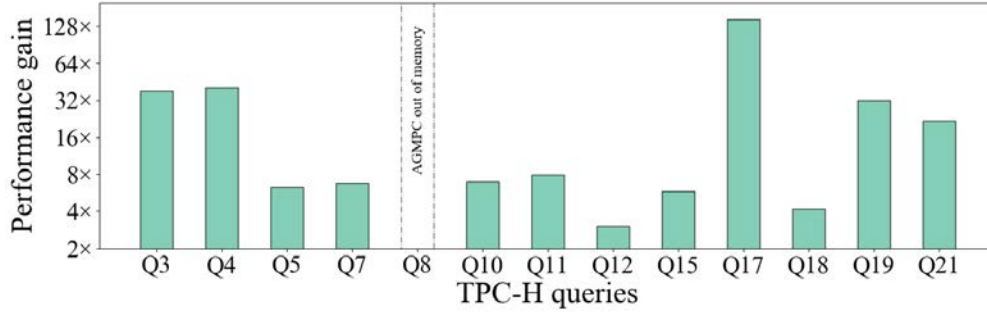
Figure 4.15: Senate's performance on TPC-H queries.
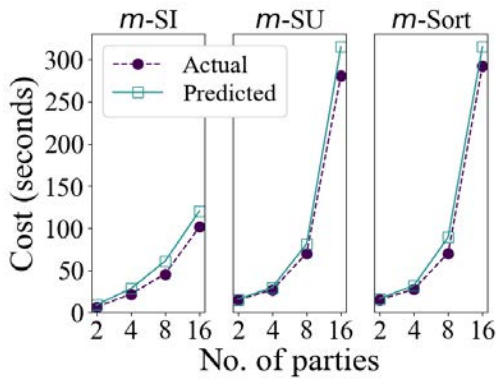
Figure 4.16: Accuracy of cost model.
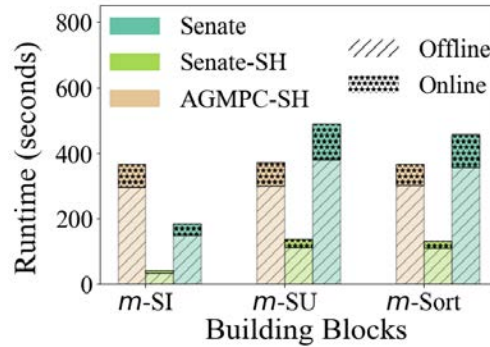
Figure 4.17: Semi-honest baselines

We measure Senate's performance on 13 out of these 22 queries; the other queries are either single-table queries, or perform operations that Senate currently does not support (namely, substring matching, regular expressions, and UDFs). For parity, we assume $1K$ inputs per party across all queries, and a filter factor of 0.1 for local computation that results from Senate's query splitting. Figure 4.15 plots the results. Overall, Senate improves performance by $3\times$ to $145\times$ over the AGMPC baseline across 12 of the 13 queries; query 8 runs out of memory in the baseline.

4.8.3   Accuracy of Senate's cost model

We evaluate our cost model (from Section 4.7.2) using Senate's circuit primitives. We compute the costs predicted by the cost model for the primitives, and compare them with the measured cost of an actual execution. As detailed in Section 4.7.2, the cost model does not consider the function independent computation in the offline phase of the MPC protocol as it does not lie in the critical path of query evaluation; we therefore ignore the function independent components from the measured cost. Figure 4.16 shows that our theoretical cost model approximates the actual costs well, with an average error of $\sim$20%.

4.8.4 Senate versus other protocols

Custom PSI protocols.   There is a rich literature on custom protocols for PSI operations. While custom protocols are faster than general-purpose systems like Senate, their functionality naturally remains limited. We quantify the tradeoff between generality and performance by comparing Senate's PSI cost to that of custom PSI protocols. We compare Senate with the protocol of Zhang *et al.* [ZLL$^+$19], a state-of-the-art protocol for multiparty PSI with malicious security.[3] The protocol implementation is not available, so we compare it with Senate based on the performance numbers reported by the authors, and replicate Senate's experiments on similar capacity servers. Overall, we find that a 4-party PSI of 2^{12} elements per party takes $\sim$3 s using the custom protocol in the online phase, versus $\sim$30 s in Senate, representing a 10$\times$ overhead.

Arithmetic MPC.   Senate builds upon a Boolean MPC framework instead of arithmetic MPC. We validate our design choice by comparing the performance of Senate with that of SCALE-MAMBA [Sca], a state-of-the-art arithmetic MPC framework. We find that though arithmetic MPC is 3$\times$ faster than Senate for aggregation operations *alone* (as expected), this benefit doesn't generalize. In Senate's target workloads, aggregations are typically performed on top of operations such as joins and group by, as exemplified by our representative queries and the TPC-H query mix. For these queries (which also represent the general case), Senate is over two orders of magnitude faster. More specifically, we measure the latency of (i) a join with sum operation, and (ii) a group by with sum operation, across 4 parties with 256 inputs per party; we find that Senate is faster by 550$\times$ and 350$\times$ for the two operations, respectively. The reason for this disparity is that joins and group by operations rely almost entirely on logical operations such as comparisons, for which Boolean MPC is much more suitable than arithmetic MPC.

Semi-honest systems.   We quantify the overhead of malicious security by comparing the performance of Senate with semi-honest baselines. To the best of our knowledge, we do not know of any modern *m*-party semi-honest garbled circuit frameworks faster than AGMPC (even though it's maliciously secure). Therefore, we implement and evaluate a semi-honest version of AGMPC ourselves, and compare Senate against it in Figure 4.17. AGMPC-SH refers to the semi-honest baseline with monolithic circuit execution. We additionally note that Senate's techniques for decomposing circuits translate naturally to the semi-honest setting, without the need for verifying intermediate outputs. Hence, we also implement a semi-honest version of Senate atop AGMPC-SH that decomposes queries across parties. We do not compare Senate to prior semi-honest multi-party systems SMCQL and Conclave, as their current implementations only support 2 to 3 parties.

Figure 4.17 plots the runtime of *m*-SI, *m*-SU and *m*-Sort across 16 parties, with 1K, 600 and 600 inputs per party respectively. We observe that Senate-SH yields performance benefits ranging from 2.7–8.7$\times$ when compared to AGMPC-SH. Senate's malicious security, however, comes with an overhead of 4.4$\times$ compared to Senate-SH. We also measure the end-to-end performance of the

---

[3]We note that the protocol of Zhang *et al.* provides malicious security only against adversaries that do not simultaneously corrupt two parties, while Senate is secure against arbitrary corruptions. However, the only custom protocols we're aware of that tolerate arbitrary corruptions (for more than two parties) either rely on expensive public-key cryptography (and are slower than general-purpose MPC, which have improved tremendously since these proposals) [DSMRY11, CJS12], or do not provide an implementation [HV17].

three sample queries, and find that Senate-SH yields performance benefits similar to Figures 4.9 to 4.11 when compared to AGMPC-SH. At the same time, we observe a maximum overhead of $3.6\times$ when running the queries in a maliciously-secure setting.

4.9 Limitations and Discussion

Applicability of Senate's techniques. Senate works best for operations that can be naturally decomposed into a tree. While many SQL queries fit this structure, not all of them do. A general case is one where the same relation is fed as input to two different operations (or nodes in the query tree). For example, consider a collaboration of 3 parties, where each party P_i holds a relation R_i, who wish to compute the join $(R_1 \cup R_2) \bowtie R_3$. In the unencrypted setting, we can decompose the operation by computing pairwise joins $R_1 \bowtie R_3$ and $R_2 \bowtie R_3$, and then take the union of the results. Unfortunately, this decomposition doesn't work in Senate because it produces a DAG (a node with two parents) and not a tree. Hence, a malicious P_3 may use different values for R_3 across the pairwise joins, leading to an input consistency issue. In such cases, Senate falls back to monolithic MPC for the operation.

Overall, Senate's techniques do not universally benefit all classes of computations, yet they encompass important and common analytics queries, as our sample queries exemplify.

Verifiability of SQL operators. As described in Section 4.7, for simplicity, Senate's compiler requires that each node in the query tree outputs values that adhere to a well-defined set of constraints. If a node constrains its outputs in any other way, the compiler marks it as unverifiable. The reason is that additional constraints restrict the space of possible inputs for future nodes in the tree (and thereby, their outputs), making it harder to deduce what needs to be verified.

For example, consider a group by operation over column a, with a sum over column b per group. If the values in b also have a range constraint, then deducing the possible values for the sums per group is non-trivial (though technically possible). Generalizing Senate's compiler to accept a richer (or possibly, arbitrary) set of constraints is interesting future work.

Additional SQL functionality. Senate does not support SQL operations such as UDFs, substring matching, or regular expressions, as we discuss in our analysis of the TPC-H benchmark Section 4.8.2.2. Adding support for missing operations requires augmenting Senate's compiler to (i) translate the operation into a Boolean circuit; and (ii) verify the invertibility of the operation as required by the MPC decomposition protocol. While this is potentially straightforward for operations such as substring matching and (some limited types of) regular expressions, verifying the invertibility of arbitrary UDFs is computationally a hard problem. Overall, extending Senate to support wider SQL functionality (including a well-defined class of UDFs) is an interesting direction for future work.

Differential privacy. Senate reveals the query results to all the parties, which may leak information about the underlying data samples. This leakage can potentially be mitigated by extending Senate to support techniques such as differential privacy (DP) [DR14] (which prevents leakage by adding noise to the query results), similar to prior work [NH12, BHE$^+$18].

In principle, one can use a general-purpose MPC protocol to implement a given DP mechanism for computing noised queries in the standard model [DKM$^+$06, EKM$^+$14]—each party contributes a share of the randomness, which is combined within MPC to generate noise and perturb the query results, depending on the mechanism. However, an open question is how the MPC decomposition protocol of Senate interacts with a given DP mechanism. The mechanism governs where and how the noise is added to the computation, *e.g.*, Chorus [JNHS18] rewrites SQL queries to transform them into intrinsically private versions. On the other hand, Senate decomposes the computation across parties, which suggests that existing mechanisms may not be directly transferable to Senate in the presence of malicious adversaries while maintaining DP guarantees. As a result, designing DP mechanisms that are compatible with Senate is a potentially interesting direction for future work.

4.10   Related work

Secure multi-party computation (MPC) [Yao82, GMW87, BGW88]**.**   A variety of MPC protocols have been proposed for malicious adversaries and dishonest majority, with SPDZ [KPR18, KOS16, DKL$^+$13] and WRK [WRK17] being the state-of-the-art for arithmetic and Boolean (and for multi/constant rounds) settings, respectively. WRK is more suited to our setting than SPDZ because relational queries map to Boolean circuits more efficiently. These protocols execute a given computation as a monolithic circuit. In contrast, Senate decomposes a circuit into a tree, and executes each sub-circuit only with a subset of parties.

MPC frameworks.   There are several frameworks for compiling and executing programs using MPC, in malicious [EMP, Sca, MGC$^+$16] as well as semi-honest [LWN$^+$15, BDNP08, MNPS04, ZSB13, BLW08, RHH14, NWI$^+$15] settings. Senate builds upon the AGMPC framework [EMP] that implements the maliciously secure WRK protocol.

Private set operations.   A rich body of work exists on custom protocols for set operations (*e.g.*, [KS05, FNO19, KRTW19, KMP$^+$17, PSTY19, CGT12, CKT10]). Senate's circuit primitives build upon protocols that express the set operation as a Boolean circuit [HEK12, BA12] in order to allow further MPC computation over the results, rather than using other primitives like oblivious transfer, oblivious PRFs, etc.

Secure collaborative systems.   Similar to Senate, recent systems such as SMCQL [BEE$^+$17] and Conclave [VSG$^+$19] also target privacy for collaborative query execution using MPC. Other proposals [ABG$^+$05, CLS09] support such computation by outsourcing it to two non-colluding servers. However, all these systems assume the adversaries are semi-honest and optimize for this use case, while Senate provides security against malicious adversaries. Prio [CGB17], Melis *et al.* [MDC16], and Prochlo [BEM$^+$17] collect aggregate statistics across many users, as opposed to general-purpose SQL. Further, the first two target semi-honest security, while Prochlo uses hardware enclaves [MAB$^+$13].

Similar objectives have been explored for machine learning (*e.g.*, [Gooc, BIK$^+$17, ZPGS19, GSB$^+$17, NWI$^+$13, MZ19, SS15]). Most of these proposals target semi-honest adversaries. Others are limited to specific tasks such as linear regression, and are not applicable to Senate.

Trusted hardware. An alternate to cryptography is to use systems based on trusted hardware enclaves (*e.g.*, [ZDB$^+$17, EZ20, PVC18]). Such approaches can be generalized to multi-party scenarios as well. However, enclaves require additional trust assumptions, and suffer from many side-channel attacks [BMW$^+$18, WCP$^+$17].

Systems with differential privacy. DJoin [NH12] and DStress [PNH17] use black-box MPC protocols to compute operations over multi-party databases, and use differential privacy [DR14] to mask the results. Shrinkwrap [BHE$^+$18] improves the efficiency of SMCQL by using differential privacy to hide the sizes of intermediate results (instead of padding them to an upper bound, as in Senate). Flex [JNS18] enforces differential privacy on the results of SQL queries, though not in the collaborative case. In general, differential privacy solutions are complementary to Senate and can possibly be added atop Senate's processing by encoding them into Senate's circuits (as discussed in Section 4.9).

4.11 Summary

We presented Senate, a system for securely computing SQL queries in federated databases. Unlike prior work, Senate targets a powerful adversary who may arbitrarily deviate from the specified protocol. Compared to traditional cryptographic solutions, Senate improves performance by being able to decompose a big cryptographic computation into smaller and parallel computations, planning an efficient decomposition, and by verifiably delegating a part of the query to local computation. Our techniques can improve query runtime by up to $145\times$ when compared to the state-of-the-art.

Chapter 5

Analyzing Encrypted Network Traffic

To meet the demands of workloads with strict performance requirements, we turn to systems that avoid the cost of cryptographic protocols with the help of trusted execution environments (or enclaves). This chapter illustrates the design principles behind such systems, and presents SafeBricks, a system that shields the analysis of network traffic from an untrusted cloud provider.

5.1   Introduction

Modern networks consist of a wide range of appliances that implement advanced network functions beyond merely forwarding packets, such as scanning for security issues (*e.g.*, firewalls, IDSes) or improving performance (*e.g.*, WAN optimizers, web caches). Traditionally, these network functions (or NFs) have been deployed as dedicated hardware devices. In recent years, however, both industry and academia have proposed the replacement of the devices with software implementations running in virtual machines [SHS$^+$12, Eur], a model called Network Function Virtualization (NFV). Inevitably, the advent of NFV has spurred the growth of a new industry wherein third-parties offer traffic processing capabilities as a cloud service to customers [SHS$^+$12, Ary, Zsc, Pala]. Such a service model enables enterprises to outsource NFs from their networks entirely to the third-party service, bringing the benefits of cloud computing and reducing costs.

However, outsourcing NFs to the cloud poses new challenges to enterprise networks—security.

Need to protect traffic from the cloud.   By allowing the cloud provider to process enterprise traffic, enterprises end up granting to the cloud the ability to see their sensitive traffic and tamper with NF processing. While the cloud itself might be a benign entity, it is vulnerable to hackers [Pri], subpoenas [Goob, Mica, Yah], and insider attacks [Bee10, Pou08, Zet10]. This is doubly worrisome because not only does network traffic contain sensitive information, but some NFs are also designed to protect enterprises against intrusions which an attacker could try to disrupt.

Need to protect traffic from NF.   What complicates matters further is that often, an enterprise must also trust another party with its traffic: NF vendors. This is the case when enterprises procure proprietary NF implementations and rulesets from NF vendors [Pala, Bar, For] instead of using their own, as shown in Figure 5.1. While such NFs typically need access only to specific portions of the
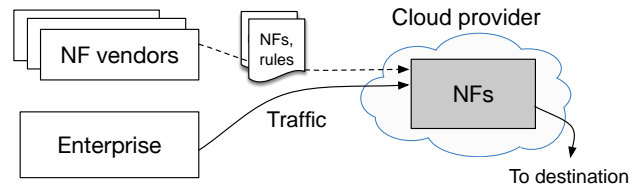
Figure 5.1: Model for outsourced NFs.

traffic (*e.g.*, IP firewalls only need read access to packet headers), the enterprise by default entrusts the NFs with *both read/write access* over *entire* packets.

Need to protect NF source code.   This model threatens the security of the NF vendors as well, who have a business interest in maintaining the privacy of their code and rulesets (often baked into the source code) from both the cloud and the enterprise. NFs have traditionally been shipped as hardware devices, so being shipped as software now exposes them further to untrusted platforms (*e.g.*, it is possible to reverse binaries).

The question is: how can we design an NF processing framework that meets all these security goals?

There has been little prior work in this space, consisting of mostly two approaches. Cryptographic approaches such as BlindBox [SLPR15] and Embark [LSP$^+$16] are significantly limited in functionality, supporting only simple functions such as = and >. They are unable to support more sophisticated operations such as regular expressions (needed in common NFs such as intrusion detection systems) or process custom NF code. Least-privilege approaches such as mcTLS [NSV$^+$15] aim to give each NF access to only part of the packet and are designed for *hardware middleboxes*; however, when used in the cloud setting, they provide weak guarantees because the cloud receives the *union* of the permissions of all middleboxes, which often, is everything. Neither of these approaches protects the NF source code, and both require significant changes to TLS, which is an impediment to adoption.

We present SafeBricks, a system for outsourcing NFs that provides protection with respect to the three security needs above. SafeBricks addresses the discussed limitations of prior work by supporting *generic* NF functionality with significantly stronger security guarantees, without requiring changes to TLS. It builds upon NetBricks [PHJ$^+$16], a framework for building and executing arbitrary NFs that uses a safe language and runtime, Rust.

To overcome the limited functionality of cryptographic approaches, SafeBricks *shields* [BPH14] traffic processing from the cloud by executing the NFs within *hardware enclaves* such as Intel SGX [MAB$^+$13] (as described in Chapter 2). This approach promises that neither an administrator with root privileges nor a compromised operating system can observe enclave-protected data in unencrypted form, or tamper with the enclave's execution. Enclaves have already been used to shield *general-purpose* computation from the cloud provider [BPH14, SCF$^+$15, HZX$^+$16, ATG$^+$16]. Applying them to network processing is a natural next step, as recent proposals have pointed out (see Section 5.11).

While this idea is simple, designing a system that provides protection with respect to the three

security goals above, and simultaneously maintains good performance, is far more challenging. We now discuss a set of challenges overcome by SafeBricks, and give insights into the techniques for addressing them.

First, general-purpose approaches result in a large trusted computing base (TCB) inside the enclaves (up to millions of LoC), any vulnerability in which can result in information leakage. In SafeBricks, we investigate *how to partition* the code stack of NF applications (from packet capture to processing) and choose a boundary that reduces the code within the trusted domain without compromising security.

Second, partitioning an application is likely to result in *transitions* between enclave and non-enclave code. These transitions are expensive, introducing a high run-time overhead due to the cost of saving/restoring the state of the secure environment. Consequently, there is a tension between TCB size and the overall performance of the application: the lesser code the enclave contains, the more transitions it is likely to make to non-enclave code. In SafeBricks, we address these challenges simultaneously by developing an architecture that leverages shared memory and splits computation across enclave and non-enclave threads (while verifying the work of the non-enclave threads) without performing transitions.

Third, NFV deployments typically comprise multiple NFs running in a chain, isolated via VMs or containers for safety. In our setting, the straightforward way of achieving this isolation would be to deploy each NF in a separate enclave. However, as we discuss in Section 5.6, such an architecture can result in a system that is ∼2–16× slower than the baseline. Instead, SafeBricks supports chains of NFs within the *same* enclave. To isolate them, SafeBricks leverages the semantics of the Rust language.

Nevertheless, this strategy introduces a new difficulty: all NFs must be assembled using a trusted compiler. Though the client enterprise is the natural site for building the NFs safely, doing so would leak the source code of the NFs to the client, which is undesirable for the NF vendors. To address this challenge, SafeBricks runs in an enclave at the cloud a *meta*-functionality: a compiler that creates an encrypted binary, and a loader that runs this binary in a separate enclave. Using the remote attestation feature of hardware enclaves, both the NF vendors and the client can verify that the agreed-upon compiler and loader are running in an enclave, before the vendors share the NF code and the client shares data and traffic.

Finally, none of the above satisfies our requirement for enforcing least privilege across NFs: each NF still has access to entire packets. SafeBricks enforces least privilege by (i) exposing an API to the client for specifying the privileges of each NF, and (ii) ensuring that the SafeBricks framework *mediates* all NF accesses to packets, both reads and writes. To enforce the latter, SafeBricks leverages the safety guarantees of Rust.

We note that SafeBricks's least privilege enforcement does not rely on hardware enclaves; it is a contribution of independent interest, for settings where the client enterprise trusts the cloud provider, but wants to reduce its trust in the NF implementations.

We evaluate SafeBricks across four different NF applications using both synthetic and real traffic. Our evaluation shows that the performance impact of SafeBricks is reasonable, ranging between ∼0–15% across NFs.

5.2 Model and Threat Model

As shown in Figure 5.1, there are four types of parties in our setting: (1) a *cloud provider* that hosts the outsourced NFs; (2) a *client enterprise* outsourcing its traffic processing to the cloud; (3) *two endpoints* that communicate over the network, at least one of which is within the enterprise; and (4) *NF vendors* that supply the code and rulesets for network functions.

The client enterprise contains a gateway (as shown in Figure 5.2) which is trusted. The endpoints are trusted only with their communication.

The core of SafeBricks's design builds on the abstract notion of a hardware enclave. Our implementation uses Intel SGX [MAB$^+$13], a popular hardware enclave, but few design decisions are tailored to SGX. We provide some relevant background on hardware enclaves, and then define the threat models for the cloud and the NF vendors.

5.2.1 Threat model for the cloud and enclaves

SafeBricks leverages hardware enclaves running on the cloud provider. Our threat model for the cloud provider is similar to prior works [BPH14, ATG$^+$16, SCF$^+$15] that build on hardware enclaves. Enclaves strive to provide an abstract security guarantee so that systems like SafeBricks can build on them in a black-box manner; however, current implementations do not yet fully achieve this guarantee because they are vulnerable to side-channel attacks (as described in Chapter 2 and as we discuss below).

Abstract enclave assumption. The attacker cannot observe any information about the protected code and data in the enclave, and the remote attestation procedure establishes a secure connection between the correct parties and loads the desired code into the enclave.

Attacker capabilities. Except the out-of-scope attacks described below, we consider an attacker that can compromise the software stack of the cloud provider outside the enclave, which includes privileged software such as the hypervisor and kernel. In particular, whenever the enclave exits or invokes code outside the enclave, the attacker can instead run arbitrary code and/or respond with arbitrary data to the enclave. For example, the OS can mount an Iago attack [CS13] and respond incorrectly to system calls. Note that this threat model implies that the attacker can observe communication between hardware enclaves as well as communication on the network.

Out-of-scope attacks. In short, all attacks that violate the abstract enclave assumption above are out of scope for SafeBricks. For example, we consider as out of scope all hardware and side-channel attacks, as well as assume that the enclave manufacturer (*e.g.*, Intel) is trusted. Intel SGX's current implementation does not fully achieve the enclave assumption above because it suffers from side-channel attacks as detailed in Chapter 2. While these are important issues with SGX, we treat them as out of scope for SafeBricks because solutions to these are orthogonal and complementary to our contribution here. Recently, a number of solutions have been proposed for solving or mitigating many of these attacks [CLD16, SCNS16, SLKP17, GLS$^+$17a, CZRZ17].
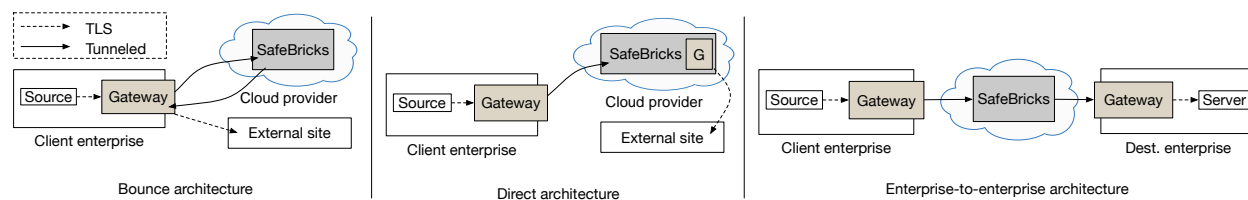
Figure 5.2: End-to-end system architecture

5.2.2 Threat model for network functions

Each NF is trusted only with the permissions given to it by the enterprise for specific packet fields. That is, if the enterprise gives a NAT read/write permissions for the IP header, the NF is trusted to not leak the header to unauthorized entities and to modify it correctly. At the same time, if the NAT attempts to access the packet payload, then SafeBricks must prevent it from doing so.

5.3 SafeBricks: End-to-end Architecture

APLOMB [SHS$^+$12] discusses in detail the architecture for outsourcing NF processing to the cloud by redirecting client traffic, as well as the merits of this architecture. Here, we focus on how SafeBricks enhances this architecture with protection against cloud attackers and TLS compatibility, while maintaining performance.

SafeBricks supports three typical architectures considered in the cloud outsourcing model [SHS$^+$12, LSP$^+$16], as shown in Figure 5.2. These architectures have different merits or constraints, and are useful for different cases.

Let *S* be the source endpoint, *G* the client gateway, *CP* the cloud provider running NFs using SafeBricks (*SB*), and *D* the destination endpoint. Let G_1 be the gateway near the source, and G_2 be the gateway near the destination. Note that in the Direct architecture, an enclave in the cloud plays the role of G_2, and in the Bounce architecture, a single gateway plays both G_1 and G_2. *CP* runs hardware enclaves; code and data are decrypted inside enclaves, but remain encrypted outside. *D* could either be an external site or an endpoint in another enterprise.

1. **Bounce:** In the bounce architecture, *SB* tunnels processed traffic to *G* over the secure channel. *G* then forwards the processed traffic to the destination. The response from *D* is similarly redirected by *G* to *SB* before forwarding it to *S*. The bounce setup is the simplest in that it does not place any added burden on *SB* or *D* from a functionality and security perspective. However, it inflates the latency between *S* and *D* as a result of bouncing the processed traffic to *G*.

2. **Direct:** The direct architecture alleviates the latency added by the bounce setup. *SB* directly forwards the enterprise traffic to *D* after processing it without bouncing it off the gateway. However, this setup comes at the cost of security: since there is no secure channel between *SB*

and *D* over which traffic can be tunneled, *SB* must necessarily send the processed packets to *D* in the clear, revealing the headers to *CP*. If *S* and *D* use TLS, *CP* will not see the payload.

3. **Enterprise-to-enterprise:** If *S* and *D* belong to the same enterprise or to enterprises that trust each other, it is possible to have the combined benefits of the bounce and direct architecture. *SB* tunnels the processed traffic to G_2, so *CP* does not see any headers at any time. At the same time, this approach does not suffer from the bounce setup's latency.

Though not the focus of this work, it is worth mentioning that SafeBricks can also be used in a local cloud deployment in which the NFs run within the client enterprise. This benefits the client by providing SafeBricks's isolation and least privilege for NFs, as well as protection against administrators of the local cloud (although the gateway administrators need to remain trusted).

Compared to the architecture of cryptographic solutions like Embark [LSP$^+$16], Embark does not support the direct architecture because the special encryption needs to be stripped off before reaching the final destination. It also requires the gateway to do much more work than for SafeBricks—*i.e.*, to compute a part of the NF functions as part of the encryption scheme.

5.3.1 Overview of the communication protocol

Our protocol for handling connections is the same for all architectures, as we now explain in terms of G_1 and G_2.

System bootstrap. The client enterprise first sets up and verifies the enclaves in the cloud as explained in Section 5.7. As part of this process, the gateways are able to set up a set of IPSec tunnels with the cloud in a secure way (such as installing certificates to avoid the risk of a man-in-the-middle attack). To load-balance flows at the cloud server via receive-side scaling (RSS), the number of IPSec tunnels depends on the number of ports at the server.

As with all such interception systems, the source endpoints need to be configured to allow interception. There are at least two approaches and SafeBricks supports both of them. The most common is the *interception proxy* [Jar12], in which the sources' browsers accept certificates from the proxy which can now terminate the TLS connection. Another approach [HKHH17] is to insert a browser plugin at the clients in the enterprise, which sends session keys to the gateway over a secure channel. In both cases, note that the interception proxy and the receiver of these session keys will be the gateways, and not the cloud provider. This is unlike the current cloud deployment where the cloud intercepts and can see all traffic. Of course, SafeBricks supports non-TLS encrypted traffic too, which is a simplified version of the protocol for TLS-encrypted traffic.

Upon a new TLS connection from a source.

1. G_1 terminates the connection using the interception mechanism of choice above and informs G_2, and

2. G_2 starts the TLS connection to the destination.

Packet processing.

1. *S* sends packets over TLS to *D*.

2. G_1 intercepts the traffic, decrypts it from TLS, and tunnels it over an IPSec connection. G_1 sends all packets that belong to the same flow on the same IPSec connection. Multiple flows might be mapped to the same IPSec connection. As part of this process, the entire packet is encrypted and encapsulated in a new header. The payload and the actual header are thus encrypted. The encryption algorithm used in IPSec is AES in GCM mode, which includes packet authentication.

3. The enclave at *CP* receives each packet, decapsulates it by stripping off the extra header, checks its integrity tag, decrypts it and executes the NFs (as discussed in Section 5.5). It then tunnels the packets over IPSec to G_2.

4. G_2 terminates the IPSec tunnel, and forwards the traffic over TLS to the destination server.

5.4 Background

Before delving into the design of SafeBricks, we provide a brief overview of NetBricks and some additional details on Intel SGX relevant to our system.

5.4.1 Intel SGX

Illegal enclave instructions.   SGX does not allow instructions within an enclave that result in a change of privilege levels (*e.g.*, system calls) or cause a VMEXIT. Applications that need to perform such instructions must exit the enclave and transfer control to host software.

Memory architecture.   Enclave pages reside in a protected memory region called the enclave page cache (EPC), whose size is limited to ∼94MB in current hardware. EPC pages are decrypted when loaded into cache lines, and integrity-protected when swapped to DRAM.

5.4.2 NetBricks

The NetBricks framework [PHJ⁺16] enables the development of arbitrary NFs by exposing a small set of customizable programming abstractions (or operators) to developers. In this respect, NetBricks is similar to Click [KMC⁺00], which also enables developers to write NFs by composing various packet processing elements. However, we choose to build our system atop NetBricks instead of Click for the following reasons:

- Unlike Click, the behavior of NetBricks' operators can be heavily customized via user-defined functions (UDFs). This allows us to protect a small number of operators within the enclave (with NetBricks), which are then composed into NFs, as opposed to routinely adding new Click modules.

- More importantly, NetBricks builds upon a safe language and runtime, Rust, to provide isolation between NFs chained together in the same process. In Section 5.6, we describe how SafeBricks extends these guarantees to provide least privilege across NFs inexpensively.

- NetBricks' zero-copy semantics also improve performance substantially [PHJ^{+}16].

We now briefly describe some features of NetBricks relevant to the design of our system.

Programming abstractions. To construct an NF, the developer specifies a directed graph consisting of NetBricks' operators as nodes. For example, the `parse` operator casts packet buffers into protocol structures; `transform` modifies packet buffers; and `filter` drops packets based on a UDF. All nodes in the NF graph process packets in batches.

Execution environment. The NetBricks scheduler implements policies to decide the order in which different nodes process their packets. Chains of NFs are run in a single process by composing their directed graphs together as function calls, instead of running each NF separately in a container or VM. For isolation between NFs, NetBricks relies on a safe language and runtime, Rust.

Packet I/O. NetBricks builds on top of DPDK [Inta], a fast packet I/O library. DPDK polls packets from the network devices, buffers them in pools of memory, and maintains a queue of pointers to the packet buffers. NF instances query DPDK via an I/O interface to retrieve pointers to the next batch of packet buffers, and process them in-place without performing any copies.

5.5 SafeBricks: Framework Design

We now describe how we build our system on top of NetBricks (while redesigning some parts of it). We carefully partition the components of NetBricks into modules that run within the enclave in the trusted environment, and modules that run outside the enclave in the untrusted environment.The former constitutes our TCB decoupled from the original framework and protected within an enclave, while the latter remains resident in memory outside the enclave and is subject to attacks. Figure 5.3 shows the overall design of the framework, highlighting the components modified or introduced by SafeBricks. Our goal in this section is to reduce the size of the TCB while minimizing the overhead of transitions between the enclave and the host. However, these two goals are often at odds with each other—the lesser code the enclave contains, the more transitions it makes to outside code. We now describe how our design balances both these aims.

5.5.1 Partitioning NetBricks

We carefully split NetBricks into two components—enclave code and host code.

SafeBricks enclave. At a bare minimum, the enclave should include the programming and state abstractions of NetBricks. However, during execution, the NetBricks scheduler takes decisions regarding which node to process next in the directed graph representing the NF (as described in Section 5.4.2). These decisions are frequent—every time a node is done processing a batch of packets, it surrenders control to the scheduler. As a result, excluding the scheduler from the TCB
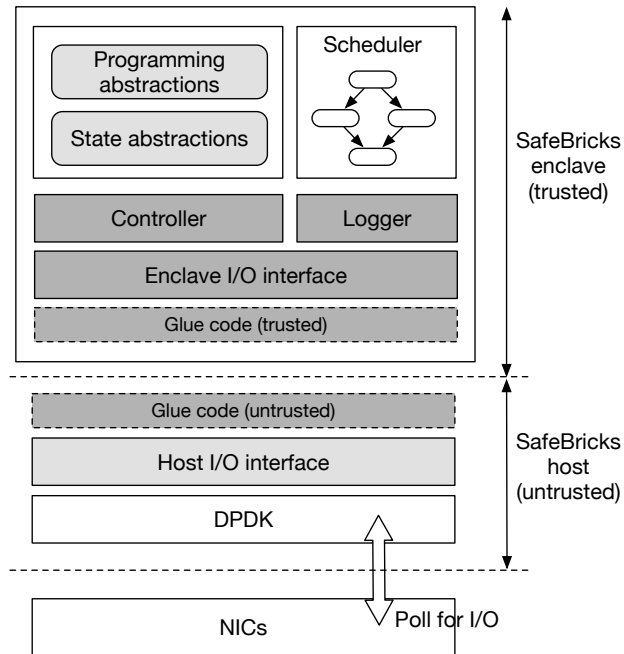
Figure 5.3: SafeBricks framework: White boxes denote existing NetBricks components, light grey boxes denote modified components, and dark grey boxes denote new components.

would result in a large number of enclave transitions per packet batch. Hence, we include the scheduler in our TCB as well.

SafeBricks host. The remaining components of NetBricks (mostly pertaining to packet I/O) together form the SafeBricks host. As described in Section 5.4.2, NFs in NetBricks directly access the packet buffers allocated by the packet capture library (DPDK) without copying them. Simply excluding DPDK from the enclave without other modifications is not a viable option because it would gain access to the packets once they are decrypted. On the other hand, including DPDK within the enclave would drastically inflate the size of the TCB by ∼516K LoC.

We circumvent this issue by introducing two new operators in NetBricks: `toEnclave` and `toHost`. The `toEnclave` operator polls the I/O interface for pointers to packet buffers, reads the encrypted buffers from DPDK-allocated memory and decrypts them inside the enclave. Once the processing is complete, the `toHost` operator re-encrypts the packet buffers and returns them outside the enclave into DPDK's memory pool.

More concretely, `toEnclave` and `toHost` implement endpoints of the IPSec tunnel. As a result, even if the host attacker attempts Iago attacks [CS13] such as modifying packet buffers or queues outside the enclave, these will be detected by the authenticity provided by IPSec.

Excluding DPDK from the TCB enables us to remove NetBricks' I/O module from the TCB as well. The module interfaces with the packet capture library and is used by the NFs to poll DPDK for packets (Figure 5.3).
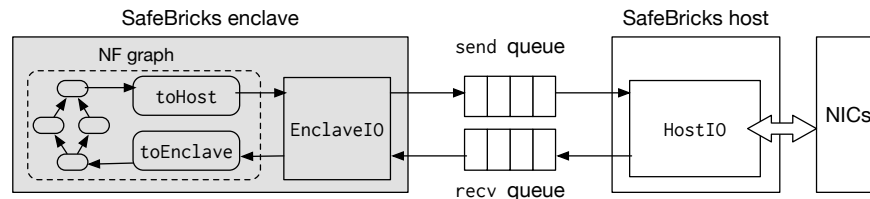
Figure 5.4: Packet I/O via shared memory.

5.5.2 Packet I/O avoiding enclave transitions

Every receive or send operation for a batch of packets results in an invocation of the I/O interface. Since we exclude the packet capture library from the TCB, every such invocation necessarily results in an enclave transition. Batch processing of packets alleviates the overhead of these transitions to some extent, but as we show in Section 5.9.2.1, it is far from being a perfect solution.

Prior works [ATG+16, OLMS17] have also explored the reduction of enclave transitions, albeit in a different context—they allow enclave threads to delegate system calls to the host with the help of shared queues. In a similar spirit, we propose an alternative design point that allows enclave code to receive and send packet batches from the host via shared memory, without the need for enclave transitions. To do so, we (i) introduce an additional trusted I/O module within the enclave (called `EnclaveIO`) that exposes the I/O APIs transparently to the rest of enclave code, and (ii) modify the NetBricks I/O interface outside the enclave (`HostIO`) to appropriately interface with the `EnclaveIO` module.

SafeBricks allocates two lockless circular queues (`recvq` and `sendq`) on heap memory outside the enclave during the application's initialization, one for receiving pointers to packet buffers and the other for sending. `HostIO` busy polls DPDK for incoming packets and populates `recvq` with the buffer addresses. Enclave code queries the `EnclaveIO` module which in turn reads the packet buffer addresses directly from `recvq` without having to exit the enclave. To send packets, `EnclaveIO` pushes the packet buffer addresses into `sendq`. `HostIO` consumes the buffers asynchronously from this queue, and finally invokes the I/O interface to emit the packets to the network. Figure 5.4 illustrates the approach.

This mechanism doesn't result in any enclave transitions because (i) enclave code can readily access memory outside the enclave, and (ii) the queue management is asynchronous—the `HostIO` module and the SafeBricks enclave (containing `EnclaveIO`) run in separate threads.

5.5.3 System calls and other illegal instructions

As described in Section 5.4.1, SGX allows neither system calls within enclaves nor instructions that could lead to a `VMEXIT` (such as `rdtsc`, used for reading the timestamp counter). There exist a set of general-purpose systems [BPH14, HZX+16, ATG+16, STTS17, OLMS17, TPV17] that add support for such system calls to enclave applications, at the expense of added complexity and/or a significant increase in TCB.

We note that many NFs simply do not make system calls or execute instructions that require VM exits, and those made are typically only of a few types: such as I/O for maintaining logs, or timestamp measurements using `rdtsc`. For example, no application in the NetBricks or Bess source trees [Ber, Net] implements system calls. This is due to the high-performance goal of NFs, aiming to run exclusively in user-space [Inta, Riz12, JMK$^+$17, The, Palb]. The same extends to user-space implementations of networking stacks as well, which are gaining in popularity [JWJ$^+$14, LKL, EYC$^+$16, lwI, Mir]. Therefore, instead of exposing an exhaustive API within the enclave for these instructions, SafeBricks focuses only on the operations essential for NFs and executes them without the need for enclave transitions. SafeBricks does not expose any other system calls or illegal instructions that would require enclave exits to NFs within the enclave.

Logging. Instead of allowing NFs to write to files, we expose a new state abstraction in SafeBricks that enables them to directly push logs to queues allocated in heap memory outside the enclave (similar to how we perform packet I/O). During system initialization, the Logger module allocates a queue in non-enclave heap per NF that logs information. NFs can push log entries to the respective queue by invoking the Logger module. Host code asynchronously reads the logs off these queues and writes them to files.

However, since this heap memory is untrusted and visible outside the enclave, we need to take additional steps to ensure the security of the logs (as they contain sensitive packet information). We encrypt and chain together log entries via authentication tags, a fairly standard technique. Suppose an NF submits a request to the Logger module for appending line L_i to its log. The Logger module first encrypts the line after prefixing it with an `id` associated with the NF, to obtain the ciphertext $C_i = \mathsf{Enc}(\texttt{id}||L_i)$. It then computes an authentication tag T_i over this ciphertext in conjunction with the tag for the previous line: $T_i = \mathsf{Auth}(C_i||T_{i-1})$. The Logger then pushes (C_i, T_i) to the log queue. By including the previous tag in the computation for the current tag, we ensure that host code cannot arbitrarily drop log items or reorder them without getting detected.

Our approach optimizes for writing to the log, instead of verification (which happens offline). A more efficient logging mechanism is beyond the scope of this paper and other existing protocols for efficient logging and verification [MT09, Mer79] can be easily integrated with SafeBricks. The Logger module maintains the root authentication tag within the enclave. Verifiers can later validate the log by obtaining the latest tag from the enclave over a secure channel and replaying the log. We note that by itself, the approach doesn't prevent rollback attacks on the logs; however, techniques for avoiding such attacks exist and can be deployed in a complementary fashion [SP16].

Timestamps. SafeBricks relies on the `HostIO` module to capture the timestamp per incoming packet batch and write it to a slot in the packet buffer reserved for external metadata. NFs that need timestamps for their functionality simply read it off the packets. This approach also reduces latency when chains of NFs are deployed together, as the cost of measuring the timestamp is borne only once. Though it is possible to ensure the monotonicity of timestamps, SafeBricks does not guarantee that the timestamps are correct—this is unavoidable in the current SGX implementation as the reporting module is not trusted hardware.

5.5.4   Execution model

As described in Section 5.4.2, a single NetBricks process runs several NF instances in parallel, and the execution environment is responsible for dividing the instances across cores. SafeBricks mirrors this model and runs the NFs in a multi-threaded enclave, each enclave thread running a separate NF instance.

In the case an NF instance performs packet I/O via transitioning to host code, SafeBricks adds no extra overhead on system resources compared to NetBricks, and enclave threads are mapped to cores as in NetBricks. On the other hand, the shared memory mechanism for packet I/O (via the `HostIO` module) adds extra burden on the system resources, as they require an extra thread for running the `HostIO` module. This cost, however, gets amortized by mapping a single `HostIO` instance to multiple NF instances.

In Section 5.9.2 we compare in greater detail the overhead incurred by the two approaches—packet I/O via enclave transitions, and packet I/O via shared memory.

5.6   SafeBricks: NF Isolation, Least Privilege

SafeBricks gives enterprises the flexibility to source NFs from different vendors and deploy them together on the same platform, while isolating them from each other and controlling which parts of a packet each NF is able to read or write. For example, consider a chained NF configuration wherein traffic is first passed through a firewall, then a DPI, and finally a NAT. The firewall application only needs read access to packet headers; the DPI needs read access to headers and payload; while the NAT needs read and write access to packet headers. SafeBricks ensures that each NF is given only the minimum level of access to each packet as required for their functions, *e.g.*, the firewall is unable to write to packet headers, or read/write to the payload. In other words, SafeBricks isolates NFs from one another while enforcing the principle of *least privilege* amongst them.

5.6.1   Strawman scheme

The importance of least privilege access to traffic has been recognized before in mcTLS [NSV$^+$15], which relies on physical isolation of NFs and enforces least privilege by encrypting and authenticating each field of the packet separately using different keys. Each NF is given the keys only for fields that it needs access to. To allow read access, the NF is given the encryption keys; for writes, the NF is given the authentication keys as well. Packets are re-encrypted before being transferred from one NF to the other. In the mcTLS model, NFs are isolated by virtue of being deployed on separate systems (hardware or VMs). Correspondingly in our setting, it suffices to run each NF concurrently in a separate enclave isolating their address spaces, as shown on the left of Figure 5.5.

Such an approach, however, eliminates much of the performance benefits of the underlying NetBricks framework. In addition to adding significant overheads due to repeated re-encryption of packets, it requires packets to cross core boundaries between NF enclaves (for enclaves affinitized to separate cores). Together, this can result in a system that is up to ~2–16× slower (as we show in
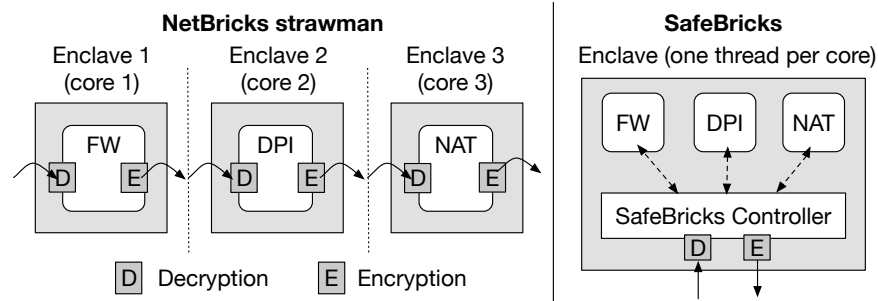
Figure 5.5: Strawman approach for enforcing least privilege versus SafeBricks. Solid arrows indicates packet transfers. Dotted arrows indicate interaction between NFs and the Controller.

Section 5.9.2.3). Instead, it would be ideal to keep all NFs in the same enclave and isolate them efficiently within.

5.6.2 NF isolation in NetBricks

Before describing how SafeBricks enforces least privilege across NFs, we revisit crucial properties of the Rust language that form the basis of our design.

The NetBricks framework provides isolation between NFs running in the same address space by building on a safe language, Rust [BBB+17, PHJ+16]. Rust's type system and runtime provide four properties crucial for memory isolation: (i) they check bounds on array accesses, (ii) prohibit pointer arithmetic, (iii) prohibit accesses to null objects, and (iv) disallow unsafe type casts.

In addition to memory isolation, NFs also require packet isolation; *i.e.*, NFs should not be able to access packets once they've been forwarded. NetBricks relies on Rust's unique types [BBB+17, GPP+12] to isolate packets. Rust enforces an *ownership model* in which only a unique reference exists for each object in memory. Variables acquire sole ownership of the objects they are bound to. When an object is transferred to a new variable, the original binding is destroyed. Rust also allows variables to temporarily *borrow* objects without destroying the original binding. By harnessing Rust's ownership model, NetBricks ensures that once an NF is done processing a packet, its ownership is transferred to the next NF and the previous NF can no longer access the packet.

Taken together, the properties of NetBricks suffice for the purpose of running NFs safely within the same address space. However, they do not provide the desired security, as we explain next.

5.6.3 Isolating NFs within the same enclave

The properties of NetBricks do not satisfy the requirements of our threat model for the following reasons:

- The isolation guarantees only hold if NFs are built using a compiler that *enforces* the safety properties above. In our model, however, enterprises may source NFs from various vendors that compiled them in their own way and lack incentive to enforce these properties.

```
pub fn chain<T:'static + Node>(input:T, pmap:HashMap) -> Node {
    let input = input.toEnclave()
        .wList(pmap.get('firewall'));
    let mut chain = firewall(input)
        .wList(pmap.get('dpi'));
    chain = dpi(chain)
        .wList(pmap.get('nat'));
    return nat(chain)
        .toHost();
}
```

Figure 5.6: Code for chaining NFs together (firewall, DPI, and NAT), generated automatically by SafeBricks from a configuration file. Lines in magenta represent code added by SafeBricks over and above NetBricks to enforce least privilege across NFs.

- Each NF still receives ownership of *entire packets*, instead of limited read / write access to specific fields.

We now describe how SafeBricks addresses both issues.

5.6.3.1 Ensuring memory safety

SafeBricks needs to ensure that NFs are built using a compiler that prohibits unsafe operations inside NFs. Instead of trusting NF providers, SafeBricks ensures that a trusted compiler gets access to the raw source code of all the NFs which it can then build in a trusted environment.

This strategy is seemingly in conflict with the confidentiality of NF rules. In Section 5.7 we show how SafeBricks performs this compilation such that neither the enterprise nor the cloud learns the source code of the NFs.

5.6.3.2 Enforcing least privilege

SafeBricks extends NetBricks' memory safety guarantees by interposing on its packet ownership model. Instead of transferring packets across NFs, SafeBricks introduces a Controller module that mediates NF access to packets as depicted in Figure 5.5 (right).

Controlling access to packets. The Controller holds ownership of packet buffers, and NFs can only *borrow* packet fields (or different fragments of the data buffers) by submitting requests to the Controller. To provide least privilege, each packet in SafeBricks encapsulates a bit vector of *permissions*. Each function in the packet API exposed by the Controller is associated with a bit in the permissions vector. Before lending the NF a reference to the requested field, the Controller checks the corresponding bit in the vector and answers the request only if the bit is set. Otherwise, the call returns an error. Furthermore, by controlling whether an API call returns a *mutable* or an *immutable* reference, the framework also disambiguates read access from writes. Rust's type system ensures that once the NF processing completes, the binding between the reference and the field is destroyed, and any later attempt by the NF to access the field will result in a compilation error.

Setting packet permissions. SafeBricks updates the permissions vector in packets with the help of a new packet processing operator: `wList` (whitelist). Chained NFs are interleaved with invocations of the `wList` operator that applies a given vector of permissions to each packet batch before it's processed by the next NF. Figure 5.6 illustrates the code for chaining NFs together while enforcing least privilege. In Section 5.7, we describe how SafeBricks generates this code automatically using a configuration file supplied by the client enterprise.

We need to fulfill two more requirements for the guarantees to hold: (i) NFs should not be able to alter the permissions vector during execution, and (ii) NFs should not be able to parse packet buffers arbitrarily—for example, an NF that has permissions only for IP headers should not be able to incorrectly parse TCP headers as IP, thereby circumventing the policy. SafeBricks therefore does not expose these operations to NFs. NFs in NetBricks invoke the `parse` operator to cast packet buffers into protocol structures before processing them. In contrast, SafeBricks mandates that packets be parsed as required before being processed by NFs (not shown in Figure 5.6 for simplicity). In Section 5.7, we describe how the SafeBricks loader interleaves NFs with `parse` nodes and stitches them together into a directed graph based on enterprise-supplied configuration data.

Runtime overhead. The permissions vector leverages portions of the packet buffers reserved for metadata, and hence does not lead to any memory allocation overhead. Setting and verifying permissions, however, lead to a small overhead at runtime: setting the permissions vector before each NF via the `wList` operator increases the depth of the NF graph, and verifying the permission adds an extra check as all requests are mediated by the Controller. As we see in Section 5.9.2.3, the impact on performance is small for real applications.

5.7 SafeBricks: System Bootstrap Protocol

We now describe the protocol for bootstrapping the overall system. Instead of compiled binaries, SafeBricks needs access to the raw source code of the NFs from the providers so it can pass them through a trusted compiler, which ensures that NFs do not perform unsafe operations and are confined to least privilege access. The natural strategy is to have the client enterprise compile these binaries and upload them to the cloud, as in prior enclave-based systems such as Haven [BPH14]. However, this approach is problematic in our case because NF code is proprietary and the client enterprise may not see it.

To address this problem, the idea in SafeBricks is to run inside the enclave a *meta*-functionality: the enclave assembles the NFs and *compiles them* using a trusted compiler, and only then starts running the resulting code. The key to why this works is that *both* the client enterprise and the NF vendors can invoke the remote attestation procedure to check that the enclave is running an agreed upon SafeBricks loader and compiler (both being public code). In this way, (i) each NF vendor can ensure that the enclave does not run some bad code that exfiltrates the source code to an attacker, and (ii) the client enterprise makes sure the NF vendor cannot change what processing happens in the enclave.

The bootstrap process consists of two phases. In the assembly phase, SafeBricks stitches together NFs obtained from various NF providers and compiles them into a single binary in the *assembly enclave*. In the deployment phase, the compiled binary is loaded into the *deployment enclave*, and the client establishes a secure channel of communication with the NFs.

5.7.1   Phase 1: NF assembly

For assembly, SafeBricks uses a special enclave provisioned with two trusted modules—a loader and a compiler—that combine the NFs into a single binary.

Loader.   The loader exposes a simple API that allows the client enterprise to specify (i) *encrypted* NF source codes, (ii) optionally, unencrypted NF source codes that might be interspersed with the proprietary encrypted NFs, (iii) a configuration file outlining the placement of each NF in the directed graph (when chained together), and (iv) a whitelist of permissions per NF indicating the fields each NF is allowed to access.

For the first two, the loader exposes the following API to the client: `load(name, code, is_encrypted)`. For the third, the client specifies the NF graph as a set of edges: ($name_i \rightarrow name_j$). For the fourth, the client supplies a configuration file with a list of items of type: (`name`, `op`, `proto:field`) where `op` $\in$ [`read`, `write`] and `proto:field` indicates a field within a protocol that access is given to. For example, for a firewall, one such entry is (`firewall`, `read`, `IP:src`), in addition to entries for other fields of the IP header.

The loader decrypts the NFs and stitches them together based on the specified graph, before invoking the compiler. (In Section 5.7.2, we discuss how the enclave obtains the keys to decrypt this code.) In doing so, it adds the following additional nodes to the composite graph: (i) a `toEnclave` node at the root of the graph, (ii) a `toHost` node at the end of the graph, and (iii) `parse` nodes followed by a `wList` node before each NF. The loader infers the arguments to the `parse` and `wList` nodes automatically from the configuration file. Thus, `parse` is run by the trusted SafeBricks framework and not by an NF or the client enterprise, ensuring that the packets are not parsed in an unintended way.

Compiler.   The compiler is a standard Rust compiler that implements a lint prohibiting unsafe code inside the enclave, as discussed in Section 5.6.3. Since launching the compiled binary requires OS support, the binary must be placed into main memory where the OS can access it post compilation. However, giving the OS access to the binary unencrypted would violate NF confidentiality.

In order to maintain the privacy of NF code while still allowing its execution by the OS, we take inspiration from VC3 [SCF$^+$15]. Similarly to VC3, our compiler links the compiled NF code to a small amount of public code NF_{load}, and then encrypts the NF code because it will be placed in main memory for the OS to load in the deployment enclave. We refer to the encrypted code as NF_{priv}. Post compilation, $NF_{load} + NF_{priv}$ are loaded and run in a separate deployment enclave by the OS. NF_{load} will be responsible for decrypting and interfacing with NF_{priv} within the deployment enclave once it's initialized.

The loader and compiler are generic modules independent of the NFs. Hence, the NF providers need to audit them only once, across all customer deployments.
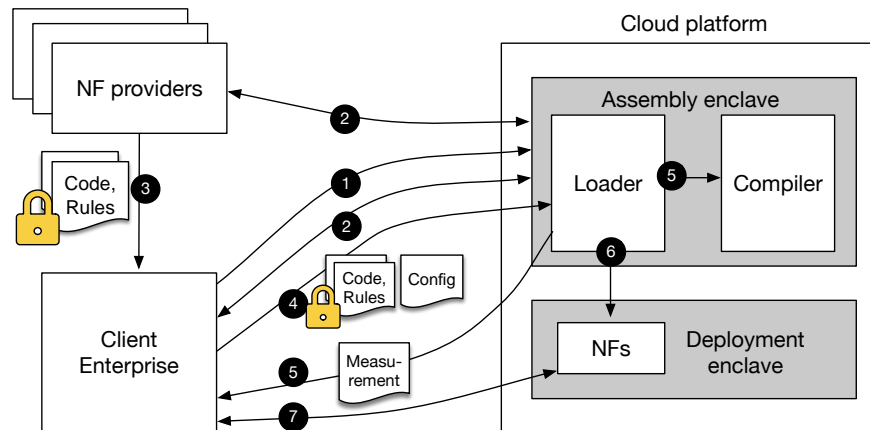
Figure 5.7: SafeBricks's NF assembly and deployment phases during bootstrap. Locks indicate that the data is encrypted.

Assembly protocol. Figure 5.7 illustrates the assembly and deployment protocol. ❶ The cloud provisions an enclave with the SafeBricks loader and compiler modules. ❷ Next, the client as well as the NF providers verify that the loader and compiler have been securely provisioned into the enclave using the remote attestation feature of SGX, as described in Chapter 2. During the attestation, the enclave also returns a securely generated public key to each NF provider. ❸ Each provider then encrypts the NF source code and rulesets with the received public key and submits it to the client enterprise. ❹ The enterprise loads the encrypted source codes and rulesets along with configuration files into the enclave via APIs exposed by the loader module. ❺ The loader decrypts the source codes, stitches them together, and builds and encrypts the assembled code using the compiler, producing $NF_{load} + NF_{priv}$. It then returns to the client a hash measurement of the compiled code so that the client can later verify it once it's deployed in a separate enclave.

5.7.2   Phase 2: NF deployment

❻ The loader finally requests the OS to deploy $NF_{load} + NF_{priv}$ in a separate enclave on the cloud platform. It attests the deployed enclave, establishes a secure channel with NF_{load}, and transfers to it the decryption key for NF_{priv}. NF_{load} decrypts the private code and starts execution. Note that since the assembly enclave attests the deployment enclave and the NF vendors attested the assembly enclave, the NF vendors are assured that the deployment enclave will not send the decrypted binary anywhere but merely run it. ❼ The client then attests the deployed enclave using the measurement it received at the end of the assembly phase, after which it establishes a secure channel of communication with the enclave.

5.8 Security Guarantees

We describe SafeBricks's guarantees assuming the threat model and assumptions in Section 5.2, including the enclave assumption.

SafeBricks's main benefit to confidentiality is that it exposes only encrypted traffic (encrypted with standard encryption) to a cloud attacker, so the attacker does not see the contents of the packets and is limited to observing only packet sizes, timing, and NF access patterns to packets and data. SafeBricks protects in this manner the packet payload and, except in the direct architecture, the header as well.

As with any system with complex processing, encryption does not mean perfect confidentiality because of the existence of side-channels. In Section 5.2.1 we mentioned some categories of side-channels that SafeBricks, and SGX in general, does not protect against. In addition, there are a few other SafeBricks-specific side channels. First, an attacker in SafeBricks knows which (encrypted) packets belong to which flow because each flow is affinitized to an IPSec tunnel for scalability. If this issue is of concern, it can be fixed by using a single tunnel for all flows at the expense of performance. Second, an attacker can measure the time taken by NFs to process a batch of packets. This could leak information in some cases, *e.g.*, whether an expensive regular expression was triggered or not. This is a classical problem, already investigated by prior work [AZM10, BJB15, ZAM11] with common solutions involving padding, *i.e.*, bounding the running time of NFs by executing dummy cycles. Third, an attacker can learn the action taken by an NF, *e.g.*, whether a connection was dropped simply by noticing that fewer packets were sent out. Like many other side-channels, this leakage can also be removed via padding—for example, the gateways could continue sending dummy traffic.

SafeBricks also protects the integrity of the traffic and of the NF processing. A cloud attacker cannot drop, insert, or modify packets, nor can it tamper with NF execution. Integrity of the NFs is guaranteed by SGX, while the integrity of the traffic is guaranteed by the IPSec tunnels between the enclave and the client.

Via the isolation and least privilege design, SafeBricks further ensures that each NF is confined to accessing only parts of the packet the enterprise desires. For the NF vendors, SafeBricks guarantees that NF source codes are hidden from all untrusted parties, including the client enterprise, a cloud attacker or other NF vendors.

5.8.1 Comparison to prior approaches

Prior approaches leak significantly more information about the traffic to the cloud provider than SafeBricks.

Cryptographic approaches. BlindBox [SLPR15] and Embark [LSP$^+$16] encrypt the traffic in a special way that allows the cloud to match encrypted tokens against the traffic and detect if a match occurs. In these schemes, the cloud learns the offset at which any string from any rule in an NF occurs in the packet, regardless of whether or not the rule as a whole matched (rules often contain several such strings). If the rule is known (as in public rulesets), the attacker learns the exact string at that offset in the packet. Even if the rule string is not known, the attacker learns its frequency, which

could lead to decryption via frequency analysis. Assuming an enclave employing side-channel protections as in Section 5.2.1, SafeBricks does not reveal this information. The attacker does not know which rule or part of a rule triggered on a packet. Moreover, BlindBox and Embark do not protect against *active attackers* who modify the traffic flow and, for example, drop packets.

We remark, however, that these prior approaches rely on cryptography alone, and not on trusted hardware as SafeBricks, which makes it much more challenging for them to achieve the properties SafeBricks achieves.

mcTLS [NSV$^+$15] aims to provide least privilege in a setting where each NF is a separate hardware middlebox and belongs to a *different trust perimeter*. Running mcTLS in the cloud in software, however, removes essentially all its security guarantees: the cloud receives the *union* of the permissions of all NFs, which often, is everything.

5.9   Evaluation

We now measure the impact of SafeBricks on NF performance versus an insecure baseline. We also measure the reduction in TCB size as a result of our design. We do not discuss the performance of SafeBricks's gateway as the protocols it implements are well understood.

5.9.1   Setup

We evaluate the performance of SafeBricks using SGX hardware on a single-socket server provisioned with an Intel Xeon E3-1280 v5 CPU with 4 cores running at 3.7GHz. We disable hyperthreading for our experiments. The server has 64GB of memory, and runs Ubuntu 14.04.1 LTS with Linux kernel version 4.4. The hardware supports the SGX v1 instruction set which does not allow dynamic page allocation. Further, the total enclave page cache memory (EPC) available to all enclaves is limited to ∼94MB. For test traffic, we use another server that runs a DPDK-based traffic generator and is directly connected to the SGX machine via Intel XL710 40Gb NICs. The SGX machine acts as the cloud, and the traffic generator is both source and sink for the client traffic.

5.9.2   Performance

We evaluate the performance of SafeBricks using (i) synthetic traces of different packet sizes, from 64B to 1KB, and (ii) the ICTF 2010 trace [ICT], captured during a wide-area security competition and commonly used in academic research. We report throughput in millions of packets per second (Mpps). In all experiments, we exchange traffic between the traffic generator and the SGX machine over an encrypted tunnel (per Section 5.3). As a result, the size of each packet exchanged between the enterprise and the cloud increases by a fixed amount, equal to the headers added by the IPSec protocol.

We compare SafeBricks against an insecure baseline comprising vanilla NetBricks augmented with support for the encrypted tunnel. The baseline represents a setup in which traffic is sent to the
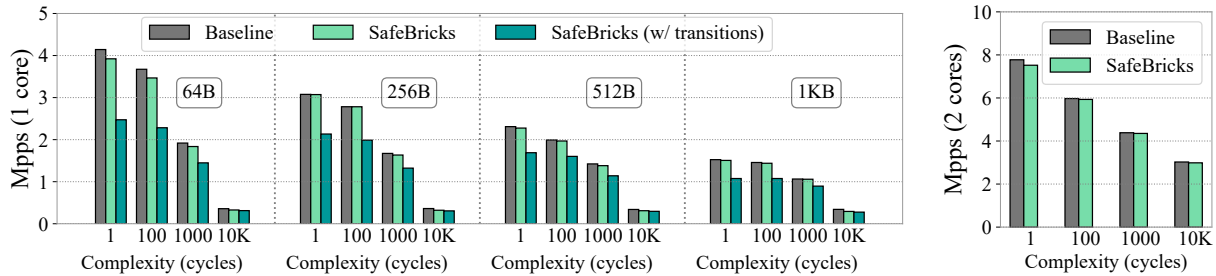
Figure 5.8: (**Left**) SafeBricks framework performance on 1 core compared to the baseline across different packet sizes, and with increasing NF complexity (*i.e.*, processing time in CPU cycles). (**Right**) Performance with 64B packets and NFs on 2 cores.

cloud over an encrypted channel (hence safe from network attackers), but lacks the protection of SafeBricks at the cloud. Finally, we report the median of 10 iterations for each experiment.

5.9.2.1 Framework overheads

We first measure the overhead introduced by SafeBricks as a result of redesigning the core NetBricks framework. To illustrate the benefits of our architecture, we also compare the overheads of the strawman approach that performs packet I/O via enclave transitions (per Section 5.5).

The net overhead of both approaches varies with the complexity of NFs and the latency the NF introduces as a result of packet processing. In this experiment, we use CPU cycles as a proxy for NF complexity, and evaluate a simple NF that first modifies each batch of packets by interchanging the source and destination IP addresses, and then loops for a given number of cycles. We use packet batches of size 32 for both NetBricks and SafeBricks.

Figure 5.8 (left) presents the results with varying packet sizes when the NF is deployed on a single core, and Figure 5.8 (right) shows the performance for 64B packets when the deployment is scaled to two cores. In the worst case with 64B packets and a delay of 1 cycle, the overhead introduced by SafeBricks is < 5%. As the processing time begins to dominate (with increasing NF complexity), the overhead of SafeBricks becomes negligible.

The results also confirm that the design of SafeBricks outperforms the strawman approach, the overhead of which is ∼40% in the worst case. It's worth noting, however, that the relative overhead of the strawman approach decreases with larger packet sizes, as the rate of I/O falls.

5.9.2.2 Impact on real NFs

Unlike the simple NF in the previous experiment, real NFs have varying state requirements. Since the sizes of both the processor caches and enclave memory are limited, the overheads of SafeBricks are also governed by the memory access patterns of the NFs in addition to their complexity. In particular, L3 cache misses are more expensive for enclave applications because cache lines need to be encrypted/decrypted before being evicted/loaded. In this experiment, we characterize the effect of state on the performance of SafeBricks by evaluating the following sample applications:
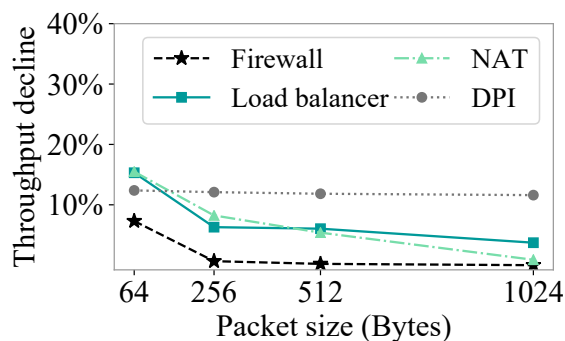
Figure 5.9: Normalized overhead (Mpps) across NFs for different packet sizes.
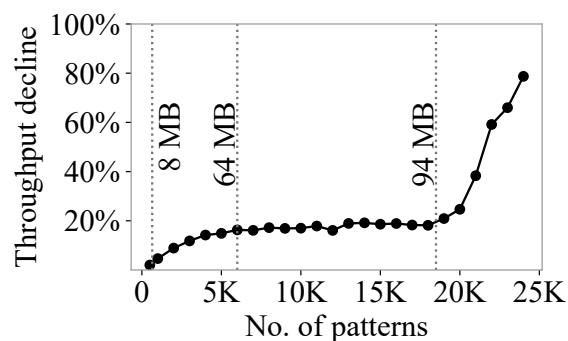
Figure 5.10: DPI performance (Mpps) on the ICTF trace, with increasing no. of rules.
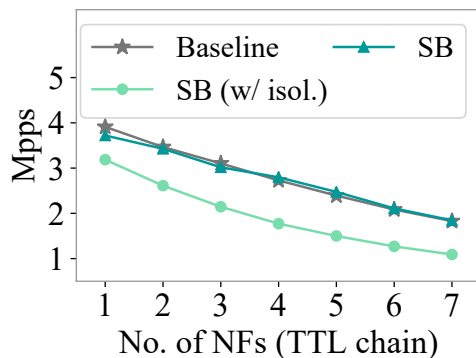
Figure 5.11: Cost of least privilege with increasing no. of NFs.

- **Firewall**: We use a stateful firewall application that linearly scans a list of access control rules and drops connections if it finds a match. Otherwise, it caches the connection. We evaluate it using a ruleset we obtained from our department (643 rules).

- **DPI**: We use a simple deep packet inspection (DPI) application that implements the Aho-Corasick pattern matching algorithm [AC75] on incoming packets, similar to the core signature matching component of the Snort IDS [Roe99]. We evaluate the DPI using patterns extracted from the Snort Community ruleset [Sno].

- **NAT**: Our implementation is based on MazuNAT [Maz] and maps incoming IP addresses to a single public IP address.

- **Load balancer**: We use a partial implementation of Google's Maglev [EYC+16], that spreads traffic between backends using a consistent hashing lookup table.

Figure 5.9 shows the normalized overhead of SafeBricks on application performance across different packet sizes with synthetic traffic. Table 5.1 summarizes the worst-case results corresponding to 64B packets. Table 5.1 also presents the performance results with the ICTF trace. Across applications,

	Synthetic (64B packets)		ICTF trace	
NF	Baseline	SB	Baseline	SB
Firewall	3.86	3.58	1.96	1.93
DPI	1.10	0.96	0.29	0.25
NAT	3.80	3.21	1.97	1.80
Maglev	3.59	3.04	1.92	1.73

Table 5.1: Performance of sample NFs (Mpps)
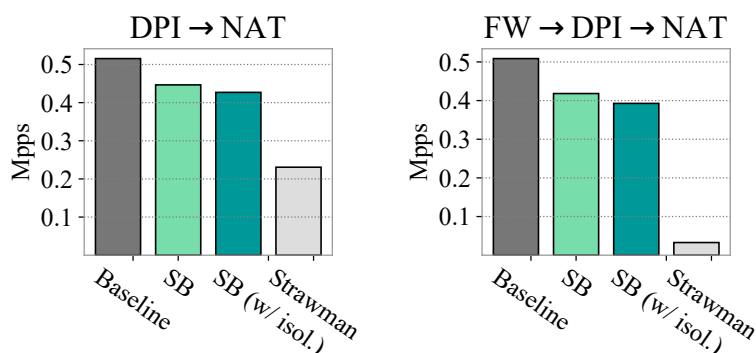
Figure 5.12: Cost of least privilege across NF chains (2 cores)

the overhead ranges between an acceptable ∼0–15% for both synthetic and real traffic, and is a result of page faults triggered by L3 cache misses.

Impact of larger memory footprint. In the previous experiment, the working sets of the applications exceeded the L3 cache but remained less than the size of the EPC (∼94MB). However, accessing memory beyond the EPC is doubly expensive because evicted EPC pages need to be encrypted and integrity-protected. We now assess the impact of a large memory footprint using the DPI application. The application builds a finite state machine over all the patterns in the ruleset, and as such has a significantly larger memory footprint than other NFs.

Figure 5.10 shows the results of our experiment using an increasing number of rules from the Emerging Threats ruleset [Eme] and the ICTF trace. At 18K rules, the working set of the DPI breached the ∼94MB EPC boundary causing its performance to sharply deteriorate thereafter.

This experiment indicates the limits of SafeBricks with regard to the nature of applications it can efficiently support. That is, the cost of securing applications with a memory footprint larger than the EPC size is high. However, we note that the ∼94MB limit is only an artifact of existing hardware and isn't fundamental to SGX enclaves. The next generation of SGX machines is likely to support larger EPC sizes.

5.9.2.3 Cost of NF isolation

We now evaluate the overhead as a result of our mechanisms for enforcing least privilege. Given a chain of NFs, SafeBricks increases the overall depth of the NF graph by one node per NF (Section 5.6.3). In this experiment, we first measure this extra cost as a function of the length of the NF chain. We then compare our approach against an mcTLS-like strawman that relies on encryption for selectively exposing packet fields to NFs (Section 5.6.1).

Effect of chain length.   For this part of the experiment, we use a simple NF that decrements the time-to-live (TTL) field in the IP header of each packet, composed together into chains of varying length. Before executing subsequent NFs in the chain, SafeBricks whitelists access to the TTL field in the permissions vector per packet.

Figure 5.11 compares the performance of SafeBricks with and without least privilege. Since the NF is stateless, in the absence of isolation SafeBricks does not introduce any discernible overhead against the baseline. With least privilege enforcement, the latency added by the additional nodes increases as the length of the chain increases. Consequently, the overhead climbs from ∼14–40% as the chain grows to a size of seven NFs. We note that these numbers represent an upper bound on the overhead of SafeBricks. As we show in the next part of this experiment, the percentage overhead is much smaller for real, more complex NFs.

Comparison with encryption-based strawman.   We now measure the performance of SafeBricks using a chain of real NFs each of which accesses different parts of packets—a firewall (given read permissions on packet headers), a DPI (with read permissions on both headers and payload), and a NAT (with read and write permissions on packet headers). The NF implementations are identical to the ones described in Section 5.9.2.2.

To quantify the benefit of our approach for enforcing least privilege, we also compare SafeBricks to an mcTLS-like strawman in which each NF in the chain is run in a separate enclave (as described in Section 5.6.1). In all setups (including the baseline), we allocate two cores for running the NFs, and reserve one core for I/O.

Figure 5.12 shows the results with two different chains: (i) a DPI followed by a NAT, and (ii) a firewall chained to a DPI and then a NAT. In the former scenario, SafeBricks results in an overhead of 15% in the absence of least privilege enforcement. With least privilege, the throughput declines by a further 3%, confirming that the cost of enforcing least privilege across real NFs is minimal. In contrast, an mcTLS-like approach (with each NF running in a separate enclave, affinitized to distinct cores) results in a sharper decline of 2.2× the performance being bottlenecked at the DPI along with the added encryption and copying of packets as they move across NFs in different enclaves. In the latter scenario with three NFs in a chain, the performance of the strawman approach falls further, by 16×. In this scenario, however, the NFs (and hence enclaves) outnumbered the available cores in our setup, leading to resource contention.

5.9.3 Comparison with BlindBox and Embark

Both SafeBricks and Embark tunnel packets to a third-party service in the cloud. For the ICTF trace, IPSec tunneling inflates the bandwidth by 16% due to both encryption and encapsulation. Embark

introduces a further 20-byte overhead per IPv4 packet because it converts them to IPv6, resulting in a net overhead of 21%. BlindBox, in contrast, does not pay the cost of tunneling as it is targeted at in-network DPI applications. However, the BlindBox encryption protocol (also used by Embark for DPI processing) inflates bandwidth consumption by up to $5\times$ in the worst case, unlike SafeBricks which only uses standard encryption schemes.

As regards throughput, both Embark and BlindBox are competitive with unencrypted baseline NFs and incur negligible overhead, whereas SafeBricks impacts performance by $\sim$0–15% across NFs due to its use of SGX enclaves (Section 5.9.2.2). At the same time, both BlindBox and Embark impact performance at the client considerably—with BlindBox, client endpoints need to implement its special encryption protocols over and above TLS and take $30\times$ longer to encrypt a packet; Embark centralizes this overhead at the enterprise's gateway instead. Clients do not need to pay these costs with SafeBricks.

5.9.4 TCB size

SafeBricks involves the use of two types of enclaves: one for assembling the NFs during system bootstrap (per Section 5.7), and another for deploying the NFs. The assembly enclave primarily contains the Rust compiler, which is necessarily part of the TCB of applications with or without SafeBricks. The deployment enclave, on the other hand, represents the TCB which we aim to reduce in redesigning the NetBricks framework.

To evaluate the reduction in TCB, we thus compare the size of the deployment enclave components in SafeBricks with that of NetBricks. The size of the enclave binary in SafeBricks is $\sim$1MB. In comparison, the aggregate size of NetBricks components is 21.3MB, representing a TCB reduction of over $20\times$. The reduction can largely be attributed to the exclusion of DPDK from the TCB as a result of partitioning NetBricks, which itself comprises $\sim$516K LoC. Furthermore, by designing for our specific use case, we avoid including a library OS within our trust perimeter, the size of which can be as large as 209MB (as in Haven [BPH14]).

5.10 Limitations and Future Work

SafeBricks inherits three primary limitations owing to its use of Intel SGX.

First, enclave memory is limited to $\sim$94MB in existing hardware, making SafeBricks impractical for applications with larger working sets. Exploring alternate architectures that combine cryptographic approaches and SGX, thereby reducing the memory burden on the enclaves, is an interesting open problem in this context.

Second, SafeBricks is unsuitable for NFs relying on operations that are illegal within SGX enclaves, such as system calls and timestamps. Though SafeBricks supports timestamps, it can only ensure their monotonicity and not correctness.

Third, SGX enclaves, and consequently SafeBricks, are vulnerable to side-channel attacks (per Chapter 2). Though a number of potential solutions have been proposed in recent work [CLD16, SCNS16, SLKP17, GLS$^+$17a, CZRZ17], their impact on application performance is often non-trivial.

Investigating the viability of these proposals in the NFV context, or developing targeted solutions for NFs is potential future work.

5.11   Related Work

We divide related work largely into two categories: (i) cryptographic approaches for securing NFs, and (ii) proposals based on trusted hardware. We do not discuss the mcTLS protocol [NSV$^+$15] further as we have already compared SafeBricks with mcTLS in Section 5.6 and Section 5.8.

Cryptographic approaches.   Recent systems propose the use of cryptographic schemes that enable NFs to operate directly over encrypted traffic [MADCK16, SLPR15, LSP$^+$16, AMS$^+$16, YWLW16]. When compared to SafeBricks, these approaches have the advantage that they do not rely on trusted hardware. However, this comes with two significant limitations. (1) Regarding functionality, they only support simple operations such as "=" and "≥", and are unable to support more sophisticated computations such as regular expressions, scripts, or application-level logic. As a result, they are not applicable to a wide range of NFs (*e.g.*, modern IDSes, application-level firewalls, etc.). To provide full functionality with cryptography, one needs schemes such as fully-homomorphic encryption [Gen09], which is orders of magnitude too slow. (2) Regarding security, we explained in Section 5.8 how these systems leak more information to the cloud than SafeBricks.

Trusted hardware proposals for legacy applications.   Other work has shown how to use hardware enclaves to run applications in the cloud without having to trust the cloud provider [BPH14, HZX$^+$16, ATG$^+$16, STTS17, OLMS17, TPV17]. The mandate of these systems is to support arbitrary, legacy applications instead of optimizing for any in particular. As a result, some of these systems inflate the size of the TCB by introducing a library OS within the enclave (to support illegal enclave instructions), or impact performance because of enclave transitions.

Trusted hardware proposals for network applications.   Recent work has proposed the use of hardware enclaves for securing network applications. Kim *et al.* use SGX to enhance the security of Tor [KHH$^+$17], and also identify NFs as a potential use case [KSH$^+$15]. Other proposals develop prototypes for specific functions: Coughlin *et al.* [CKW17] present a proof-of-concept Click element for pattern matching within enclaves; and Shih *et al.* [SKKG16] propose SGX for isolating the state of NFs, applying it to a subset of the Snort IDS. In contrast, SafeBricks is a *general*-purpose framework that additionally enforces least privilege across NFs. At the same time, SafeBricks balances the interests of NF vendors by maintaining the confidentiality of NF code and rulesets.

Concurrent to our work, SGX-Box [HKHH17] and ShieldBox [TKG$^+$18] also propose frameworks for executing NFs within enclaves. SGX-Box [HKHH17] does not explicitly handle NF isolation or chaining; ShieldBox integrates SGX with Click and isolates each NF in a separate enclave. In such cases, ShieldBox reports a throughput decline of up to $3\times$. SafeBricks, in contrast, avoids this overhead by isolating NFs within the same enclave with the help of language-based enforcement. However, unlike SafeBricks, ShieldBox also supports NFs with system calls by leveraging the Scone framework [ATG$^+$16]. Both SGX-Box and ShieldBox also allow NFs to access entire packets, while SafeBricks enforces least privilege.

5.12 Summary

In summary, SafeBricks leverages a combination of hardware enclaves and language-based enforcement to shield network functions in the cloud. SafeBricks is suitable for a wide range of commonly used NFs, and our evaluation demonstrates that it is practical, adding an overhead of $\sim$0%–15% across applications.

Chapter 6

Encrypted Video Analytics and Machine Learning

This chapter presents Visor, a system that extends trusted execution to workloads that require hardware accelerators (such as machine learning and video analytics). In addition, the design of Visor illustrates how to efficiently mitigate side-channel leakage in enclave-based systems.

6.1   Introduction

Cameras are being deployed pervasively for the many applications they enable, such as traffic planning, retail experience, and enterprise security [Vis, Tel, Ver]. Videos from the cameras are streamed to the cloud, where they are processed using video analytics pipelines [ZAB⁺17, JAB⁺18, KEA⁺17] composed of computer vision techniques (e.g., OpenCV [Ope]) and convolutional neural networks (e.g., object detector CNNs [RDGF16]); as illustrated in Figure 6.1. Indeed, "video-analytics-as-a-service" is becoming an important offering for cloud providers [Micc, Ama].

Privacy of the video contents is of paramount concern in the "video analytics-as-a-service" offerings. Videos often contain sensitive information, such as users' home interiors, people in workspaces, or license plates of cars. For example, the Kuna home monitoring service [Kun] transmits videos from users' homes to the cloud, analyzes the videos, and notifies users when it detects movement in areas of interest. For user privacy, video streams must remain *confidential* and not be revealed to the cloud provider or other co-tenants in the cloud.

Trusted execution environments (TEEs) [MAB⁺13, VVB18] are a natural fit for privacy-preserving video analytics in the cloud. In contrast to cryptographic approaches, such as homomorphic encryption, TEEs rely on the assumption that cloud tenants also trust the hardware. The hardware provides the ability to create secure "enclaves" that are protected against privileged attackers. TEEs are more compelling than cryptographic techniques since they are orders of magnitude faster. In fact, CPU TEEs (e.g., Intel SGX [MAB⁺13]) lie at the heart of confidential cloud computing [IBM, Micb]. Meanwhile, recent advancements in GPU TEEs [VVB18, JTK⁺19] enable the execution of ML models (e.g., neural networks) with strong privacy guarantees as well. CPU and

GPU TEEs, thus, present an opportunity for building privacy-preserving video analytics systems.

Unfortunately, TEEs (e.g., Intel SGX) are vulnerable to a host of side-channel attacks (e.g., [WCP⁺17, BMD⁺17, BMW⁺18, XCP15]). For instance, in Section 6.2.3 we show that by observing just the memory access patterns of a widely used bounding box detection OpenCV module, an attacker can infer the *exact shapes and positions of all moving objects* in the video. In general, an attacker can infer crucial information about the video being processed, such as the times when there is activity, objects that appear in the video frame, all of which when combined with knowledge about the physical space being covered by the camera, can lead to serious violations of confidentiality.

We present Visor, a system for privacy-preserving video analytics services. Visor protects the confidentiality of the videos being analyzed from the service provider and other co-tenants. When tenants host their own CNN models in the cloud, it also protects the model parameters and weights. Visor protects against a powerful enclave attacker who can compromise the software stack outside the enclave, as well as observe any *data-dependent accesses* to network, disk, or memory via side-channels (similar to prior work [OSF⁺16, RLT15]).

Visor makes two primary contributions, combining insights from ML systems, security, computer vision, and algorithm design. First, we present a privacy-preserving framework for machine-learning-as-a-service (MLaaS), which supports CNN-based ML applications spanning both CPU and GPU resources. Our framework can potentially power applications beyond video analytics, such as medical imaging, recommendation systems, and financial forecasting. Second, we develop novel *data-oblivious* algorithms with provable privacy guarantees within our MLaaS framework, for commonly used vision modules. The modules are efficient and can be composed to construct many different video analytics pipelines. In designing our algorithms, we formulate a set of design principles that can be broadly applied to other vision modules as well.

1) Privacy-Preserving MLaaS Framework. Visor leverages a *hybrid* TEE that spans CPU and GPU resources available in the cloud. Recent work has shown that scaling video analytics pipelines requires judicious use of both CPUs and GPUs [PCHF18, HAB⁺18]. Some pipeline modules can run on CPUs at the required frame rates (*e.g.*, video decoding or vision algorithms) while others (*e.g.*, CNNs) require GPUs, as shown in Figure 6.1. Thus, our solution spans both CPU and GPU TEEs, and combines them into a unified trust domain.

Visor systematically addresses access-pattern-based leakage across the components of the hybrid TEE, from video ingestion to CPU-GPU communication to CNN processing. In particular, we take the following steps:

a) Visor leverages a suite of data-oblivious primitives to remove access pattern leakage from the CPU TEE. The primitives enable the development of oblivious modules with provable privacy guarantees, the access patterns of which are always independent of private data.

b) Visor relies on a novel oblivious communication protocol to remove leakage from the CPU-GPU channel. As the CPU modules serve as filters, the data flow in the CPU-GPU channel (on which objects of each frame are passed to the GPU) leaks information about the contents of each frame, enabling attackers to infer the number of moving objects in a frame. At a high level, Visor pads the channel with dummy objects, leveraging the observation that our

application is not constrained by the CPU-GPU bandwidth. To reduce GPU wastage, Visor intelligently minimizes running the CNN on the dummy objects.

c) Visor makes CNNs running in a GPU TEE oblivious by leveraging *branchless* CUDA instructions to implement conditional operations (e.g., ReLU and max pooling) in a data-oblivious way.

2) Efficient Oblivious Vision Pipelines. Next, we design novel data-oblivious algorithms for vision modules that are foundational for video analytics, and implement them using the oblivious primitives provided by the framework described above. Vision algorithms are used in video analytics pipelines to extract the moving foreground objects. These algorithms (*e.g.*, background subtraction, bounding box detection, object cropping, and tracking) run on CPUs and serve as cheap *filters* to discard frames instead of invoking expensive CNNs on the GPU for each frame's objects (more in Section 6.2.1). The modules can be composed to construct various vision pipelines, such as medical imaging and motion tracking.

As we demonstrate in Section 6.8, naïve approaches for making these algorithms data-oblivious, such that their operations are independent of each pixel's value, can slow down video pipelines by several orders of magnitude. Instead, we carefully craft oblivious vision algorithms for each module in the video analytics pipeline, including the popular VP8 video decoder [BWX11]. Our overarching goal is to transform each algorithm into a pattern that processes each pixel identically. To apply this design pattern efficiently, we devise a set of algorithmic and systemic optimization strategies based on the properties of vision modules, as follows. First, we employ a divide-and conquer approach—*i.e.*, we break down each algorithm into independent subroutines based on their functionality, and tailor each subroutine individually. Second, we cast sequential algorithms into a form that *scans* input images while performing identical operations on each pixel. Third, identical pixel operations allow us to systemically amortize the processing cost across groups of pixels in each algorithm. For each vision module, we derive the operations applied per pixel in conjunction with these design strategies. Collectively, these strategies improve performance by up to $1000\times$ over naïve oblivious solutions. We discuss our approach in more detail in Section 6.5; nevertheless, we note that it can potentially help inform the design of other oblivious vision modules as well, beyond the ones we consider in Visor.

In addition, as shown by prior work, bitrate variations in encrypted network traffic can also leak information about the underlying video streams [SST17], beyond access pattern leakage at the cloud. To prevent this leakage, we modify the video encoder to carefully pad video streams *at the source* in a way that optimizes the video decoder's latency. Visor thus provides an end-to-end solution for private video analytics.

Evaluation Highlights. We have implemented Visor on Intel SGX CPU enclaves [MAB$^+$13] and Graviton GPU enclaves [VVB18]. We evaluate Visor on commercial video streams of cities and datacenter premises containing sensitive data. Our evaluation shows that Visor's vision components perform up to $1000\times$ better than naïve oblivious solutions, and over 6 to 7 orders of magnitude better than a state-of-the-art general-purpose system for oblivious program execution. Against a *non-oblivious baseline*, Visor's overheads are limited to $2\times$–$6\times$ which still enables us to analyze

(a) Pipeline with object classifier (e.g., ResNet).

(b) Pipeline with object detector (e.g., Yolo).
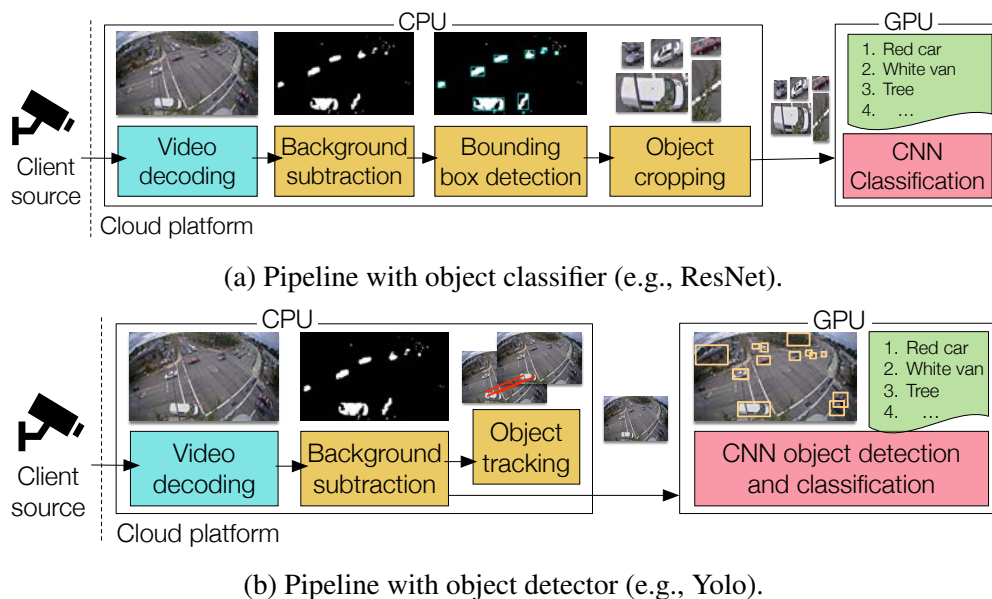
Figure 6.1: Video analytics pipelines. Pipeline (a) extracts the objects using vision algorithms and classifies the cropped objects using a CNN classifier on the GPU. Pipeline (b) also uses the vision algorithms as a filter, but sends the entire frame to the CNN detector. Both pipelines may optionally use object tracking.

multiple streams simultaneously in real-time on our testbed. Visor is versatile and can accommodate different combinations of vision components used in real-world applications. Thus, Visor provides an efficient solution for private video analytics.

6.2 Background and Motivation

6.2.1 Video Analytics as a Service

Figure 6.1 depicts the canonical pipelines for video analytics [HAB$^+$18, KEA$^+$17, ZAB$^+$17, ZJR$^+$18, Micd]. The client (*e.g.*, a source camera) feeds the video stream to the service hosted in the cloud, which (a) decodes the video into frames, (b) extracts objects from the frames using vision algorithms, and (c) classifies the objects using a pre-trained convolutional neural network (CNN). Cameras typically offer the ability to control the resolution and frame rate at which the video streams are encoded.

Recent work demonstrates that scaling video analytics pipelines requires judicious use of both CPUs and GPUs [PCHF18, HAB$^+$18]. In Visor, we follow the example of Microsoft's Rocket platform for video analytics [Micd, Mice]—we split the pipelines by running video decoding and vision modules on the CPU, while offloading the CNN to the GPU (as shown in Figure 6.1). The vision modules process each frame to detect the moving "foreground" objects in the video using

background subtraction [BBV08], compute each object's bounding box [SA85], and crop them from the frame for the CNN classifier. These vision modules can sustain the typical frame rates of videos even on CPUs, thereby serving as vital "filters" to reduce the expensive CNN operations on the GPU [HAB+18, KEA+17], and are thus widely used in practical deployments. For example, CNN classification in Figure 6.1a is invoked only if moving objects are detected in a region of interest in the frame. Optionally, the moving objects are also tracked to infer directions (say, cars turning left). The CNNs can either be object classifiers (e.g., ResNet [HZRS16]) as in Figure 6.1a; or object detectors (*e.g.*, Yolo [RDGF16]) as in Figure 6.1b, which take whole frames as input. The choice of pipeline modules is application dependent [JAB+18, HAB+18] and Visor targets confidentiality for all pipeline modules, their different combinations, and vision CNNs.

While our description focuses on a multi-tenant cloud service, our ideas equally apply to multi-tenant *edge compute* systems, say, at cellular base stations [ETS]. Techniques for lightweight programmability on the cameras to reduce network traffic (*e.g.*, using smart encoders [Viv] or dynamically adapting frame rates [ABB+17]) are orthogonal to Visor's techniques.

6.2.2 Trusted Execution Environments

As described in Chapter 2, trusted execution environments, or enclaves, protect application's code and data from all other software in a system. Code and data loaded in an enclave—CPU and GPU TEEs—can be verified by clients using the *remote attestation* feature. Here, we briefly introduce the enclave implementations that Visor builds upon.

Intel SGX. [MAB+13] enables TEEs on CPUs and enforces isolation by storing enclave code and data in a protected memory region called the Enclave Page Cache (EPC). The hardware ensures that no software outside the enclave can access EPC contents.

Graviton. [VVB18] enables TEEs on GPUs in tandem with trusted applications hosted in CPU TEEs. Graviton prevents an adversary from observing or tampering with traffic (data and commands) transferred to/from the GPU. A trusted GPU runtime (*e.g.*, CUDA runtime) hosted in a CPU TEE attests that all code/data have been securely loaded onto the GPU.

6.2.3 Attacks based on Access Pattern Leakage

In Chapter 2 we described in detail how existing TEEs are vulnerable to a host of side-channel attacks. A large subset of these attacks exploit *data-dependent memory access patterns* (*e.g.*, branch-prediction, cache-timing, or controlled page fault attacks). We now demonstrate how devastating this leakage can be for video analytics pipelines. In particular, we analyzed the impact of access pattern leakage at cache-line granularity [GESM17, BMD+17, SWG+17, MIE17] on the bounding box detection algorithm [SA85] (see Figure 6.1a; Section 6.2.1). We simulated existing attacks by capturing the memory access trace during an execution of the algorithm, and then examined the trace to reverse-engineer the contents of the input frame. Since images are laid out predictably in memory, we found that the attacker is able to infer the locations of all the pixels touched during execution, and thus, the *shapes and positions of all objects* (as shown in Figure 6.2). Shapes and

Figure 6.2: Attacker obtains all the frame's objects (right) using access pattern leakage in the bounding box detection module.

positions of objects are the core content of any video, and allow the attacker to infer sensitive information like times when patients are visiting private medical centers or when residents are inside a house, and even infer if the individuals are babies or on wheelchairs based on their size and shapes. In fact, conversations with customers of one of the largest public cloud providers indeed confirm that *privacy of the videos is among their top-two concerns* in signing up for the video analytics cloud service.

6.3 Threat Model and Security Guarantees

We describe the attacker's capabilities and lay out the attacks that are in scope and out of scope for our work.

6.3.1 Hardware Enclaves and Side-Channels

Our trusted computing base includes: (*i*) the GPU package and its enclave implementation, (*ii*) the CPU package and its enclave implementation, and (*iii*) the video analytics pipeline implementation and GPU runtime hosted in the CPU enclave.

The design of Visor is not tied to any specific hardware enclave; instead, Visor builds on top of an *abstract* model of hardware enclaves where the attacker controls the server's software stack outside the enclave (including the OS), but cannot perform any attacks to glean information from inside the processor including processor keys (similar to our abstract enclave assumption in the previous chapter). However, unlike our assumption in the previous chapter, we assume that the attacker can additionally observe the contents and access patterns of all (encrypted) pages in memory, for both data and code. In particular, we assume that the attacker can observe the enclave's memory access patterns at cache line granularity [OSF+16]. Note that our attacker model includes the cloud service provider as well as other co-tenants.

We instantiate Visor with the widely-deployed Intel SGX enclave. However, recent attacks show that SGX does not quite satisfy the abstract enclave model that Visor requires. For example, attackers may be able to distinguish *intra* cache line memory accesses [YGH16, MWES18]. In Visor, we mitigate these attacks by disabling hyperthreading in the underlying system, disallowing attackers from observing intra-core side-channels; clients can verify that hyperthreading is disabled during

remote attestation [Att]. One may also employ complementary solutions for closing hyperthreading-based attacks [OTK$^+$18, CWC$^+$18].

Other attacks that violate our abstract enclave model are out of scope: such as attacks based on timing analysis or power consumption [MOG$^+$20, TSS17], DoS attacks [JLLK17, GLS$^+$17b], or rollback attacks [PLD$^+$11] (which have complementary solutions [BCLK17, MAK$^+$17]). Transient execution attacks (*e.g.*, [BMW$^+$18, SLM$^+$19, CCX$^+$19, VBMS$^+$20, RMR$^+$21, vSMO$^+$19, vSMK$^+$20]) are also out of scope; these attacks violate the threat model of SGX and are typically patched promptly by the vendor via microcode updates (see Chapter 2 for a detailed discussion). In the future, one could swap out Intel SGX in our implementation for upcoming enclaves such as MI6 [BLW$^+$19] and Keystone [LKS$^+$19] that address many of the above drawbacks of SGX.

Visor provides protection against *any channel of attack that exploits data-dependent access patterns* within our abstract enclave model, which represent a large class of known attacks on enclaves (*e.g.*, cache attacks [GESM17, BMD$^+$17, SWG$^+$17, MIE17, HCP17], branch prediction [LSG$^+$17], paging-based attacks [XCP15, BWK$^+$17], or memory bus snooping [LJF$^+$20]). We note that even if co-tenancy is disabled (which comes at considerable expense), privileged software such as the OS and hypervisor can still infer access patterns (*e.g.*, by monitoring page faults), thus still requiring data-oblivious solutions.

Recent work has shown side-channel leakage on GPUs [NKAG17, NNQAG18, JFK17, JFK16] including the exploitation of data access patterns out of the GPU. We expect similar attacks to be mounted on GPU *enclaves* as video and ML workloads gain in popularity, and our threat model applies to GPU enclaves as well.

6.3.2 Video Streams and CNN Model

Each client owns its video streams, and it expects to protect its video from the cloud and co-tenants of the video analytics service. The vision algorithms are assumed to be public.

We assume that the CNN model's architecture is public, but its weights are private and may be proprietary to either the client or the cloud service. Visor protects the weights in both scenarios within enclaves, in accordance with the threat model and guarantees from Section 6.3.1; however, when the weights are proprietary to the cloud service, the client may be able to learn some information about the weights by analyzing the results of the pipeline [TZJ$^+$16, FJR15, FLJ$^+$14]. Such attacks are out of scope for Visor.

Finally, recent work has shown that the camera's encrypted network traffic leaks the video's bitrate variation to an attacker observing the network [SST17], which may consequently leak information about the video contents. Visor eliminates this leakage by padding the video segments *at the camera*, in such a way that optimizes the latency of decoding the padded stream at the cloud (Section 6.6.1).

6.3.3 Provable Guarantees for Data-Obliviousness

Visor provides *data-obliviousness* within our abstract enclave model from Section 6.3.1, which guarantees that the memory access patterns of enclave code does not reveal any information about

sensitive data. We rely on the enclaves themselves to provide integrity, along with authenticated encryption.

We formulate the guarantees of data-obliviousness using the "simulation paradigm" [Gol04a]. First, we define a *trace of observations* that the attacker sees in our threat model. Then, we define the *public information*, *i.e.*, information we do not attempt to hide and is known to the attacker. Using these, we argue that there exists a simulator, such that for all videos V, when given *only* the public information (about V and the video algorithms), the simulator can produce a trace that is indistinguishable from the real trace visible to an attacker who observes the access patterns during Visor's processing of V. By "indistinguishable", we mean that no polynomial-time attacker can distinguish between the simulated trace and the real trace observed by the attacker. The fact that a simulator can produce the same observations as seen by the attacker *even without knowing the private data in the video stream* implies that the attacker does not learn sensitive data about the video.

In our attacker model, the trace of observations is the sequence of the addresses of memory references to code as well as data, along with the accessed data (which is encrypted). The public information is all of Visor's algorithms, formatting and sizing information, but not the video data. For efficiency, Visor also takes as input some public parameters that represent various upper bounds on the properties of the video streams, *e.g.*, the maximum number of objects per frame, or upper bounds on object dimensions.

In Appendix C, we provide a formal definition of data-obliviousness (Definition 7); a summary of public information for each algorithm; and proofs of security along with detailed pseudocode for each algorithm. Since Visor's data-oblivious algorithms (Section 6.6 and Section 6.7) follow an *identical sequence of memory accesses* that depend only on public information and are *independent* of data content, our proofs are easy to verify.

6.4   A Privacy-Preserving MLaaS Framework

In this section, we present a privacy-preserving framework for machine-learning-as-a-service (MLaaS), that supports CNN-based ML applications spanning both CPU and GPU resources. Though Visor focuses on protecting video analytics pipelines, our framework can more broadly be used for a range of MLaaS applications such as medical imaging, recommendation systems, and financial forecasting.

Our framework comprises three key features that collectively enable data-oblivious execution of ML services. First, it protects the computation in ML pipelines using a *hybrid* TEE that spans both the CPU and GPU. Second, it provides a secure CPU-GPU communication channel that additionally prevents the leakage of information via traffic patterns in the channel. Third, it prevents access-pattern-based leakage on the CPU and GPU by facilitating the development of data-oblivious modules using a suite of optimized primitives.
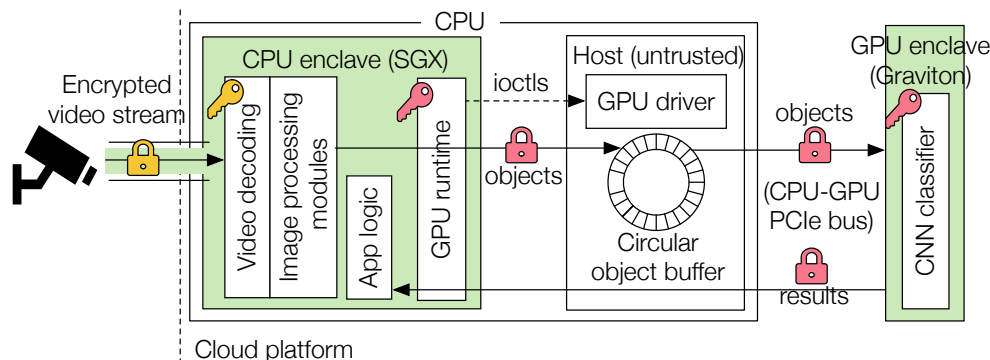
Figure 6.3: Visor's hybrid TEE architecture. Locks indicate encrypted data channels, and keys indicate decryption points.

6.4.1 Hybrid TEE Architecture

Figure 6.3 shows Visor's architecture. Visor receives encrypted video streams from the client's camera, which are then fed to the video processing pipeline. We refer to the architecture as a *hybrid* TEE as it spans both the CPU and GPU TEEs, with different modules of the video pipeline (Section 6.2.1) being placed across these TEEs. We follow the example of prior work that has shown that running the non-CNN modules of the pipeline on the CPU, and the CNNs on the GPU [HAB⁺18, PCHF18, Micd], results in efficient use of the expensive GPU resources while still keeping up with the incoming frame rate of videos.

Regardless of the placement of modules across the CPU and GPU, we note that attacks based on data access patterns can be mounted on *both* CPU and GPU TEEs, as explained in Section 6.3.1. As such, our data-oblivious algorithms and techniques are broadly applicable irrespective of the placement, though our description is based on non-CNN modules running on the CPU and the CNNs on the GPU.

CPU and GPU TEEs. We implement the CPU TEE using Intel SGX enclaves, and the GPU TEE using Graviton secure contexts [VVB18]. The CPU TEE also runs Graviton's trusted GPU runtime, which enables Visor to securely bootstrap the GPU TEE and establish a single trust domain across the TEEs. The GPU runtime talks to the untrusted GPU driver (running on the host outside the CPU TEE) to manage resources on the GPU via `ioctl` calls. In Graviton, each `ioctl` call is translated to a sequence of commands submitted to the command processor. Graviton ensures *secure* command submission (and subsequently `ioctl` delivery) as follows: (*i*) for task submission, the runtime uses authenticated encryption to protect commands from being dropped, replayed, or reordered, and (*ii*) for resource management, the runtime validates signed summaries returned by the GPU upon completion. The GPU runtime *encrypts all inter-TEE communication*.

We port the non-CNN video modules (Figure 6.1) to SGX enclaves using the Graphene Li-bOS [TPV17]. In doing so, we instrument Graphene to support the `ioctl` calls that are used by the runtime to communicate with the GPU driver.

Pipeline execution. The hybrid architecture requires us to protect against attacks on the CPU

TEE, GPU TEE, and the CPU-GPU channel. As Figure 6.3 illustrates, Visor decrypts the video stream inside the CPU TEE, and obliviously decodes out each frame (in Section 6.6). Visor then processes the decoded frames using oblivious vision algorithms to extract objects from each frame (in Section 6.7). Visor extracts the *same* number of objects of *identical dimensions* from each frame (some of which are dummies, up to an upper-bound) and feeds them into a circular buffer. This avoids leaking the *actual* number of objects in each frame and their *sizes*; the attacker can observe accesses to the buffer, even though objects are encrypted. Objects are dequeued from the buffer and sent to the GPU (Section 6.4.2) where they are decrypted and processed obliviously by the CNN in the GPU TEE (Section 6.4.3).

6.4.2 CPU-GPU Communication

Although the CPU-GPU channel in Figure 6.3 transfers encrypted objects, Visor needs to ensure that its traffic patterns are independent of the video content. Otherwise, an attacker observing the channel can infer the processing rate of objects, and hence the number (and size) of the detected objects in each frame. To address this leakage, Visor ensures that (*i*) the CPU TEE transfers the same number of objects to the GPU per frame, and (*ii*) CNN inference runs at a fixed rate (or batch size) in the GPU TEE. Crucially, Visor ensures that the CNN processes as few *dummy objects* as possible. While our description focuses on Figure 6.1a to hide the processing rate of *objects of a frame* on the GPU, our techniques directly apply to the pipeline of Figure 6.1b to hide the processing rate of complete frames using *dummy frames*.

Since the CPU TEE already extracts a fixed number of objects per frame (say k_{max}) for obliviousness, we enforce an inference rate of k_{max} for the CNN as well, regardless of the number of *actual* objects in each frame (say k). The upper bound k_{max} is easy to learn for each video stream in practice. However, this leads to a wastage of GPU resources, which must now also run inference on $(k_{max} - k)$ dummy objects per frame. To limit this wastage, we develop an oblivious protocol that leads to processing as few dummy objects as possible.

Oblivious protocol. Visor runs CNN inference on $k'(<< k_{max})$ objects per frame. Visor's CPU pipeline extracts k_{max} objects from each frame (extracting dummy objects if needed) and pushes them into the head of the circular buffer (Figure 6.3). At a fixed rate (*e.g.*, once per frame, or every 33ms for a 30fps video), k' objects are dequeued from the *tail* of the buffer and sent to the GPU that runs inference on all k' objects.

We reduce the number of dummy objects processed by the GPU as follows. We sort the buffer using osort in ascending order of "priority" values (dummy objects are assigned lower priority), thus moving dummy objects to the *head* of the buffer and actual objects to the *tail*. Dequeuing from the tail of the buffer ensures that actual objects are processed first, and that dummy objects at the head of the buffer are likely *overwritten* before being sent to the GPU. The circular buffer's size is set large enough to avoid overwriting actual objects.

The consumption (or inference) rate k' should be set relative to the actual number of objects that occur in the frames of the video stream. Too high a value of k' results in GPU wastage due to dummy inferences, while too low a value leads to delay in the processing of the objects in the frame

(and potentially overwriting them in the circular buffer). In our experiments, we use a value of $k' = 2 \times k_{\mathsf{avg}}$ (k_{avg} is the average number of objects in a frame) that leads to little delay and wastage.

Bandwidth consumption. The increase in traffic on the CPU-GPU PCIe bus (Figure 6.3) due to additional dummy objects for obliviousness is not an issue because the bus is not bandwidth-constrained. Even with Visor's oblivious video pipelines, we measure the data rate to be $< 70\,\mathrm{MB/s}$, in contrast to the several $\mathrm{GB/s}$ available in PCIe interconnects.

6.4.3 CNN Classification on the GPU

The CNN processes identically-sized objects at a fixed rate on the GPU. The vast majority of CNN operations, such as matrix multiplications, have inherently input-independent access patterns [OSF$^+$16, TGS$^+$18]. The operations that are *not* oblivious can be categorized as conditional assignments. For instance, the ReLU function, when given an input *x*, replaces *x* with *max*(0,x); likewise, the max-pooling layer replaces each value within a square input array with its maximum value.

Oblivious implementation of the *max* operator may use CUDA `max`/`fmax` intrinsics for integers/ floats, which get compiled to `IMNMX`/`FMNMX` instructions [NVI] that execute the *max* operation branchlessly. This ensures that the code is free of data-dependent accesses, making CNN inference oblivious.

6.4.4 Oblivious Modules on the CPU

After providing a data-oblivious CPU-GPU channel and CNN execution on the GPU, we address the video modules (in Figure 6.1) that execute on the CPU. We carefully craft oblivious versions of the video modules using novel efficient algorithms (which we describe in the subsequent sections). To implement our algorithms, we use a set of oblivious primitives which we summarize below.

Oblivious primitives. We use three basic primitives, similar to prior work [RLT15, OSF$^+$16, SGF18]. Fundamental to these primitives is the x86 `CMOV` instruction, which takes as input two registers—a source and a destination—and moves the source to the destination if a condition is true. Once the operands have been loaded into registers, the instructions are immune to memory-access-based pattern leakage because registers are private to the processor, making any register-to-register operations oblivious by default.

1) Oblivious assignment (`oassign`). The `oassign` primitive is a wrapper around the `CMOV` instruction that conditionally assigns a value to the destination operand. This primitive can be used for performing dummy write operations by simply setting the input condition to false. We implement multiple versions of this primitive for different integer sizes. We also implement a vectorized version using SIMD instructions.

2) Oblivious sort (`osort`). The `osort` primitive obliviously sorts an array with the help of a bitonic sorting network [Bat68]. Given an input array of size *n*, the network sorts the array by performing $O(n \log^2(n))$ compare-and-swap operations, which can be implemented using the

`oassign` primitive. As the network layout is fixed given the input size n, execution of each network has identical memory access patterns.

3) Oblivious array access (`oaccess`). The `oaccess` primitive accesses the i-th element in an array, without leaking the value of i. The simplest way of implementing `oaccess` is to scan the entire array. However, as discussed in our threat model (Section 6.3.1), hyperthreading is disabled, preventing any sharing of intra-core resources (e.g., L1 cache) with an adversary, and consequently mitigating known attacks [MWES18, YGH16] that can leak access patterns at sub-cache-line granularity using shared intra-core resources. Therefore, we assume access pattern leakage at the granularity of cache lines, and it suffices for `oaccess` to scan the array at cache-line granularity for obliviousness, instead of per element or byte.

6.5 Designing Oblivious Vision Modules

Naïve approaches and generic tools for oblivious execution of vision modules can lead to prohibitive performance overheads. For instance, a naïve approach for implementing oblivious versions of CPU video analytics modules (as in Figure 6.1) is to simply rewrite them using the oblivious primitives outlined in Section 6.4.4. Such an approach: (i) eliminates all branches and replaces conditional statements with `oassign` operations to prevent control flow leakage via access patterns to code, (ii) implements all array accesses via `oaccess` to prevent leakage via memory accesses to data, and (iii) performs all iterations for a fixed number of times while executing dummy operations when needed. The simplicity of this approach, however, comes at the cost of high overheads: two to three orders of magnitude. Furthermore, as we show in Section 6.8.3, generic tools for executing programs obliviously such as Raccoon [RLT15] and Obfuscuro [AJX$^+$19] also have massive overheads—six to seven orders of magnitude.

Instead, we demonstrate that by carefully crafting oblivious vision modules using the primitives outlined in Section 6.4.4, Visor improves performance over naïve approaches by several orders of magnitude. In the remainder of this section, we present an overview of our design strategy, before diving into the detailed design of our algorithms in Section 6.6 and Section 6.7.

6.5.1 Design Strategy

Our overarching goal is to transform each algorithm into a pattern that processes each pixel identically, regardless of the pixel's value. To apply this design pattern efficiently, we devise a set of algorithmic and systemic optimization strategies. These strategies are informed by the properties of vision modules, as follows.

1) Divide-and-conquer for improving performance. We break down each vision algorithm into independent subroutines based on their functionality and make each subroutine oblivious individually. Intuitively, this strategy improves performance by (i) allowing us to tailor each subroutine separately, and (ii) preventing the overheads of obliviousness from getting compounded.

2) Scan-based sequential processing. Data-oblivious processing of images demands that each pixel in the image be indistinguishable from the others. This requirement presents an opportunity

Component	Input parameters
Video decoding (Section 6.6)	Number of bits used to encode each (padded) row of blocks;
Background subtraction (Section 6.7.1)	–
Bounding box detection (Section 6.7.2)	(*i*) Maximum number of objects per image; (*ii*) Maximum number of different labels that can be assigned to pixels (an object consists of all labels that are adjacent to each other).
Object cropping (Section 6.7.3)	Upper bounds on object dimensions.
Object tracking (Section 6.7.4)	(*i*) An upper bound on the intermediate number of features; (*ii*) An upper bound on the total number of features.
CNN Inference (Section 6.4.3)	–

Table 6.1: Public input parameters in Visor's oblivious modules.

to revisit the design of sequential image processing algorithms. Instead of simply rewriting existing algorithms using the data-oblivious primitives from Section 6.4.4, we find that recasting the algorithm into a form that scans the image, while applying the same functionality to each pixel, yields superior performance. Intuitively, this is because any non-sequential pixel access implicitly requires a scan of the image for obliviousness (*e.g.*, using `oaccess`); therefore, by transforming the algorithm into a scan-based algorithm, we get rid of such non-sequential accesses.

3) Amortize cost across groups of pixels. Processing each pixel in an identical manner lends itself naturally to optimization strategies that enable batched computation over pixels—*e.g.*, the use of data-parallel (SIMD) instructions.

In Visor, we follow the general strategy above to design oblivious versions of popular vision modules that can be composed and reused across diverse pipelines. However, our strategy can potentially help inform the design of other oblivious vision modules as well, beyond the ones we consider.

6.5.2 Input Parameters for Oblivious Algorithms

Our oblivious algorithms rely on a set of public input parameters that need to be provided to Visor before the deployment of the video pipelines. These parameters represent various upper bounds on the properties of the video stream, such as the maximum number of objects per frame, or the maximum size of each object. Table 6.1 summarizes the list of input parameters across all the modules of the vision pipeline.

There are multiple ways by which these parameters may be determined. (*i*) The model owner may obtain these parameters simultaneously while training the model on a public dataset. (*ii*) The client may perform offline empirical analysis of their video streams and choose a reasonable set
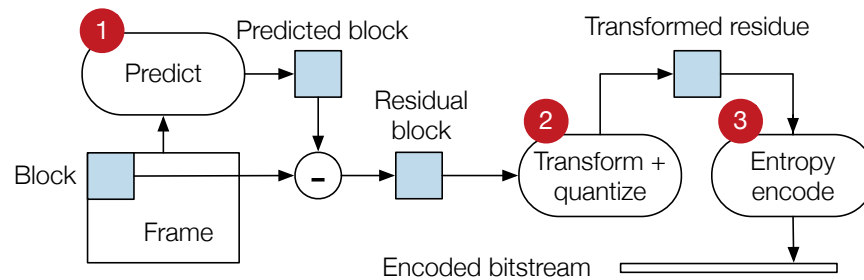
Figure 6.4: Flowchart of the encoding process.

of parameters. (*iii*) Visor may also be augmented to compute these parameters dynamically, based on historical data (though we do not implement this). We note that providing these parameters is not strictly necessary, but meaningful parameters can significantly improve the performance of our algorithms.

6.6 Oblivious Video Decoding

Video encoding converts a sequence of raw images, called *frames*, into a compressed bitstream. Frames are of two types: *keyframes* and *interframes*. Keyframes are encoded to only exploit redundancy across pixels *within the same frame*. Interframes, on the other hand, use the prior frame as reference (or the most recent keyframe), and thus can exploit temporal redundancy in pixels *across frames*.

Encoding overview. We ground our discussion using the VP8 encoder [BWX11], but our techniques are broadly applicable. A frame is decomposed into square arrays of pixels called *blocks*, and then compressed using the following steps (see Figure 6.4). ❶ An estimate of the block is first *predicted* using reference pixels (in a previous frame if interframe or the current frame if keyframe). The prediction is then subtracted from the actual block to obtain a *residue*. ❷ Each block in the residue is *transformed* into the frequency domain (*e.g.*, using a discrete cosine transform), and its coefficients are *quantized* thus improving compression. At the end of this step, each block comprises a sequence of 16 data values, the last several of which are typically zeros as the quantization factors for the later coefficients are larger than those of the initial ones. ❸ Each (quantized) block is compressed into a variable-sized bitstream using a binary prefix tree and arithmetic encoding. The last few coefficients that are zeros are not encoded, and an end-of-block symbol (EOB) is encoded instead. Block prediction modes, cosine transformation, and arithmetic encoding are core to all video encoders (*e.g.*, H264 [H26], VP9 [VP9]) and thus our oblivious techniques carry over to all popular codecs.

The *decoder* reverses the steps of the encoder: (*i*) the incoming video bitstream is entropy decoded (Section 6.6.2); (*ii*) the resulting coefficients are dequantized and inverse transformed to obtain the residual block (Section 6.6.3); and (*iii*) previously decoded pixels are used as reference to obtain a prediction block, which are then added to the residue (Section 6.6.4).

6.6.1 Video Encoder Padding

While the video stream is in transit, the bitrate variation of each frame is visible to an attacker observing the network even if the traffic is TLS-encrypted. This variability can be exploited for fingerprinting video streams [SST17] and understanding its content. Overcoming this leakage requires changes to the video *encoder* to "pad" each frame with dummy bits to an upper bound before sending the stream to Visor.

We modify the video encoder to pad the encoded video streams. However, instead of applying padding at the level of frames, we pad each individual *row of blocks* within the frames. Compared to frame-level padding, padding individual rows of blocks significantly improves latency of oblivious decoding, but at the cost of an increase in network bandwidth.

Padding the frames of the video stream, however, negates the benefit of using *interframes* during encoding of the raw video stream, which are typically much smaller than keyframes. We therefore configure the encoder to encode all raw video frames into keyframes, which eliminates the added complexity of dealing with interframes, and consequently simplifies the oblivious decoding procedure.

We note that it may not always be possible to modify legacy cameras to incorporate padding. In such cases, potential solutions include the deployment of a lightweight edge-compute device that pads input camera feeds before streaming them to the cloud. For completeness, we also discuss the impact of the lack of padding in Appendix D, along with the accompanying security-performance tradeoff.

6.6.2 Bitstream Decoding

The bitstream decoder reconstructs blocks with the help of a *prefix tree*. At each node in the tree it decodes a single bit from the compressed bitstream via arithmetic decoding, and traverses the tree based on the value of the bit. While decoding the bit, the decoder first checks whether any more bits can be decoded at the current bitstream position, and if not, it advances the bitstream pointer by two bytes. Once it reaches a leaf node, it outputs a coefficient based on the position of the leaf, and assigns the coefficient to the current pixel in the block. If an EOB symbol is decoded, then all the coefficients remaining in the block are assigned a value of zero. This continues for all the coefficients in the frame.

Requirements for obliviousness. The above algorithm leaks information about the compressed bitstream. First, the traversal of the tree leaks the *value of the parsed coefficient*. For obliviousness, we need to ensure that during traversal, the identity of the current node being processed remains secret. Second, not every position in the bitstream encodes the same number of coefficients, and the bitstream pointer advances variably during decoding. Hence, this leaks the *number of coefficients* that are encoded per two-byte chunk (which may convey their values). Finally, the presence of EOB coefficients, coupled with the assignment of decoded coefficients to pixels, leaks the number of zero coefficients per block of the frame—prior work has demonstrated attacks that exploit similar leakage to infer the outlines of all objects in the frame [XCP15]. We design a solution that *decouples*

the parsing of coefficients, *i.e.*, prefix tree traversal (Section 6.6.2.1), from the assignment of the parsed coefficients to pixels (Section 6.6.2.2).

6.6.2.1 Oblivious prefix tree traversal

A simple way to make tree traversal oblivious is to represent the prefix tree as an array. We can then obliviously fetch any node in the tree using `oaccess` (Section 6.4.4). Though this hides the identity of the fetched node, we need to also ensure that *processing* of the nodes does not leak their identity.

In particular, we need to ensure that nodes are indistinguishable from each other by performing an identical set of operations at each node. Unfortunately, this requirement is complicated by the following facts. (1) Only leaf nodes in the tree produce outputs (*i.e.*, the parsed coefficients) and not the intermediate nodes. (2) We do not know beforehand which nodes in the tree will cause the bitstream pointer to be advanced; at the same time, we need to ensure that the pointer is advanced predictably and independent of the bitstream. To solve these problems, we take the following steps.

1. We modify each node to output a coefficient regardless of whether it is a leaf state or not. Leaves output the parsed coefficient, while other states output a dummy value.

2. We introduce a dummy node into the prefix tree. While traversing the tree, if no more bits can be decoded at the current bitstream position, we transition to the dummy node and perform a bounded number of dummy decodes.

These modifications ensure that while traversing the prefix tree, all that an attacker sees is that at *some* node in the tree, a single bit was decoded and a single value was outputted.

Note that in this phase, we do not assign coefficients to pixels, and instead collect them in a list. If we were to assign coefficients to pixels in this phase, then the decoder would need to obliviously scan the entire frame (using `oaccess`) at every node in the tree, in order to hide the pixel's identity. Instead, by *decoupling* parsing from assignment, we are able to perform the assignment obliviously using a super-linear number of accesses (instead of quadratic), as we explain next.

6.6.2.2 Oblivious coefficient assignment

At the end of Section 6.6.2.1, we have a list of actual and dummy coefficients. The key idea is that if we can obliviously sort this set of values using `osort` such that all the actual coefficients are contiguously ordered while all dummies are pushed to the front, then we can simply read the coefficients off the end of the list sequentially and assign them to pixels one by one.

However, recall that in lieu of the trailing zeros within each block, the encoder encodes an EOB symbol instead. Therefore, we need to append the requisite zeros to the set and move them to the appropriate indices before we can carry out the assignment. To achieve this, our algorithm makes a single forward pass over the set to add the zeros, while updating all index values per tuple in a way that ensures the zeros will be sorted to the correct positions, as illustrated in Figure 6.5.

To enable such a sort, we modify the prefix tree traversal to additionally output a tuple (`flag`, `index`) per coefficient; `flag` is 0 for dummies and 1 otherwise; `index` is an increasing
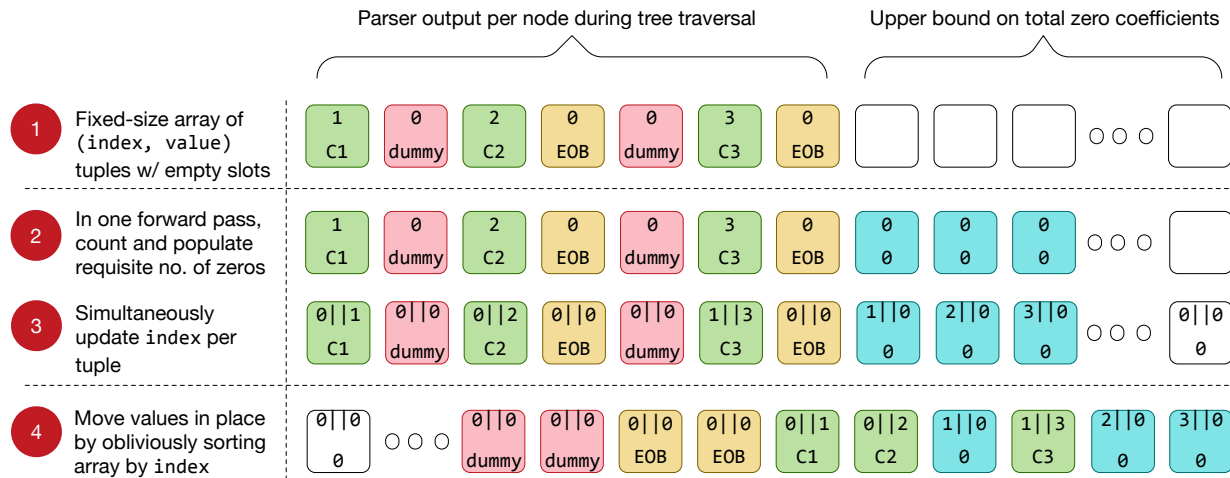
Figure 6.5: Steps for obliviously sorting the coefficients into place after populating it with zero coefficients. For simplicity, this illustration assumes that there are two subblocks, with three coefficients per subblock.

counter as per the pixel's index. Then, the desired sort can be achieved by sorting the list based on the value of the tuple.

As the complexity of oblivious sort is super-linear in the number of elements being sorted, an important optimization is to decode and assign coefficients to pixels at the granularity of *rows of blocks* rather than frames. While the number of bits per row of blocks may be observed, the algorithm's obliviousness is not affected as each row of blocks in the video stream is padded to an upper bound (Section 6.6.1); had we applied frame-level padding, this optimization would have revealed the number of bits per row of blocks. In Section 6.8.1.1, we show that this technique improves oblivious decoding latency by $\sim 6\times$.

6.6.3 Dequantization and Inverse Transformation

The next step in the decoding process is to (*i*) dequantize the coefficients decoded from the bitstream, followed by (*ii*) inverse transformation to obtain the residual blocks. Dequantization just multiplies each coefficient by a quantization factor. The inverse transformation also performs a set of identical arithmetic operations irrespective of the coefficient values.

6.6.4 Block Prediction

Prediction is the final stage in decoding. The residual block obtained after Section 6.6.3 is added to a *predicted block*, obtained using a previously constructed block as reference, to obtain the raw pixel values. In keyframes, each block is *intra*-predicted—*i.e.*, it uses a block in the same frame as referenced. In the presence of video encoder padding, the padded input video stream only contains keyframes as described in Section 6.6.1.

Intra-predicted blocks are computed using one of several *modes*. A mode to encode a block refers to a combination of pixels on its top row and left column used as reference. Obliviousness requires that the prediction mode remains private. Otherwise, an attacker can identify the pixels that are most similar to each other, thus revealing details about the frame.

We investigate two different ideas for making intra-prediction oblivious. First, we note that in each prediction mode, the value of a pixel in the predicted block can be expressed as a linear combination $\Sigma a_i p_i$ of all the pixels that lie above and to the left of the block. Here p_i represents the adjoining pixels and a_i are weights. Thus, to compute the value of the predicted pixel obliviously, we can simply evaluate the expression after using the `oassign` primitive to obliviously assign each a_i a value based on the mode and the location of the current pixel.

A second approach is to simply evaluate all possible predictions for the pixel and store them in an array, indexing each prediction by its mode. Then, use the `oaccess` primitive to obliviously select the correct prediction from the array.

We implemented both approaches, and found that the second offers better performance in practice. This is because in the second approach, we can compute the predicted values for several pixels simultaneously at the level of individual rows, which amortizes the cost of our operations.

6.7 Oblivious Image Processing

After obliviously decoding frames in Section 6.6, the next step as shown in Figure 6.1 is to develop data-oblivious techniques for background subtraction (Section 6.7.1), bounding box detection (Section 6.7.2), object cropping (Section 6.7.3), and tracking (Section 6.7.4). We present the key ideas here; detailed pseudocode and proofs of obliviousness are available in Appendix C.2. Note that Section 6.7.1 and Section 6.7.4 modify popular algorithms to make them oblivious, while Section 6.7.2 and Section 6.7.3 propose new oblivious algorithms.

6.7.1 Background Subtraction

The goal of background subtraction is to detect moving objects in a video. Specifically, it dynamically learns stationary pixels that belong to the video's background, and then subtracts them from each frame, thus producing a binary image with black background pixels and white foreground pixels.

Zivkovic *et al.* proposed a mechanism [Ziv04, ZvdH06] that is widely used in practical deployments, that models each pixel as a mixture of Gaussians [BBV08]. The number of Gaussian components M differs across pixels depending on their value (but is no more than M_{max}, a predefined constant). As more data arrives (with new frames), the algorithm updates each Gaussian component along with their weights (π), and adds new components if necessary.

To determine if a pixel $\vec{x}$ belongs to the background or not, the algorithm uses the B Gaussian components with the largest weights and outputs true if $p(\vec{x})$ is larger than a threshold:

$$p(\vec{x}) = \sum_{m=1}^{B} \pi_m \mathcal{N}(\vec{x} \mid \vec{\mu}_m, \Sigma_m)$$
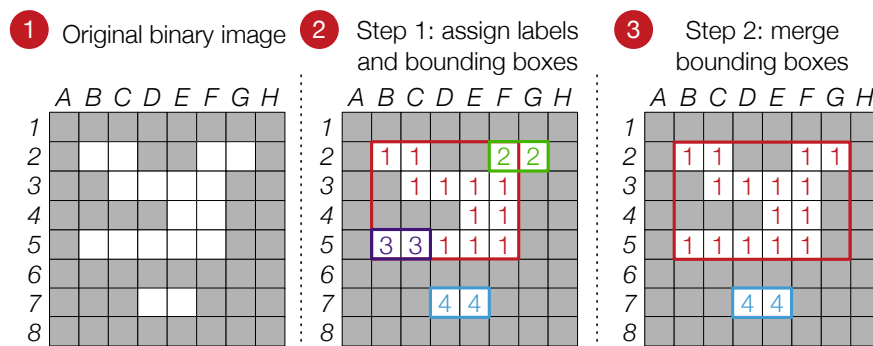
(a) CCL-based algorithm for bounding box detection
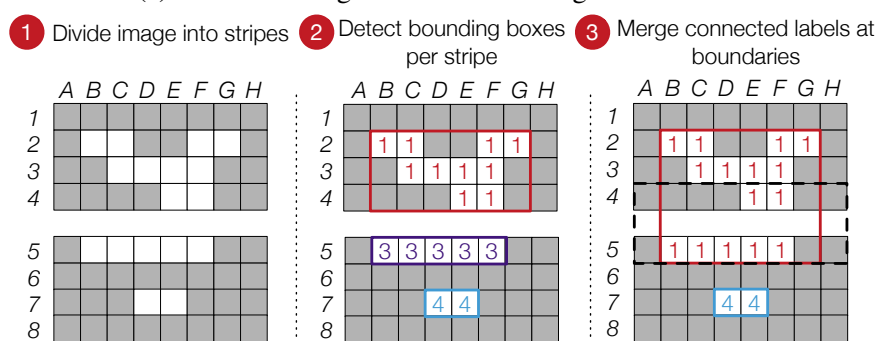
(b) Enhancement via parallelization

Figure 6.6: Oblivious bounding box detection

where $\vec{\mu}_m$ and Σ_m are parameters of the Gaussian components, and π_m is the weight of the m-th Gaussian component.

This algorithm is not oblivious because it maintains a different number of Gaussian components per pixel, and thus performs different steps while updating the mixture model per pixel. These differences are visible via access patterns, and these leakages reveal to an attacker how *complex* a pixel is in relation to others—*i.e.*, whether a pixel's value stays stable over time or changes frequently. This enables the attacker to identify the positions of moving objects in the video.

For obliviousness, we need to perform an identical set of operations per pixel (regardless of their value); we thus *always* maintain $M_{\max}$ Gaussian components for each pixel, of which $(M_{\max} - M)$ are dummy components and assigned a weight $\pi = 0$. When newer frames arrive, we use oassign operations to make all the updates to the mixture model, making dummy operations for the dummy components. Similarly, to select the B largest components by weight, we use the osort primitive.

6.7.2   Bounding Box Detection

The output from Section 6.7.1 is a binary image with black background pixels where the foreground objects are white blobs (Figure 6.6a). To find these objects, it suffices to find the *edge contours* of all blobs. These are used to compute the *bounding rectangular box* of each object. A standard

approach for finding the contours in a binary image is the border following algorithm of Suzuki and Abe [SA85]. As the name suggests, the algorithm works by scanning the image until it locates an edge pixel, and then follows the edge around a blob. As Figure 6.2 in Section 6.2.3 illustrated, the memory access patterns of this algorithm leak the details of all the objects in the frame.

A naïve way to make this algorithm oblivious is to implement each pixel access using the `oaccess` primitive (along with other minor modifications). However, we measure that this approach slows down the algorithm by over $\sim 1200\times$.

We devise a two-pass oblivious algorithm for computing bounding boxes by adapting the classical technique of connected component labeling (CCL) [RP66]. The algorithm's main steps are illustrated in Figure 6.6a (whose original binary image contains two blobs). In the first pass, it scans the image and assigns each pixel a temporary label if it is "connected" to other pixels. In the second pass, it merges labels that are part of a single object. Even though CCL on its own is less efficient for detecting blobs than border following, it is far more amenable to being adapted for obliviousness.

We make this algorithm oblivious as follows. First, we perform identical operations regardless of whether the current pixel is connected to other pixels. Second, for efficiency, we restrict the maximum number of temporary labels (in the first pass) to a parameter N provided as input to Visor (per Section 6.5.2, Table 6.1). Note that the value of the parameter may be much lower than the worst case upper bound (which is the total number of pixels), and thus is more efficient.

Enhancement via parallelization. We observe that the oblivious algorithm can be parallelized using a divide-and-conquer approach. We divide the frame into horizontal *stripes* (**1** in Figure 6.6b) and process *each stripe in parallel* (**2**). For objects that span stripe boundaries, each stripe outputs only a *partial* bounding box containing the pixels within the stripe. We combine the partial boxes by re-applying the oblivious CCL algorithm to the boundaries of adjacent stripes (**3**). Given two adjacent stripes S_i and S_{i+1} one below the other, we compare each pixel in the top row of S_{i+1} with its neighbors in the bottom row of S_i, and merge their labels as required.

6.7.3 Object Cropping

The next step after detecting bounding boxes of objects is to *crop* them out of the frame to be sent for CNN classification (Figure 6.1a). Visor needs to ensure that the cropping of objects does not leak (*i*) their positions, or (*ii*) their dimensions.

6.7.3.1 Hiding object positions

A naïve way of obliviously cropping an object of size $p \times q$ is to slide a window (of size $p \times q$) horizontally in raster order, and copy the window's pixels if it aligns with the object's bounding box. Otherwise, perform a dummy copy. This, however, leads to a slow down of $4000\times$, with the major reason being redundant copies: while sliding the window forward by one pixel results in a new position in the frame, a majority of the pixels copied are the same as in the previous position. For a $m \times n$ frame, and an object of size $p \times q$, the technique results in $pq(m-p)(n-q)$ pixel copies, as compared to pq pixel copies when directly cropping the object.

(a) Localizing objects.

(b) Bilinear interpolation.

(c) Improved Bilinear interpolation.
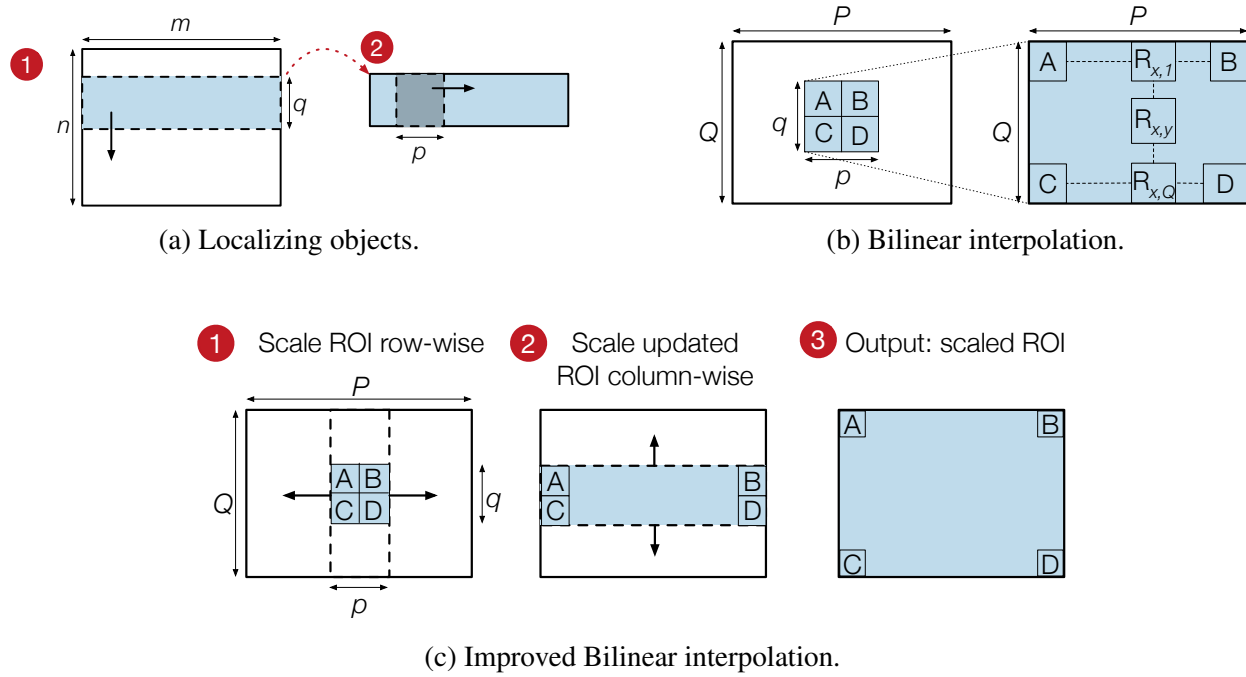
Figure 6.7: Oblivious object cropping

We get rid of this redundancy by *decoupling* the algorithm into multiple passes—one pass along each dimension of the image—such that each pass performs only a subset of the work. As Figure 6.7a shows, the first phase extracts the horizontal strip containing the object; the second phase extracts the object from the horizontal strip.

① Instead of sliding a window (of size $p \times q$) across the frame (of size $m \times n$), we use a horizontal strip of $m \times q$ that has width m equal to that of the frame, and height q equal to that of the object. We slide the strip vertically down the frame *row by row*. If the top and bottom edges of the strip are aligned with the object, we copy all pixels covered by the strip into the buffer; otherwise, we perform dummy copies. This phase results in $mq(n-q)$ pixel copies.

② We allocate a window of size $p \times q$ equal to the object's size and then slide it *column by column* across the extracted strip in **①**. If the left and right edges of the window are aligned with the object's bounding box, we copy the window's pixels into the buffer; if not, we perform dummy copies. This phase performs $pq(m-p)$ pixel copies.

Algorithm 4 in Appendix C.2 provides the detailed steps.

6.7.3.2 Hiding object dimensions

The algorithm in Section 6.7.3.1 leaks the dimensions $p \times q$ of the objects. To hide object dimensions, Visor takes as input parameters P and Q representing upper bounds on object dimensions (as described in Section 6.5.2, Table 6.1), and instead of cropping out the exact $p \times q$ object, we obliviously crop out a larger image of size $P \times Q$ that *subsumes* the object. While the object sizes

vary depending on their position in the frame (e.g., near or far from the camera), the maximum values (P and Q) can be learned from profiling just a few sample minutes of the video, and they tend to remain unchanged in our datasets.

This larger image now contains extraneous pixels surrounding the object, which might lead to errors during the CNN's object classification. We remove the extraneous pixels surrounding the $p \times q$ object by obliviously scaling it up to fill the $P \times Q$ buffer. Note that all objects we send to the CNN across the CPU-GPU channel are of size $P \times Q$ (Section 6.4.2), and recall from Section 6.4.1 that we extract the same number of objects from each frame (by padding dummy objects, if needed).

We develop an oblivious routine for scaling up using bilinear interpolation [Jai89]. Bilinear interpolation computes the value of a pixel in the scaled up image using a linear combination of a 2×2 array of pixels from the original image (see Figure 6.7b). The simplest way to implement this routine obliviously is to fetch the 4 pixel values obliviously using `oaccess` for each pixel in the scaled up image. This would entail PQ scans of the entire image, yielding a total of $O(P^2Q^2)$ pixel accesses. We once again use decoupling of the algorithm into two passes to improve its efficiency (Figure 6.7c) by scaling up along a single dimension per pass. The two passes perform a total of $O(P^2Q + PQ^2)$ pixel accesses, improving asymptotic performance over the $O(P^2Q^2)$ algorithm.

Cache locality. Since the second pass of our (decoupled bilinear interpolation) algorithm performs column-wise interpolations, each pixel access during the interpolation touches a different cache line. To exploit cache locality, we *transpose* the image before the second pass, and make the second pass to also perform *row-wise* interpolations (as in the first pass). This results in another order of magnitude speedup (Section 6.8.1.4).

6.7.4   Object Tracking

Object tracking consists of two main steps: feature detection in *each frame* and feature matching *across frames*.

Feature detection.   SIFT [Low99, Low04] is a popular algorithm for extracting features for *keypoints*, i.e., pixels that are the most "valuable" in the frame. In a nutshell, it generates candidate keypoints, where each candidate is a local maxima/minima; the candidates are then filtered to get the legitimate keypoints.

Based on the access patterns of the SIFT algorithm, an attacker can infer the locations of all the keypoints in the image, which in turn, can reveal the location of all object "corners" in the image. A naïve way of making the algorithm oblivious is to treat each pixel as a keypoint, performing all the above operations for each. However, the SIFT algorithm's performance depends critically on its ability to filter out a small set of good keypoints from the frame.

To be oblivious *and* efficient, Visor takes as input two parameters N_{temp} and N (per Table 6.1). The parameter N_{temp} represents an upper bound on the number of candidate keypoints, and N on the number of legitimate keypoints. These parameters, coupled with `oassign` and `osort`, allow for efficient and oblivious identification of keypoints. Finally, computing the *feature descriptors* for each keypoint requires accessing the pixels around it. For this, we use oblivious extraction (Section 6.7.3). Appendix C.2's Algorithm 6 has the pseudocode.

Feature matching. The next step after detecting features is to match them across images. Feature matching computes a distance metric between two sets of features, and identifies features that are "nearest" to each other in the two sets. In Visor, we simply perform brute-force matching of the two sets, using `oassign` operations to select the closest features.

6.8 Evaluation

Implementation. We implement our oblivious video decoder atop FFmpeg's VP8 decoder [FFm] and oblivious vision algorithms atop OpenCV 3.2.0 [Ope]. We use Caffe [JSD+14] for running CNNs. We encrypt data channels using AES-GCM. We implement the oblivious primitives of Section 6.4.4 using inline assembly code (as in [OSF+16, RLT15, SGF18]), and manually verified the binary to ensure that compiler optimizations do not undo our intent; one can also use tools such as Vale [BHK+17] to do the same.

Testbed. We evaluate Visor on Intel i7-8700K with 6 cores running at 3.7 GHz, and an NVIDIA GTX 780 GPU with 2304 CUDA cores running at 863 MHz. We disable hyperthreading for experiments with Visor (per Section 6.3), but retain hyperthreading in the insecure baseline. Disabling hyperthreading for security does not sacrifice the performance of Visor (due to its heavy utilization of vector units) unlike the baseline system that favors hyperthreading.[1] The server runs Linux v4.11; supports AVX2 and SGX-v1 instruction sets; and has 32 GB of memory, with 93.5 MB of enclave memory. The GPU has 3 GB of memory.

Datasets. We use four real-world video streams (obtained with permission) in our experiments: streams 1 and 4 are from traffic cameras in the city of Bellevue (resolution 1280×720) while streams 2 and 3 are sourced from cameras surveilling commercial datacenters (resolution 1024×768). All these videos are privacy-sensitive as they involve government regulations or business sensitivity. For experiments that evaluate the cost of obliviousness across different resolutions and bitrates, we re-encode the videos accordingly. A recent body of work [KEA+17, JAB+18, ZAB+17] has found that the accuracy of object detection in video streams is not affected if the resolution is decreased (while consuming significantly lesser resources), and 720p videos suffice. We therefore chose to use streams closer to 720p in resolution because we believe they would be a more accurate representation of real performance.

Evaluation highlights. We summarize the key takeaways of our evaluation.

1. Visor's optimized oblivious algorithms (Section 6.6, Section 6.7) are up to $1000\times$ faster than naïve competing solutions. (Section 6.8.1)

---

[1]We measured the impact of disabling hyperthreading on Visor's performance to be 5%. Visor heavily utilizes vector units due to the increased data-level parallelism of oblivious algorithms, leaving little space for performance improvement when hyperthreading is enabled [KOV18]. As such, the increased security comes with negligible performance overhead. We also note that disabling hyperthreading in cloud VMs is considered to be good practice due to the reduced impact of microarchitectural data-sampling vulnerabilities that affect commodity Intel CPUs (not just Intel SGX) [vSMO+19, CGG+19, SLM+19, vSMK+20]. Our experiments demonstrate that disabling hyperthreading in the baseline system reduces its performance by 30%; which bridges considerably the performance gap between Visor and insecure baseline systems in hyperthreading-disabled cloud deployments.
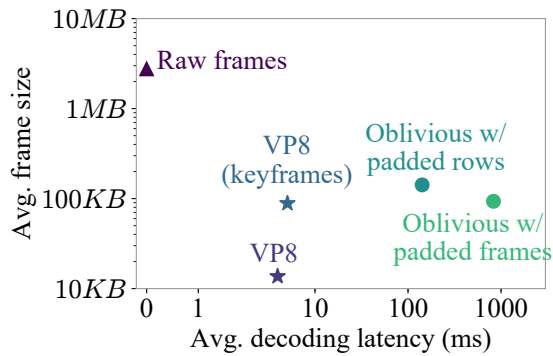
Figure 6.8: Decoding latency vs. B/W.
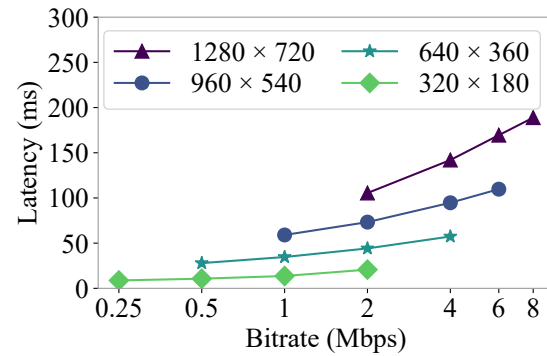
Figure 6.9: Latency of oblivious decoding.
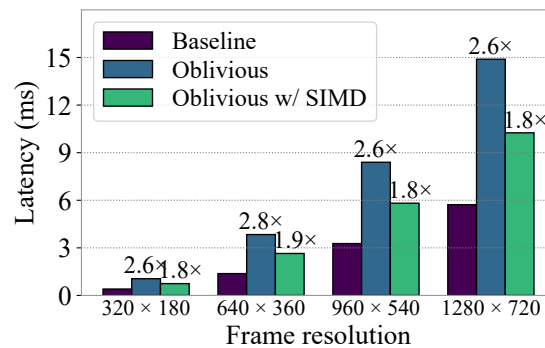
Figure 6.10: Background subtraction.

2. End-to-end overhead of obliviousness for real-world video pipelines with state-of-the-art CNNs are limited to $2\times$–$6\times$ over a *non-oblivious* baseline. (Section 6.8.2)

3. Visor is generic and can accommodate multiple pipelines (Section 6.2.1; Figure 6.1) that combine the different vision processing algorithms and CNNs. (Section 6.8.2)

4. Visor's performance is over 6 to 7 orders of magnitude better than a state-of-the-art general-purpose system for oblivious program execution. (Section 6.8.3)

Overall, Visor's use of properties of the video streams has *no impact on the accuracy* of the analytics outputs.

6.8.1 Performance of Oblivious Components

We begin by studying the performance of Visor's oblivious modules: we quantify the raw overhead of our algorithms (without enclaves) over non-oblivious baselines; we also measure the improvements over naïve oblivious solutions.

6.8.1.1 Oblivious video decoding

Decoding of the compressed bitstream dominates decoding latency, consuming up to ∼90% of the total latency. Further, this stage is dominated by the oblivious assignment subroutine which sorts coefficients into the correct pixel positions using osort, consuming up to ∼83% of the decoding latency. Since the complexity of oblivious sort is super-linear in the number of elements being sorted, our technique for decoding at the granularity of *rows of blocks* rather than frames significantly improves the latency of oblivious decoding.

Overheads. Figure 6.8 shows the bandwidth usage and decoding latency for different oblivious decoding strategies (*i.e.*, decoding at the level of frames, or at the level of *row of blocks*) for a video stream of resolution 1280×720. We also include two reference points: non-encoded frames and VP8 encoding. The baseline latency of decoding VP8 encoded frames is 4–5 ms. Non-encoded raw frames incur no decoding latency but result in frames that are three orders of magnitude larger than the VP8 average frame size (10s of kB) at a bitrate of 4 Mb/s.

Frame-level oblivious decoding introduces high latency (∼850 ms), which is two orders of magnitude higher than non-oblivious counterparts. Furthermore, padding each frame to prevent leakage of the frame's bitrate increases the average frame size to ∼95 kB. On the contrary, oblivious decoding at the level of rows of blocks delivers ∼140 ms, which is ∼6× lower than frame-level decoding. However, this comes with a modest increase in network bandwidth as the encoder needs to pad each row of blocks individually, rather than a frame. In particular, the frame size increases from ∼95 kB to ∼140 kB.

Apart from the granularity of decoding, the latency of the oblivious sort is also governed by: (*i*) the frame's resolution, and (*ii*) the bitrate. The higher the frame's resolution / bitrate, the more coefficients there are to be sorted. Figure 6.9 plots oblivious decoding latency at the granularity of rows of blocks across video streams with different resolutions and bitrates. The figure shows that lower resolution/bitrates introduce lower decoding overheads. In many cases, lower image qualities are adequate for video analytics as it does not impact the accuracy of the object classification [JAB⁺18].

6.8.1.2 Background subtraction

We set the maximum number of Gaussian components per pixel $M_{\mathsf{max}} = 4$, following prior work [Ziv04, ZvdH06]. Our changes for obliviousness enable us to make use of SIMD instructions for updating the Gaussian components in parallel. This is because we now maintain the same number of components per pixel, and update operations for each component are identical.

Figure 6.10 plots the overhead of obliviousness on background subtraction across different resolutions. The SIMD implementation increases the latency of the routine only by $1.8\times$ over the baseline non-oblivious routine. As the routine processes each pixel in the frame independent of the rest, its latency increases linearly with the total number of pixels.
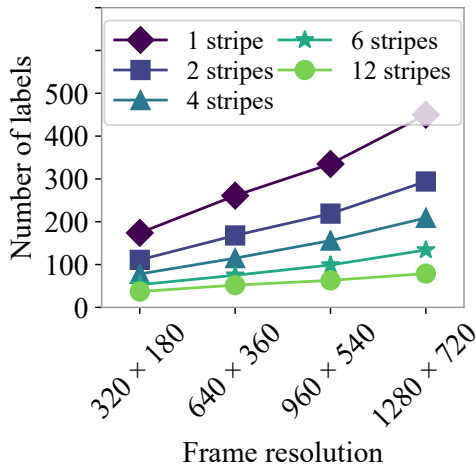
Figure 6.11: Number of labels for bounding box detection.
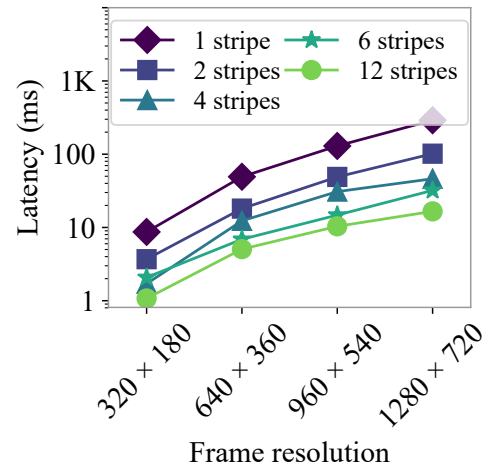
Figure 6.12: Latency of oblivious bounding box detection.

6.8.1.3 Bounding box detection

For non-oblivious bounding box detection, we use the border-following algorithm of Suzuki and Abe [SA85] (per Section 6.7.2); this algorithm is efficient, running in sub-millisecond latencies.

The performance of our oblivious bounding box detection algorithm is governed by two parameters: (*i*) the number of stripes used in the divide-and-conquer approach, which controls the degree of parallelism, and (*ii*) an upper bound L on the maximum number of labels possible per stripe, which determines the size of the algorithm's data structures.

Figure 6.11 plots L for streams of different frame resolutions while varying the number of stripes into which each frame is divided. As expected, as the number of stripes increases, the value of L required per stripe decreases. Similarly, lower resolution frames require smaller values of L.

Figure 6.12 plots the latency of detecting all bounding boxes in a frame based on the value of the parameter L, ranging from a few milliseconds to hundreds of milliseconds. For a given resolution, the latency decreases as the number of stripes increase, due to two reasons: (*i*) increased parallelism, and (*ii*) smaller sizes of L required per stripe. Overall, the divide-and-conquer approach reduces latency by an order of magnitude down to a handful of milliseconds.

6.8.1.4 Object cropping

We first evaluate oblivious object cropping while leaking object sizes. We include three variants: the naïve approach; the two-phase approach; and a further optimization that advances the sliding window forward multiple rows/columns at a time. Figure 6.13 plots the cost of cropping variable-sized objects from a 1280×720 frame, showing that the proposed refinements reduce latency by three orders of magnitude .

Figure 6.14 plots the latency of obliviously resizing the target ROI within a cropped image to
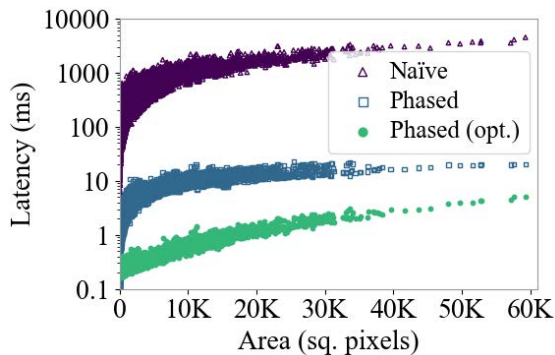
Figure 6.13: Oblivious object cropping.
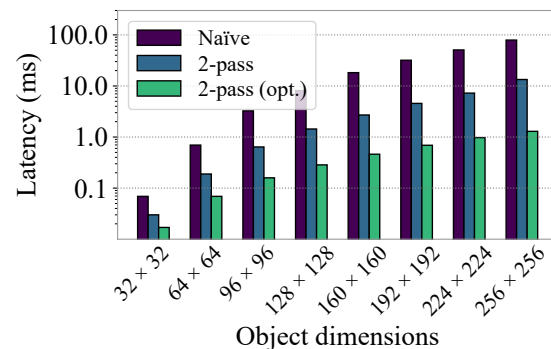
Figure 6.14: Oblivious object resizing.
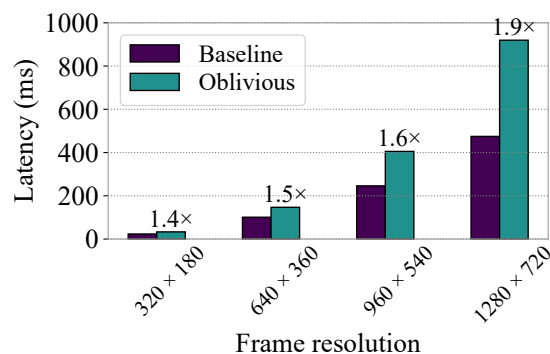
Figure 6.15: Oblivious object tracking.

hide the object's size. While the latency of naïve bilinear interpolation is high (10s of milliseconds) for large objects, the optimized two-pass approach (that exploits cache locality by transposing the image before the second pass; Section 6.7.3.2) reduces latency by two orders of magnitude down to one millisecond for large objects.

6.8.1.5 Object tracking

Figure 6.15 plots the latency of object tracking with and without obliviousness. We examine our sample streams at various resolutions to determine upper bounds on the maximum number of features in frames. As the resolution increases, the overhead of obliviousness increases as well because our algorithm involves an oblivious sort of the intermediate set of detected features, the cost of which is superlinear in the size of the set. Overall, the overhead is $< 2\times$.

6.8.1.6 CNN classification on GPU

Buffer. Figure 6.16 benchmarks the sorting cost as a function of the object size and the buffer size. For buffer sizes smaller than 50, the sorting cost remains under 5 ms.
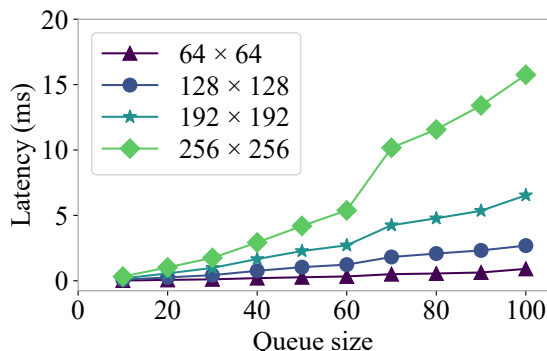
Figure 6.16: Oblivious queue sort.

CNN	Batches/s	Max no. of models
AlexNet	40.3	7
ResNet-18	18.4	4
ResNet-50	8.2	1
VGG-16	5.4	1
VGG-19	4.4	1
Yolo	3.9	1

Table 6.2: CNN throughput (batch size 10).

Inference.   We measure the performance of CNN object classification on the GPU. As discussed in Section 6.4.3, oblivious inference comes free of cost. Table 6.2 lists the throughput of different CNN models using the proprietary NVIDIA driver, with CUDA version 9.2. Each model takes as input a batch of 10 objects of size 224×224. Further, since GPU memory is limited to 3 GB, we also list the maximum number of concurrent models that can run on our testbed. As we show in Section 6.8.2, the latter has a direct bearing on the number of video analytics pipelines that can be concurrently served.

6.8.2   System Performance

We now evaluate the end-to-end performance of the video analytics pipeline using four real video streams. We present the overheads of running Visor's data-oblivious techniques and hosting the pipeline in a hybrid enclave. We evaluate the two example pipelines in Figure 6.1: pipeline 1 uses an object classifier CNN; pipeline 2 uses an object detector CNN (Yolo), and performs object tracking on the CPU.

Pipeline 1 configuration. We run inference on objects that are larger than 1% of the frame size as smaller detected objects do not represent any meaningful value. Across our videos, the number of such objects per frame is small—no frame has more than 5 objects, and 97-99% of frames have less than 2 to 3 objects. Therefore, we configure: (*i*) Visor's object detection stage to conservatively
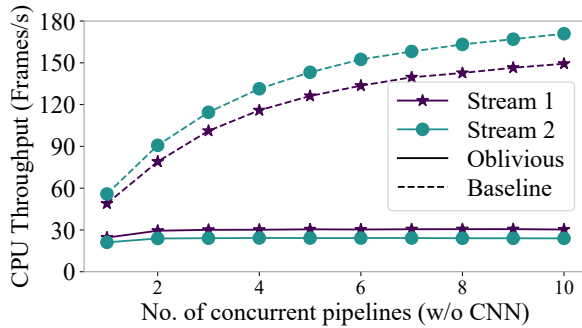
Figure 6.17: CPU throughput (pipeline 1).
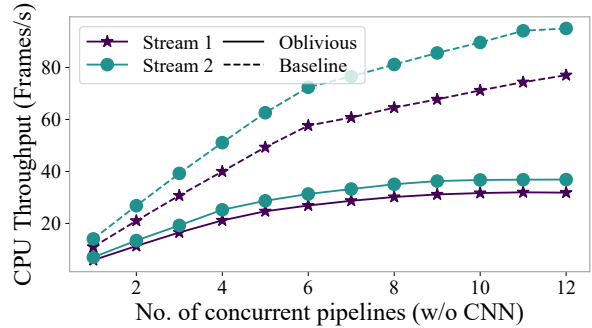
Figure 6.18: CPU throughput (pipeline 2).

output 5 objects per frame (including dummies) into the buffer, (*ii*) the consumption rate of Visor's CNN module to 2 or 3 objects per frame (depending on the stream), and (*iii*) the buffer size to 50, which suffices to prevent non-dummy objects from being overwritten.

Pipeline 2 configuration. The Yolo object detection CNN ingests entire frames, instead of individual objects. In the baseline, we filter frames that don't contain any objects using background subtraction. However, we forego this filtering in the oblivious version since most frames contain foreground objects in our sample streams. Additionally, Yolo expects the frames to be of resolution 448×448. So we resize the input video streams to be of the same resolution.

Cost of obliviousness. Figures 6.17 and 6.18 plot the overhead of Visor on the CPU-side components of pipelines 1 and 2, while varying the number of concurrent pipelines. Visor reduces *peak* CPU throughput by $\sim 2.6\times$–$6\times$ across the two pipelines, compared to the non-oblivious baseline. However, the throughput of the system ultimately depends on the number of models that can fit in GPU memory.

Figure 6.19 plots Visor's end-to-end performance for both pipelines, across all four sample video streams. In the presence of CNN inference, Visor's overheads depend on the model complexity. Pipelines that utilize light models, such as AlexNet and ResNet-18, are bottlenecked by the CPU. In such cases, the overhead is determined by the cost of obliviousness incurred by the CPU components. With heavier models such as ResNet-50 and VGG, the performance bottleneck shifts to the GPU. In this case, the overhead of Visor is governed by the amount of dummy objects processed by the GPU (as described in Section 6.4.2). Overall, the cost of obliviousness remains in the range of $2.2\times$–$5.9\times$ across video streams for the first pipeline. In the second pipeline, the overhead is $\sim 2\times$. The GPU can fit only a single Yolo model. The overall performance, however, is bottlenecked at the CPU because the object tracking routine is relatively expensive.

Cost of enclaves. We measure the cost of running the pipelines in CPU/GPU enclaves by replacing the NVIDIA stack with Graviton's stack, which comprises open-source CUDA runtime (Gdev [KMMB12]) and GPU driver (Nouveau [Nou]).

Figure 6.20 compares Visor against a non-oblivious baseline when both systems are hosted in CPU/GPU enclaves. As SGX's EPC size is limited to 93.5 MB, workloads with large memory footprints incur high overhead. For pipeline 1, and for large frame resolutions, the latency of
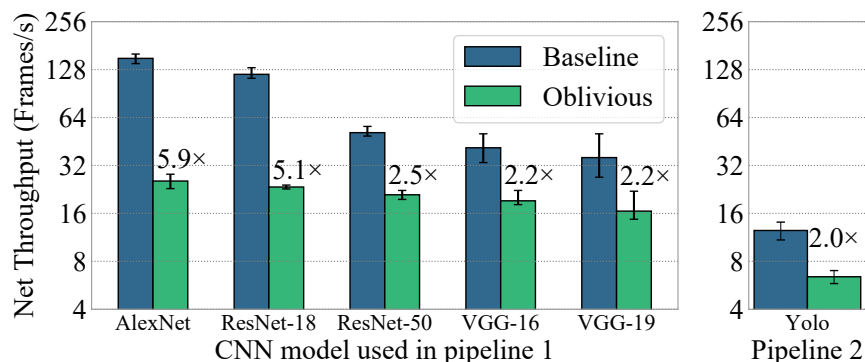
Figure 6.19: Overall pipeline throughput.
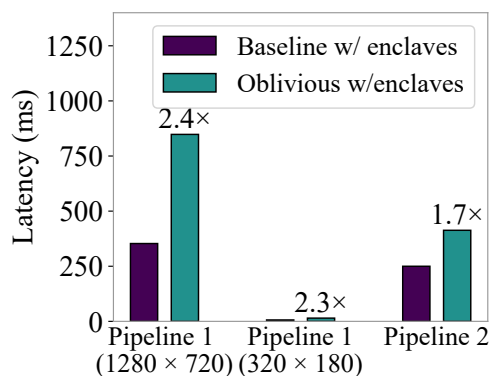
Figure 6.20: Cost of enclaves.

background subtraction increases from ∼6 ms to 225 ms due to its working set size being 132 MB. In Visor, the pipeline's net latency increases by 2.4× (as SGX overheads mask some of Visor's overheads) while increasing the memory footprint to 190 MB. When the pipeline operates on lower frame resolutions, such that its memory footprint fits within current EPC, the latency of the non-oblivious baseline tracks the latency of the insecure baseline (a few milliseconds); the additional overhead of obliviousness is 2.3×.

For pipeline 2, the limited EPC increases the latency of object tracking from ∼90 ms to ∼240 ms. With Visor's obliviousness, the net latency increases by 1.7×.

6.8.3 Comparison against Prior Work

We conclude our evaluation by comparing Visor against Obfuscuro [AJX+19], a state-of-the-art general-purpose system for oblivious program execution.

The current implementation of Obfuscuro supports a limited set of instructions, and hence cannot run the entire video analytics pipeline. On this note, we ported the OpenCV object cropping module to Obfuscuro, which requires only simple assignment operations. Cropping objects of size 128×128 and 16×16 (from a 1280×720 image) takes 8.5 hours and 8 minutes in Obfuscuro respectively,

versus 800 μs and 200 μs in Visor; making Visor faster by over 6 to 7 orders of magnitude. We note, however, that Obfuscuro targets stronger guarantees than Visor as it also aims to obfuscate the programs; hence, it is not a strictly apples-to-apples comparison. Nonetheless, the large gap in performance is hard to bridge, and our experiments demonstrate the benefit of Visor's customized solutions.

Other tools for automatically synthesizing or executing oblivious programs are either closed-source [RLT15, WGSW18], require special hardware [NFR⁺17, LHM⁺15, MLS⁺13], or require custom language support [CSJ⁺19]. However, we note that the authors of Raccoon [RLT15] (which provides similar levels of security as Visor) report up to $1000\times$ overhead on toy programs; the overhead would arguably be higher for complex programs like video analytics.

6.9 Discussion

Attacks on upper bounds. For efficiency, Visor extracts a fixed number of objects per frame based on a user-specified upper bound. However, this leaves Visor open to adversarial inputs: an attacker who knows this upper bound can attempt to confuse the analytics pipeline by operating many objects in the frame at the same time.

To mitigate such attacks, we suggest two potential strategies: (i) For frames containing $>= N$ objects (as detected in Section 6.7.2), process those frames off the critical path using worst-case bounds (*e.g.*, total number of pixels). While this approach leaks which specific frames contain $>= N$ objects, the leakage may be acceptable considering these frames are suspicious. (ii) Filter objects based on their properties like object size or object location: *e.g.*, for a traffic feed, only select objects at the center of the traffic intersection. This limits the number of valid objects possible per frame, raising the bar for mounting such attacks. One can also apply richer filters on the pipeline results and reprocess frames with suspicious content.

Oblivious-by-design encoding. Instead of designing oblivious versions of existing codecs, it may be possible to construct an oblivious-by-design coding scheme that is (i) potentially simpler, and (ii) performs better than Visor's oblivious decoding. This alternate design point is an interesting direction for future work. We note, however, that any such codec would need to produce a perfectly constant bitrate (CBR) per frame to prevent bitrate leakage over the network. While CBR codecs have been explored in the video literature, they are inferior to variable bitrate schemes (VBR) such as VP8 because they are lossier. In other words, an oblivious CBR scheme would consume greater bandwidth than VP8 to match its video quality (and therefore, VP8 with padding), though it may indeed be simpler. In Visor, we optimize for quality.

6.10 Related Work

To the best of our knowledge, Visor is the first system for the secure execution of vision pipelines. We discuss prior work related to various aspects of Visor.

Video processing systems.   A wide range of optimizations have been proposed to improve the efficiency of video analytic pipelines [HAB$^+$18, KEA$^+$17, JAB$^+$18, ZAB$^+$17]. These systems offer different design points for enabling trade-offs between performance and accuracy. Their techniques are complementary to Visor which can benefit from their performance efficiency.

Data-oblivious techniques.   Eppstein *et al.* [EGT10] develop data-oblivious algorithms for geometric computations. Ohrimenko *et al.* [OSF$^+$16] propose data-oblivious machine learning algorithms running inside CPU TEEs. These works are similar in spirit to Visor, but are not applicable to our setting.

Oblivious RAM [GO96] is a general-purpose cryptographic solution for eliminating access-pattern leakage. While recent advancements have reduced its computational overhead [SvDS$^+$13], it still remains several orders of magnitude more expensive than customized solutions. Oblix [MPC$^+$18] and Zerotrace [SGF18] enable ORAM support for applications running within hardware enclaves, but have similar limitations.

Various systems [RLT15, LHM$^+$15, AJX$^+$19, SRS17, NFR$^+$17, MLS$^+$13, WGSW18, CSJ$^+$19] also offer generic solutions for hiding access patterns at different levels, with the help of ORAM, specialized hardware, or compiler-based techniques. Generic solutions, however, are less efficient than customized solutions (such as Visor) which can exploit algorithmic patterns for greater efficiency.

Side-channel defenses for TEEs.   Visor provides systemic protection against attacks that exploit access pattern leakage in enclaves. Systems for data-oblivious execution (such as Obfuscuro [AJX$^+$19] and Raccoon [RLT15]) provide similar levels of security for general-purpose workloads, while Visor is tailored to vision pipelines.

In contrast, a variety of defenses have also been proposed to detect [CZRZ17] or mitigate *specific* classes of access-pattern leakage. For example, Cloak [GLS$^+$17a], Varys [OTK$^+$18], and Hyperrace [CWC$^+$18] target cache-based attacks; while T-SGX [SLKP17] and Shinde *et al.* [SCNS16] propose defenses for paging-based attacks. DR.SGX [BCD$^+$19] mitigates access pattern leakage by frequently re-randomizing data locations, but can leak information if the enclave program makes predictable memory accesses.

Telekine [HJM$^+$20] mitigates side-channels in GPU TEEs induced by CPU-GPU communication patterns, similar to Visor's oblivious CPU-GPU communication protocol (though the latter is specific to Visor's use case).

Secure inference.   Several recent works propose cryptographic solutions for CNN inference [JVC18, LJLA17, DGBL$^+$16, RRK18, RWT$^+$18] relying on homomorphic encryption and/or secure multi-party computation [Yao86]. While cryptographic approaches avoid the pitfalls of TEE-based CNN inference, the latter remains faster by orders of magnitude [TB19, HSS$^+$18].

6.11   Summary

We presented Visor, a system that enables privacy-preserving video analytics services. Visor uses a hybrid TEE architecture that spans both the CPU and the GPU, as well as novel data-oblivious

vision algorithms. Visor provides strong confidentiality and integrity guarantees, for video streams and models, in the presence of privileged attackers and malicious co-tenants. Our implementation of Visor shows limited performance overhead for the provided level of security.

Chapter 7

Collaborative Machine Learning on Encrypted Data

This chapter presents Secure XGBoost, a system that enables collaborative machine learning training and inference using trusted execution environments with side-channel mitigation.

7.1   Introduction

Secure XGBoost is a platform for secure collaborative gradient-boosted decision tree learning, based on the popular XGBoost library. In a nutshell, multiple clients (or data owners) can *collaboratively* use Secure XGBoost to train an XGBoost model on their collective data in a cloud environment while preserving the privacy of their individual data. Even though training is done on the cloud, Secure XGBoost ensures that the data of individual clients is revealed to neither the cloud environment nor other clients. Clients collaboratively orchestrate the training pipeline remotely, and Secure XGBoost guarantees that each client retains control of the computation that runs on its individual data.

At its core, Secure XGBoost leverages the protection offered by *secure hardware enclaves* to preserve the privacy of the data and the integrity of the computation even in the presence of a hostile cloud environment. On top of enclaves, Secure XGBoost adds a second layer of security that additionally protects the enclaves against a large class of *side-channel attacks*—namely, attacks induced by access pattern leakage (see Chapter 2). Even though the attacker cannot directly observe the data protected by the enclave, it can still infer sensitive information about the data by monitoring the enclave's memory access patterns during execution. To prevent such leakage, we redesign the training and inference algorithms in XGBoost to be *data-oblivious*, guaranteeing that the memory access patterns of enclave code does not reveal any information about sensitive data. In particular, our algorithms produce an *identical sequence* of disk, network and memory accesses that depend only on the public information, and are *independent* of the input data. Hence, they provably prevent all side-channels induced by access pattern leakage.

In implementing Secure XGBoost, we strived to preserve the XGBoost API as much as possible so that our system remains easy to use for data scientists. Our implementation has been adopted by
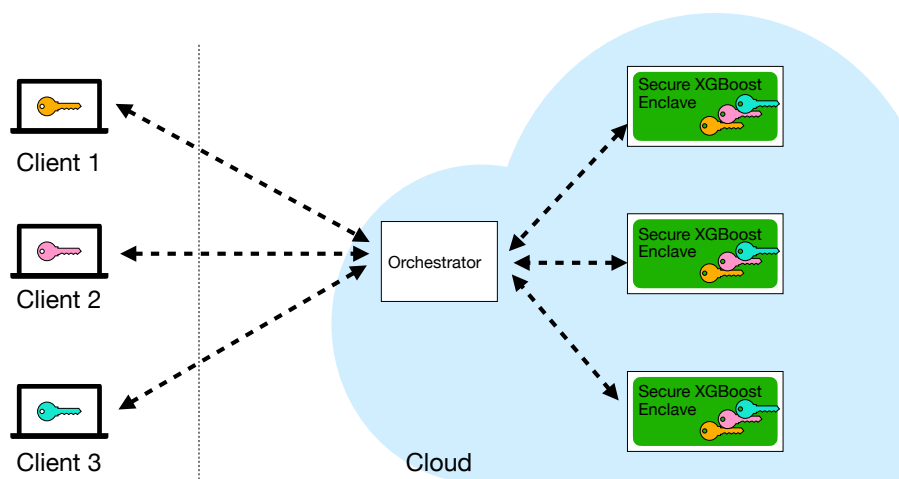
Figure 7.1: Parties invoke an orchestrator service at the cloud, which waits for calls from all parties before relaying the commands to the enclave cluster. Enclave inputs and outputs are always encrypted, and are decrypted only within the enclave or at client premises.

multiple industry partners, and is available as open-source software [Sec].

7.2 Overview

7.2.1 System model

In this section, we describe the different entities in a Secure XGBoost deployment. The entities consist of: (i) multiple data owners (or clients) who wish to collaboratively train a model on their individual data; and (ii) an untrusted cloud service that hosts the Secure XGBoost platform within a cluster of machines. The general architecture of Secure XGBoost is depicted in Figure 7.1.

Clients. A client refers to a party who wants to jointly train a model with other clients. The clients collectively execute the computation pipeline on the Secure XGBoost platform by remotely invoking its APIs.

Cloud service with enclaves. The cloud service consists of a cluster of virtual machines, each with hardware enclave support. Secure XGBoost distributes the computation across the cluster of hardware enclaves, which communicate with each other over TLS channels that begin and end inside the enclaves.

Additionally, an orchestrator service at the cloud mediates communication between clients and the Secure XGBoost platform deployed within enclaves.

7.2.2 Workflow

The following describes an end-to-end example workflow for using Secure XGBoost. We use the term 'command' to refer to a client's desired execution of a step in the computation process, *i.e.*, the APIs exposed by Secure XGBoost for data loading, training, etc.

1. The clients agree on a pre-determined sequence of commands that will be executed on Secure XGBoost (Section 7.4.1).

2. Clients attest the enclaves on the cloud (via the remote attestation procedure) to verify that the expected Secure XGBoost code has been securely loaded within each enclave (Section 7.4.2).

3. Each client C_i encrypts its data with a symmetric key k_i and uploads it to cloud storage (Section 7.4.3).

4. The clients submit signed commands to the orchestrator. The orchestrator aggregates all the client signatures and relays each command to Secure XGBoost. Secure XGBoost authenticates the signatures, ensuring that every client indeed issued the same command, and executes the command (Section 7.4.4).

5. Secure XGBoost returns the results of the command (*e.g.*, an encrypted trained model, or encrypted prediction results) to the orchestrator, who relays it to the clients. The process continues until all commands have been executed.

7.3 Threat Model and Security Guarantees

We describe the aims and capabilities of the attackers that Secure XGBoost protects against.

7.3.1 Threat model for the cloud and hardware enclaves

The threat model for the cloud and hardware enclaves is similar to our threat model for Visor, described in Chapter 6. In summary, the cloud service provider and the orchestrator service are untrusted. The trusted computing base includes the CPU package and its hardware enclave implementation, as well as our implementation of Secure XGBoost.

Similar to Visor, the design of Secure XGBoost is not tied to any specific hardware enclave; instead, Secure XGBoost builds on top of an *abstract* model of hardware enclaves where the attacker controls the server's software stack outside the enclave (including the OS), but cannot perform any attacks to glean information from inside the processor (including processor keys). The attacker can additionally observe the contents and access patterns of all (encrypted) pages in memory, for both data and code. We assume that the attacker can observe the enclave's memory access patterns at cache line granularity.

Secure XGBoost provides protection against *all channels of attack that exploit data-dependent access patterns at cache-line granularity*, which represent a large class of known attacks on enclaves

(*e.g.*, [GESM17, BMD$^+$17, SWG$^+$17, MIE17, HCP17, LSG$^+$17, XCP15, BWK$^+$17, LJF$^+$20]). Other attacks that violate our abstract enclave model, as discussed in Chapter 6, are out of scope.

7.3.2   Threat model for the clients

Each client expects to protect its data from the cloud service hosting the enclaves, as well as the other clients in the collaboration. Malicious clients may collude with each other and/or the cloud service to try and learn a victim client's data. They may also attempt to subvert the integrity of the computation by tampering with the computation steps (*i.e.*, the commands submitted for execution). Secure XGBoost protects the client data and computation in accordance with the threat model and guarantees from Section 7.3.1.

7.4   System Design

7.4.1   System setup

Secure XGBoost is launched at the cloud service within enclaves. It contains an embedded list of client names, along with the public key of a trusted certificate authority (CA), which it uses to verify a client's identity before establishing a connection with the client (described in Section 7.4.2). A single "master" enclave generates a 2048-bit RSA key pair (pk, sk) and a nonce N. The public key will be used to establish a secure channel of communication with the clients, and the nonce to ensure freshness of communicated messages.

Each client C_i generates a 256-bit symmetric key k_i. Each client also has its own 2048-bit RSA key pair (pk_i, sk_i), along with a certificate signed by a certificate authority (CA); the CA's public key is embedded in Secure XGBoost. The clients will use the certificate to authenticate themselves to Secure XGBoost.

7.4.2   Client-server attestation

Clients authenticate the Secure XGBoost deployment within the enclave cluster via remote attestation. More precisely, we logically arrange the enclaves in a tree topology; the enclave at the root of the tree is the "master" enclave. During attestation, each client attests only the "master" enclave to verify that the expected Secure XGBoost code has been securely loaded; in turn, each enclave in the tree (including the master) attests its children enclaves. As part of the attestation process, the enclaves establish TLS sessions with their neighboring enclaves. In addition, the master enclave sends the generated public key pk and a nonce N to the clients along with the signed attestation report.

Each client encrypts its key k_i using the enclave's public key pk, and signs the message. It then sends the signed message to the master enclave along with its certificate. The master enclave verifies each client's signed message, decrypts the symmetric key k_i, and percolates k_i to all attested enclaves in the cluster, giving each enclave the ability to decrypt data belonging to the client.

7.4.3 Data preparation and transfer

Each client uploads its encrypted data to cloud storage; enclaves retrieve the encrypted data from storage before training. To enable distributed data processing, each enclave must retrieve only a partition of the encrypted training data. This requirement precludes each client from encrypting its data as a single blob. Instead, to facilitate distributed processing of the encrypted data, clients encrypt each row in their data separately, which enables each enclave to retrieve, decrypt, and process only a subset of the rows.

Specifically, client C_i encrypts each row in its data (using its symmetric key k_i) as follows:

$$j, \ n_i, \ \texttt{Enc(row}_j), \ \texttt{MAC}(j||n_i||\texttt{Enc(row}_j))$$

Here, j is the index number of the row being encrypted; n_i is the total number of rows in C_i's data; $\texttt{Enc(row}_j)$ is an AES-GCM ciphertext over the j-th row; and $\texttt{MAC}(j||n_i||\texttt{Enc(row}_j))$ is an AES-GCM authentication tag computed over the ciphertext, the index number j and the total number of rows n_i. Including j and n_i within the authentication tag prevents the untrusted cloud service from tampering with the data (*e.g.*, by deleting or duplicating rows).

While processing a client's data, each enclave retrieves a subset of the encrypted rows. The enclaves then communicate to ensure that they together loaded n_i rows, and that all row indices from $j = 1 \ldots n_i$ were present in the retrieved data.

7.4.4 Collaborative API execution

Once all clients have uploaded their data to the cloud, they collectively invoke the APIs exposed by Secure XGBoost. Each API invocation requires consensus—Secure XGBoost executes an API call only if it receives the command from every client. This ensures that no processing can be performed on a particular client's data without that client's consent.

To make an API call, each client submits a signed command to the orchestrator:

$$\texttt{cmd} = \texttt{<seqn, func, params >}, \ \texttt{Sign(cmd)}$$

A command contains three fields: (i) a sequence number $\texttt{seqn} = (N||\texttt{ctr})$ that consists of the nonce N (obtained from the enclaves during attestation) concatenated with an incrementing counter; (ii) the API function $\texttt{func}$ being invoked; and (iii) the function parameters $\texttt{params}$. Including the sequence number ensures the freshness of the command, and prevents replay attacks on the system. The orchestrator aggregates the signed commands and relays them to the enclave cluster. Each enclave verifies that an identical command was submitted by every client before executing the corresponding function.

Once the function completes, Secure XGBoost produces a signed response and returns it to the clients via the orchestrator:

$$\texttt{resp} = \texttt{<seqn, result >}, \ \texttt{Sign(resp)}$$

```
void max(int x, int y,
         int* z) {
  if (x > y)
    *z = x;
  else
    *z = y;
}
```

Figure 7.2: Regular code

```
void max(int x, int y,
         int* z) {

  bool cond = ogreater(x, y);
  oassign(cond, x, y, z);

}
```

Figure 7.3: Oblivious code

The response contains the sequence number of the request (to cryptographically bind the response to the request), along with the results of the function (which are potentially encrypted with the clients' keys, depending on the function that was invoked).

7.5   Data-oblivious training and inference

To prevent side-channel leakage via access patterns, we design data-oblivious algorithms for training and inference. To implement the algorithms, we use a small set of data-oblivious primitives, similar to our implementation of Visor in Chapter 6. In this section, we first describe the primitives for completeness, and then show their usage in our algorithms.

7.5.1   Oblivious primitives

Our oblivious primitives operate solely on registers whose contents are loaded from and stored into memory using deterministic memory accesses. Since registers are private to the processor, any register-to-register operations cannot be observed by the attacker.

1) Oblivious comparisons (`oless`, `ogreater`, `oequal`).   These primitives can be used to obliviously compare variables, and are wrappers around the x86 `cmp` instruction.

2) Oblivious assignment (`oassign`).   The `oassign` primitive performs conditional assignments, moving a source to a destination register if a condition is true.

3) Oblivious sort (`osort`).   The `osort` primitive obliviously sorts a size n array by passing its inputs through a bitonic sorting network [Bat68], which performs an identical sequence of $O(n\log^2(n))$ carefully arranged compare-and-swap operations regardless of the input array values.

4) Oblivious array access (`oaccess`).   The `oaccess` primitive accesses the i-th element in an array without leaking i itself by scanning the array at cache-line granularity while performing `oassign` operations, setting the condition to true only at index i.

Example.   As an example, consider the code in Figure 7.2 that determines the maximum of two integers using a non-oblivious if-else statement. An attacker observing the memory addresses of accessed program instructions can identify whether $x > y$, depending on whether the code within the if-block or the else-block gets executed. To show how the oblivious primitives above can be used to implement higher-level data-oblivious code, Figure 7.3 depicts the data-oblivious version of the max
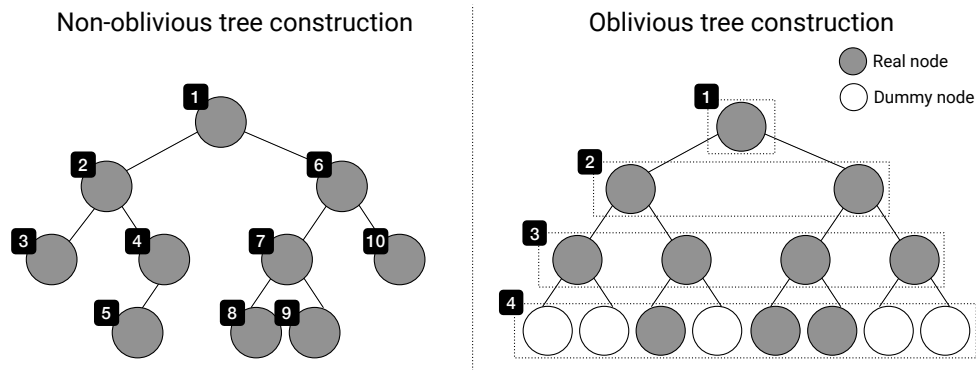
Figure 7.4: Illustration of oblivious training in Secure XGBoost. Numbers indicate the order in which nodes are added. Non-oblivious training adds nodes sequentially to the tree, while our algorithm constructs a full binary tree while adding nodes level-wise.

program from Figure 7.2. In this version, all instructions are executed sequentially, without any secret-dependent branches, causing the program to have identical memory access patterns regardless of the inputs values.

7.5.2 Oblivious training

Each enclave in the cluster loads a subset of the collected data, and then uses a distributed algorithm to train the model. In particular, we use XGBoost's histogram-based distributed algorithm (hist) for training an approximate model [CG16, His20], but redesign the algorithm in order to make it data-oblivious. In this algorithm, the data samples always remain distributed across all the enclave machines in the cluster, and the machines only exchange data summaries with each other. The summaries are used to construct a single tree globally and add it to the model's ensemble.

At a high level, the hist algorithm builds a tree in rounds, adding a node to the tree per round. Given a data sample $x \in \mathbb{R}^d$, at each node the algorithm chooses a feature j and a threshold t according to which the data samples are partitioned (*i.e.*, if $x(j) < t$, the sample is partitioned into the left subtree, otherwise the right). To add a node to the tree, each enclave in the cluster builds a histogram over its data for each feature; the boundaries of the bins in the histogram serve as potential splitting points for the corresponding feature. The algorithm combines the histograms across enclaves, and uses the aggregate statistics to find the best feature and splitting point. Note that in the absence of data-obliviousness the algorithm reveals a large amount of information via access-pattern leakage: *e.g.*, it leaks which feature was chosen at each node in the tree, as well the complete ordering of the data samples. We now describe the oblivious algorithm in more detail.

Oblivious histogram initialization.   Before a tree can be constructed, all the enclaves in the cluster first align on the boundaries of the histograms per feature. These boundaries are computed once and re-used for adding all the nodes in the tree, instead of computing new histogram boundaries per node.

1. Each enclave first obliviously creates a summary S of its data (one summary per feature): each element in the summary is a tuple (y, w), where y_j are the unique feature values in the list of data samples, and w_j are the sum of the weights of the corresponding samples. To create the summary, the enclave sorts its samples using `osort`. Then, it initializes an empty array S of size equal to the number of samples. Next, it scans the samples to identify unique values while maintaining a running aggregate of the weights: for each sample $\{x_i\}$ it updates $S[i]$ using `oselect`, either setting it to 0 (if $x_{i-1} = x_i$), or to the aggregated weight. At the end, it sorts S using `osort` to push all 0 values to the end of the list.

2. Each enclave then obliviously prunes its summary to a size $b+1$ (where b is a user-defined parameter for the maximum number of bins in the histogram). The aim of the pruning operation is to select $b+1$ elements from the list with ranks $0, \frac{|S|}{b}, \frac{2|S|}{b} \dots |S|$, where $|S|$ is the size of the summary. We do this obliviously as follows. First, the enclave sorts the summary using `osort`. Next, it scans the sorted summary, and for each element in the summary, it selects the element (using `oassign`) if its rank matches the next rank to be selected, otherwise it selects a dummy. Finally, it sorts the selected elements (which includes dummies), pushing the dummy elements to the end, and truncates the list.

3. Next, each enclave broadcasts its summary S. The summaries are pairwise combined into a "global" summary (one summary per feature) as follows: (i) Each pair of summaries is first merged into a single list using `osort`. The tuples in the merged summary are then scanned to identify adjacent values that are duplicates; the duplicates are zeroed out using `oaccess` while aggregating the weights. The merged summary is then sorted using `osort` to push all 0 values to the end of the list, and then truncated. (ii) Next, the merged summary is pruned as before into a summary of size b.

The global summary per feature computed in this manner represents the bins of a histogram, with the constituent values in the summary as the boundaries of different bins.

Oblivious node addition. The algorithm uses the feature histograms to construct a tree, adding nodes to the tree starting with the root. As nodes get added to the tree, the data gets partitioned at each node across its children. Here, we describe an oblivious subroutine for obliviously adding a node by finding the optimal split for the node, using the data samples that belong to the node.

1. Each enclave computes a histogram for each feature by scanning its data samples to compute a gradient per sample, followed by updating a single bin in each histogram using `oaccess` combined with `oassign`. The enclaves then broadcast their histograms.

2. The enclaves collectively sum up the histograms. Each enclave then computes a score function over the aggregated histogram, deterministically identifying the best feature to split by, as well as the split value.

3. Finally, each enclave partitions its data based on the split value: it simply updates a marker per sample (using `oassign`) that identifies which child node the sample belongs to.

Level-wise oblivious tree construction. A simple way to construct a tree is to sequentially add nodes to the tree as described above, until the entire tree is constructed. To prevent leaking information about the data or the tree: (i) the order in which nodes are added needs to be independent of the data; and (ii) a fixed number of nodes need to be added to the tree. At the same time, adding nodes sequentially by repeatedly invoking the node addition subroutine above is sub-optimal for performance. This is because oblivious node addition only uses the data that belongs to the node; however, concealing which data samples belong to the node either requires accessing each sample using `oaccess`, or scanning all the samples while performing dummy operations for those that do not belong to the node. Both these options impact performance adversely.

We simultaneously solve all the problems above by sequentially adding entire levels to the tree, instead of individual nodes. That is, we obliviously add all the nodes at a particular level of a tree in a *single scan* of all the data samples, as follows. For each data sample, we first use `oaccess` to obliviously fetch the histograms of the node that the sample belongs to. We then update the histograms as described in the subroutine above, and then obliviously write back the histogram to the node using `oaccess`.

Note that as a result of level-wise tree construction, we always build a full binary tree (unlike the non-oblivious algorithm) and some nodes in the tree are "dummy" nodes. These nodes are ignored during inference. Figure 7.4 illustrates how nodes are added to the tree during our oblivious training routine.

7.5.3 Oblivious inference

Inference normally occurs by traversing a tree from root to leaf and comparing the feature value of each interior node with the corresponding feature in the test data instance. To obliviously evaluate an XGBoost model on a data instance, we follow [OSF$^+$16]. In summary, we store each layer in the tree as an array, use the `oaccess` primitive to obliviously select the proper node at that layer, and use the `oless` primitive for comparison.

7.6 Implementation

Our Implementation of Secure XGBoost is open source [Sec]. Following XGBoost's implementation model, we provide a Python API on top of a core C++ library, imitating the XGBoost API as much as possible. An example of the Secure XGBoost API is shown in Figure 7.5. We used the Open Enclave SDK [Ope20] to interface between the untrusted host and the enclave and to enable Secure XGBoost to run agnostic of a specific hardware enclave; Mbed TLS [Mbe20] for cryptography and for secure communication between enclaves; and gRPC [gRP20] for client-server communication.

7.7 Evaluation

We ran experiments on Secure XGBoost using a synthetic dataset obtained from Ant Financial, consisting of $100,000$ data samples with 126 features. Our experiments compare three systems: vanilla

```
import securexgboost as xgb

# Initialize client and connect to enclave cluster
xgb.init_client(user_name="user1",
            sym_key_file="key.txt",
            priv_key_file="user1.pem",
            cert_file="user1.crt")

# Client side remote attestation to authenticate enclaves
xgb.attest()

# Load the encrypted data and associate it with a user
dtrain = xgb.DMatrix({"user1": "train.enc"})
dtest = xgb.DMatrix({"user1": "test.enc"})

params = {
    "objective": "binary:logistic",
    "gamma": "0.1",
    "max_depth": "3"
}

# Train a model
num_rounds = 5
booster = xgb.train(params, dtrain, num_rounds)

# Get encrypted predictions and decrypt them
predictions, num_preds = booster.predict(dtest)
```

Figure 7.5: Example client code in Secure XGBoost. Functions highlighted in red are additions to the existing XGBoost library. Functions highlighted in blue exist in XGBoost but were modified for Secure XGBoost.

XGBoost; encrypted Secure XGBoost (a version of Secure XGBoost without obliviousness); and oblivious Secure XGBoost (Secure XGBoost with obliviousness enabled). We ran our experiments on Microsoft's Azure Confidential Computing service. We used DC4s_V2 machines, which have support for Intel SGX enclaves, and are equipped with 4 vCPUs, 16 GiB of memory, and a 112 MiB enclave page cache.

Figure 7.6 shows our training results. In general, encrypted Secure XGBoost incurs $4.5 \times -5.1 \times$ overhead compared to vanilla XGBoost, which provides no security. Oblivious Secure XGBoost incurs $16.7 \times -178.2 \times$ overhead over encrypted Secure XGBoost. The main takeaway is that one has to be careful in tuning the hyperparameters by adjusting the number of bins, the number of levels per tree and the number of trees. For example, decreasing the number of bins while increasing the number of trees could improve performance while maintaining the same accuracy.

7.8 Conclusion

In this chapter we described Secure XGBoost, a privacy-preserving system that enables multiparty training and inference of XGBoost models. Secure XGBoost protects the privacy of each party's
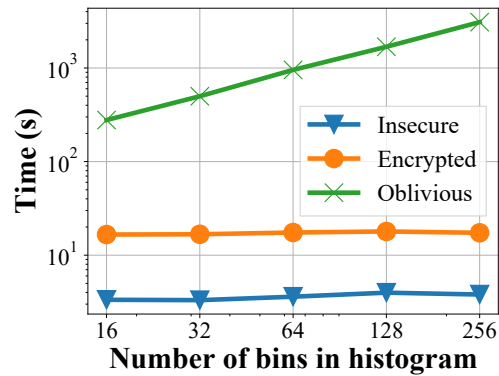
Figure 7.6: Evaluation comparison among the insecure baseline, and encrypted as well as oblivious Secure XGBoost

data as well as the integrity of the computation with the help of hardware enclaves. Crucially, Secure XGBoost augments the security of the enclaves using novel *data-oblivious* algorithms that prevent access side-channel attacks on enclaves induced via access pattern leakage. Our implementation is available as open-source software.

Chapter 8

Conclusion

How should we design systems that can analyze confidential data, while simultaneously meeting the goals of security and efficiency? Existing approaches for secure computation often fall short on one or both of these counts—cryptographic approaches are often not efficient, and solutions based on trusted hardware suffer from side-channel vulnerabilities. To support complex workloads, it is important for systems to satisfy both requirements in order to be meaningfully useful.

This dissertation shows that it is possible to design secure *and* efficient systems that can meet the demands of modern applications. To guide the design of such systems, we distill a set of design principles based on the properties of the different approaches for secure computation. We illustrate these principles by applying them in the design of a range of applications, across various scenarios where confidential computing is necessary. Specifically, in Chapters 3 and 4 we design cryptographic systems for query execution and analytics, Arx and Senate, that respectively target outsourced computing and collaborative computing scenarios. We then turn to the design of applications with stricter performance requirements. In Chapters 5 to 7 we design SafeBricks, Visor, and Secure XGBoost respectively with the help of trusted execution environments, for applications such as network functions, machine learning, and video analytics, in the outsourced as well as collaborative settings. We have also combined the ideas in this dissertation into the MC^2 platform which is available as open-source software [MC2]. Several teams in industry have adopted MC^2 for their own use cases, from applications in telecommunications to financial services.

We expect that the demand for confidential computing platforms will only increase in the future. The growing threat of data breaches has been mirrored by a swell in public concern around the privacy of data. In response to this concern, governments around the world are enacting stricter privacy laws governing how confidential data might be used and processed [Kar19]. Privacy-preserving platforms, like the ones designed in this dissertation, offer a promising solution for protecting data while still being able to use it. Undoubtedly, such platforms will continue to evolve and improve. We hope that the approach we take in this dissertation will, at the very least, help inform the design of future systems and serve as a useful point of reference.

8.1 Future Directions

Looking ahead, there are several avenues for improving and building upon this work.

Synthesizing complex data analysis pipelines. From a cryptographic perspective, the work in this dissertation focused mainly on database query execution and analytics. Efficiently synthesizing cryptographic protocols for machine learning pipelines is a timely direction for future work. Existing efforts in privacy-preserving machine learning focus on narrower problems—they develop specialized protocols for individual tasks such as linear regression, ridge regression, or neural network inference. However, real-world machine learning workflows are complex and typically consist of a pipeline of stages: from data preparation to feature engineering to model training, followed by model deployment. A privacy-preserving system that can support such pipelines would be a valuable contribution.

Augmenting security with complementary approaches. In the case of collaborative computation, our secure computation-based approach reveals the results to the parties, which may also leak information about the underlying data. This leakage can potentially be mitigated by techniques such as differential privacy which prevents leakage by adding noise to the results, and is complementary to secure computation. Given a differentially private mechanism for noising the computation results, it is fairly straightforward to integrate it within trusted execution. However, reconciling differentially privacy with cryptographic protocols is challenging and requires care, especially in the presence of malicious behavior.

Combining cryptographic schemes with trusted execution. An alternate design point ripe for exploration would be to combine the guarantees of cryptographic schemes with trusted execution into a "hybrid" architecture. This can potentially benefit system performance for several reasons. It can help offset the overhead of cryptographic protocols by offloading some compute (with side-channel mitigation) to enclaves, and simplify the design of the overall scheme. It can help alleviate the memory burden on enclaves by moving some computation outside the enclaves and executing it cryptographically instead. It also allows system designers to limit the security properties required of either approach, which may overall lead to a more efficient system—*e.g.*, using enclaves only for integrity protection, and cryptographic protocols only for privacy. At the same time, the success of such a hybrid design depends on whether it is also able to circumvent the shortcomings of the two approaches, which is challenging.

Bibliography

[ABB⁺17]   Ganesh Ananthanarayanan, Victor Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath Sivalingam, and Sudipta Sinha. Real-time Video Analytics – the killer app for edge computing. *IEEE Computer*, 2017.

[ABG⁺05]   Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep A secret: A distributed architecture for secure database services. In *CIDR*, 2005.

[AC75]   Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 1975.

[AEAEM09]   Divyakant Agrawal, Amr El Abbadi, Faith Emekci, and Ahmed Metwally. Database Management as a Service: Challenges and Opportunities. In *ICDE*, 2009.

[AEK⁺13]   Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. A secure coprocessor for database applications. In *FPL*, 2013.

[AGJS13]   Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *HASP*, 2013.

[AHMR15]   Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In *EUROCRYPT*, 2015.

[AJX⁺19]   Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A Commodity Obfuscation Engine on Intel SGX. In *NDSS*, 2019.

[AKL12]   Emmanuel A. Abbe, Amir E. Khandani, and Andrew W. Lo. Privacy-Preserving Methods for Sharing Financial Risk Exposures. *American Economic Review*, 2012.

[AKSX04]   Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *SIGMOD*, 2004.

[Ama]   Amazon Rekognition. https://aws.amazon.com/rekognition/.

[AMS+16] Hassan Jameel Asghar, Luca Melis, Cyril Soldani, Emiliano De Cristofaro, Mohamed Ali Kaafar, and Laurent Mathy. SplitBox: Toward Efficient Private Network Function Virtualization. In *HotMiddlebox*, 2016.

[Ary] Aryaka. http://www.aryaka.com/.

[AS89] C. R. Aragon and R. G. Seidel. Randomized search trees. In *FOCS*, 1989.

[ATG+16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 2016.

[Att] Attestation Service for Intel SGX. https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf.

[AWW12] George Weilun Ang, John Harold Woelfel, and Terrence Peter Woloszyn. System and Method of Sort-Order Preserving Tokenization. US Patent Application 13/450,809, 2012.

[AZM10] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive Black-box Mitigation of Timing Channels. In *CCS*, 2010.

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In *AsiaCCS*, 2012.

[Bar] Barracuda Networks. https://www.barracuda.com/.

[Bat68] K. E. Batcher. Sorting Networks and Their Applications. In *Proceedings of the Spring Joint Computer Conference*, 1968.

[BBB+17] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System Programming in Rust: Beyond Safety. In *HotOS*, 2017.

[BBV08] Thierry Bouwmans, Fida El Baf, and Bertrand Vachon. Background Modeling using Mixture of Gaussians for Foreground Detection – A Survey. *Recent Patents on Computer Science*, 2008.

[BCD+19] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX Using Data Location Randomization. In *ACSAC*, 2019.

[BCLK17] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory. In *DSN*, 2017.

[BCLO09]   Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-Preserving Symmetric Encryption. In *EUROCRYPT*, 2009.

[BCO11]   Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, 2011.

[BDNP08]   Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A System for Secure Multi-party Computation. In *CCS*, 2008.

[Bea18]   Steven Bearak. 2018 Data Breaches, 2018. https://www.identityforce.com/blog/2018-data-breaches.

[Bee10]   Dean Beeby. Rogue tax workers snooped on ex-spouses, family members, 2010. https://goo.gl/WNKoCS.

[BEE+17]   Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. SMCQL: Secure Querying for Federated Databases. In *VLDB*, 2017.

[Bek20]   Eugene Bekker. 2020 Data Breaches, 2020. https://www.identityforce.com/blog/2020-data-breaches.

[BEM+17]   Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnés, and Bernhard Seefeld. Prochlo: Strong Privacy for Analytics in the Crowd. In *SOSP*, 2017.

[Ber]   Berkeley Extensible Software Switch (BESS). http://span.cs.berkeley.edu/bess.html.

[BFLV12]   Dimitrios Bisias, Mark Flood, Andrew W. Lo, and Stavros Valavanis. A Survey of Systemic Risk Analytics. *Annual Review of Financial Economics*, 2012.

[BGW88]   Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, 1988.

[BHE+18]   Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. Shrinkwrap: Differentially-Private Query Processing in Private Data Federations. In *VLDB*, 2018.

[BHJP14]   Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. A Survey of Provably Secure Searchable Encryption. *ACM Computing Surveys (CSUR)*, 2014.

[BHK+17]   Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security*, 2017.

[BHR12]   Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of Garbled Circuits. In *CCS*, 2012.

[BIK⁺17]   Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *CCS*, 2017.

[BJB15]   Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and Efficient Elimination of Cache and Timing Side Channels. *arxiv:1506.00189*, 2015.

[BKQ⁺11]   J. Banksoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu. VP8 Data Format and Decoding Guide. RFC 6386, 2011.

[BLR⁺14]   Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. In *EUROCRYPT*, 2014.

[BLW08]   Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.

[BLW⁺19]   Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srini Devadas. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *MICRO*, 2019.

[BMD⁺17]   Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[BMO17]   Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *CCS*, 2017.

[BMR90]   Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, 1990.

[BMW⁺18]   Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[Bos16]   Raphael Bost. Sophos - Forward Secure Searchable Encryption. In *CCS*, 2016.

[BPH14]   Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*, 2014.

[BPP16]   Tobias Boelter, Rishabh Poddar, and Raluca Ada Popa. A Secure One-Roundtrip Index for Range Queries. Cryptology ePrint Archive, Report 2016/568, 2016. https://eprint.iacr.org/2016/568.

[BS13]   Sumeet Bajaj and Radu Sion. HIFS: History Independence for File Systems. In *CCS*, 2013.

[Bud]   Budget Manager. https://goo.gl/chFmct.

[BWK+17]   Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

[BWX11]   Jim Bankoski, Paul Wilkins, and Yaowu Xu. Technical overview of VP8, an open source video codec for the web. In *International Conference on Multimedia and Expo (ICME)*, 2011.

[CBC+18]   Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious Serializable Transactions in the Cloud. In *OSDI*, 2018.

[CCX+19]   Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*, 2019.

[Cen17]   Center for Disease Control and Prevention (CDC): Diseases and Conditions A-Z Index, 2017. https://www.cdc.gov/DiseasesConditions.

[CG16]   Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. *CoRR*, abs/1603.02754, 2016.

[CGB17]   Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *NSDI*, 2017.

[CGG+19]   Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[CGKO06]   Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*, 2006.

[CGPR15]   David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks against Searchable Encryption. In *CCS*, 2015.

[CGT12]   Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and Private Computation of Cardinality of Set Intersection and Union. In *CANS*, 2012.

[Chi]   Chino.io: Security and Privacy for Health Data in the EU. https://chino.io/.

[Cipa]   CipherCloud. CASB+ Platform. http://www.ciphercloud.com.

[Cipb]   CipherCloud. Tokenization. http://www.ciphercloud.com/tokenization.

[CJJ+13]   David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, 2013.

[CJJ+14]   David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*, 2014.

[CJS12]   Jung Hee Cheon, Stanislaw Jarecki, and Jae Hong Seo. Multi-party privacy-preserving set intersection with quasi-linear complexity. *IEICE Transactions*, 95-A(8):1366–1378, 2012.

[CKT10]   Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-Complexity Private Set Intersection Protocols Secure in Malicious Model. In *ASIACRYPT*, 2010.

[CKW17]   Michael Coughlin, Eric Keller, and Eric Wustrow. Trusted Click: Overcoming Security Issues of NFV in the Cloud. In *SDN-NFV Security*, 2017.

[CLD16]   Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*, 2016.

[CLS09]   Sherman S. M. Chow, Jie-Han Lee, and Lakshminarayanan Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*, 2009.

[CLWW16]   Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. Practical Order-Revealing Encryption with Limited Leakage. In *IACR-FSE*, 2016.

[Cod70]   E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 1970.

[CS13]   Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *ASPLOS*, 2013.

[CSJ+19]   Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A DSL for Timing-Sensitive Computation. In *PLDI*, 2019.

[CST+11]   Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SOCC*, 2011.

[CWC+18]   Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, Xi-aoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *IEEE S&P*, 2018.

[CZRZ17]   Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *AsiaCCS*, 2017.

[DDC16]   F. Betül Durak, Thomas M. DuBuisson, and David Cash. What Else is Revealed by Order-Revealing Encryption? In *CCS*, 2016.

[DGBL+16] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *ICML*, 2016.

[dir19]   Privilege Escalation in Ubuntu Linux, 2019. https://shenaniganslabs.io/2019/02/13/Dirty-Sock.html.

[DKL+13]   Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, 2013.

[DKM+06]   Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our Data, Ourselves: Privacy via Distributed Noise Generation. In *EURO-CRYPT*, 2006.

[DR14]   Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.*, 2014.

[DSMRY11] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Secure Efficient Multiparty Computing of Multivariate Polynomials and Applications. In *ACNS*, 2011.

[EGT10]   David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving Data-oblivious Geometric Algorithms for Geographic Data. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, 2010.

[EKM+14]   Fabienne Eigner, Aniket Kate, Matteo Maffei, Francesca Pampaloni, and Ivan Pry-valov. Differentially private data aggregation with optimal utility. In *ACSAC*, 2014.

[Eme]   Emerging Threats Open Rulesets. https://rules.emergingthreats.net/.

[EMP]   AGMPC Framework. https://github.com/emp-toolkit/emp-agmpc.

[ETS]   ETSI White Paper No. 11. Mobile Edge Computing – A key technology towards 5G. https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf.

[Eur]   European Telecommunications Standards Institute. NFV Whitepaper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[EW16]   Michael Egorov and MacLane Wilkison. ZeroDB white paper. *arXiv:1602.07168*, 2016.

[EYC+16]   Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*, 2016.

[EZ20]   Saba Eskandarian and Matei Zaharia. ObliDB: Oblivious Query Processing using Hardware Enclaves. 2020.

[FFm]   FFmpeg. https://ffmpeg.org/.

[FJK+15]   Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*, 2015.

[FJN+13]   Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient Secure Two-Party Computation from General Assumptions. In *EUROCRYPT*, 2013.

[FJNT15]   Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. TinyLEGO: An Interactive Garbling Scheme for Maliciously Secure Two-party Computation. Cryptology ePrint Archive, Report 2015/309, 2015. https://eprint.iacr.org/2015/309.

[FJR15]   Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures. In *CCS*, 2015.

[FLJ+14]   Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in Pharmacogenetics: An End-to-end Case Study of Personalized Warfarin Dosing. In *USENIX Security*, 2014.

[FNO19]   Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Private Set Intersection with Linear Communication from General Assumptions. In *WPES*, 2019.

[For]   Fortinet. https://www.fortinet.com/.

[Gen09]   Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[GESM17]   Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *EuroSec*, 2017.

[GGMP16]   Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty RAM computation in constant rounds. In *TCC*, 2016.

[GHH⁺14]   Patrick Grofig, Martin Haerterich, Isabelle Hang, Florian Kerschbaum, Mathias Kohler, Andreas Schaad, Axel Schroepfer, and Walter Tighzert. Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data. In *Sicherheit*, 2014.

[GLMP19]   Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *IEEE S&P*, 2019.

[GLO15]   Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-Box Garbled RAM. In *FOCS*, 2015.

[GLOS15]   Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *STOC*, 2015.

[GLS⁺17a]   Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*, 2017.

[GLS⁺17b]   Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *IEEE S&P*, 2017.

[GMP15]   Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption. Cryptology ePrint Archive, Report 2015/1010, 2015. http://eprint.iacr.org/2015/1010.

[GMW87]   Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play ANY Mental Game. In *STOC*, 1987.

[GO96]   Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 1996.

[Gol04a]   Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques*. Cambridge University Press, 2004.

[Gol04b]   Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[Gol09]   Daniel Golovin. B-Treaps: A Uniquely Represented Alternative to B-Trees. In *ICALP*, 2009.

[Gooa]   Google. Encrypted BigQuery client. https://github.com/google/encrypted-bigquery-client.

[Goob]   Google. Transparency Report. https://www.google.com/transparencyreport/userdatarequests/US/.

[Gooc]   Google AI. Federated Learning: Collaborative Machine Learning without Centralized Training Data. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[GPP+12]   Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*, 2012.

[gRP20]   gRPC. 2020. https://grpc.io/docs/.

[GRS17]   Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why Your Encrypted Database Is Not Secure. In *HotOS*, 2017.

[GSB+16]   Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-Abuse Attacks against Order-Revealing Encryption. Cryptology ePrint Archive, Report 2016/895, 2016. http://eprint.iacr.org/2016/895.

[GSB+17]   Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-Preserving Distributed Linear Regression on High-Dimensional Data. In *PETS*, 2017.

[H26]   H264 Codec. https://www.itu.int/rec/T-REC-H.264.

[HAB+18]   Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *OSDI*, 2018.

[HAJ+14]   Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Xiaodong Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *CCS*, 2014.

[HCP17]   Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *ATC*, 2017.

[HEK12]   Yan Huang, David Evans, and Jonathan Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *NDSS*, 2012.

[HILM02]   Hakan Hacigumus, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *SIGMOD*, 2002.

[His20]   Histogram-based training in XGBoost. 2020. https://github.com/dmlc/xgboost/issues/1950.

[HJM⁺20]   Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure Computing with Cloud GPUs. In *NSDI*, 2020.

[HKHH17]   Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module. In *APNet*, 2017.

[HSS⁺18]   Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving Machine Learning as a Service. *arXiv:1803.05961*, 2018.

[HV17]   Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable Multi-Party Private Set-Intersection. In *PKC*, 2017.

[HY16]   Carmit Hazay and Avishay Yanai. Constant-round maliciously secure two-party computation in the RAM model. In *TCC*, 2016.

[HZRS16]   Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[HZX⁺16]   Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *OSDI*, 2016.

[IBM]   IBM Cloud Data Shield. https://www.ibm.com/cloud/data-shield.

[ICT]   ICTF data. https://ictf.cs.ucsb.edu/.

[IKN⁺17]   Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private Intersection-Sum Protocol with Applications to Attributing Aggregate Ad Conversions. Cryptology ePrint Archive, Report 2017/738, 2017. https://eprint.iacr.org/2017/738.

[Inta]   Intel Data Plane Development Kit (DPDK). http://dpdk.org/.

[Intb]   Intel SGX and Side-Channels. https://software.intel.com/content/www/us/en/develop/articles/intel-sgx-and-side-channels.html.

[Intc]   Intel SGX Developer Guide: Protection from Side-Channel Attacks. https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top/protection-from-sidechannel-attacks.html.

[iQr]   iQrypt: Encrypt and query your database. http://iqrypt.com/.

[JAB⁺18]   Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable Adaptation of Video Analytics. In *SIGCOMM*, 2018.

[Jai89]   Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall, 1989.

[Jar12]   Jeff Jarmoc. SSL/TLS Interception Proxies and Transitive Trust. In *Black Hat*, 2012.

[JFK16]   Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A Complete Key Recovery Timing Attack on a GPU. In *HPCA*, 2016.

[JFK17]   Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A Novel Side-Channel Timing Attack on GPUs. In *Proceedings of the on Great Lakes Symposium on VLSI (GLSVLSI)*, 2017.

[JLLK17]   Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[JMK⁺17]   Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *NSDI*, 2017.

[JNHS18]   Noah M. Johnson, Joseph P. Near, Joseph M. Hellerstein, and Dawn Song. Chorus: Differential Privacy via Query Rewriting. *arXiv:1809.07750*, 2018.

[JNS18]   Noah Johnson, Joseph P. Near, and Dawn Song. Towards Practical Differential Privacy for SQL Queries. In *VLDB*, 2018.

[JSD⁺14]   Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM*, 2014.

[JTK⁺19]   Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous Isolated Execution for Commodity GPUs. In *ASPLOS*, 2019.

[JVC18]   Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security*, 2018.

[JWJ⁺14]   EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[KAK10]   Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa. MV-OPES: Multivalued-Order Preserving Encryption Scheme: A Novel Scheme for Encrypting Integer Value to Many Different Values. *IEICE Trans. Info. & Sys.*, 2010.

[Kar19]   Karolina Lubowicka. 6 new privacy laws around the globe you should pay attention to, 2019. https://piwik.pro/blog/privacy-laws-around-globe/.

[KBV13]   Liina Kamm, Dan Bogdanov, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 2013.

[KEA$^+$17]   Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. No-Scope: Optimizing Neural Network Queries over Video at Scale. In *VLDB*, 2017.

[KFPC16]   Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *IEEE S&P*, 2016.

[KGM$^+$14]   Jeremy Kepner, Vijay Gadepally, Peter Michaleas, Nabil Schear, Mayank Varia, Arkady Yerukhimovich, and Robert K. Cunningham. Computing on Masked Data: A High Performance Method for Improving Big Data Veracity. *arXiv:1406.5751*, 2014.

[KHH$^+$17]   Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *NSDI*, 2017.

[KKNO16]   Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic Attacks on Secure Outsourced Databases. In *CCS*, 2016.

[KLL$^+$16]   Myungsun Kim, Hyung Tae Lee, San Ling, Shu Qin Ren, Benjamin Hong Meng Tan, and Huaxiong Wang. Better Security for Queries on Encrypted Databases. Cryptology ePrint Archive, Report 2016/470, 2016. http://eprint.iacr.org/2016/470.

[KM16]   Seny Kamara and Tarik Moataz. SQL on Structurally-Encrypted Databases. Cryptology ePrint Archive, Report 2016/453, 2016. http://eprint.iacr.org/2016/453.

[KMC$^+$00]   Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 2000.

[KMMB12]   Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class GPU Resource Management in the Operating System. In *ATC*, 2012.

[KMP$^+$17]   Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical Multi-party Private Set Intersection from Symmetric-Key Techniques. In *CCS*, 2017.

[KNR$^+$17]   Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. DUPLO: unifying cut-and-choose for garbled circuits. In *CCS*, 2017.

[KOS16]   Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, 2016.

[KOV18]   Ian Kash, Greg O'Shea, and Stavros Volos. DC-DRF: Adaptive multi-resource sharing at public cloud scale. In *SOCC*, 2018.

[KPR12]   Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *CCS*, 2012.

[KPR18]   Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ Great Again. In *EUROCRYPT*, 2018.

[KRTW19]   Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable Private Set Union from Symmetric-Key Techniques. In *ASIACRYPT*, 2019.

[KS05]   Lea Kissner and Dawn Song. Privacy-Preserving Set Operations. In *CRYPTO*, 2005.

[KSH+15]   Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *HotNets*, 2015.

[KSS09]   Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *CANS*, 2009.

[Kun]   Kuna AI. https://getkuna.com/pages/kuna-ai.

[Kur14]   Kaoru Kurosawa. Garbled Searchable Symmetric Encryption. In *FC*, 2014.

[KY18]   Marcel Keller and Avishay Yanai. Efficient maliciously secure multiparty computation for RAM. In *EUROCRYPT*, 2018.

[LCS+14]   Billy Lau, Simon P. Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. Mimesis Aegis: A Mimicry Privacy Shield-A System's Approach to Data Privacy on Public Cloud. In *USENIX Security*, 2014.

[Lea]   Leanote. https://leanote.com/.

[LHKR10]   Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Authenticated Index Structures for Aggregation Queries. *ACM Trans. Info. & Sys. Sec.*, 2010.

[LHM+15]   Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS*, 2015.

[LJF+20]   Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security*, 2020.

[LJLA17]   Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *CCS*, 2017.

[LKL]   LKL: Linux Kernel Library. https://lkl.github.io.

[LKS+19]   Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. Keystone: An Open Framework for Architecting TEEs. *arXiv:1907.10119*, 2019.

[LLP+20]   Andrew Law, Chester Leung, Rishabh Poddar, Raluca Ada Popa, Chenyu Shi, Octavian Sima, Chaofan Yu, Xingmeng Zhang, and Wenting Zheng. Secure Collaborative Training and Inference for XGBoost. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice (PPMLP)*, 2020.

[LMP18]   Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage, 2018.

[LO17]   Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In *CRYPTO*, 2017.

[Low99]   David Lowe. Object Recognition from Local Scale-Invariant Features. In *ICCV*, 1999.

[Low04]   David Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vision*, 2004.

[LPL+09]   Seungmin Lee, Tae-Jun Park, Donghyeok Lee, Taekyong Nam, and Sehun Kim. Chaotic Order Preserving Encryption for Efficient and Secure Queries on Databases. *IEICE Trans. Info. & Sys.*, 2009.

[LSG+17]   Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[LSP+16]   Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In *NSDI*, 2016.

[LW12]   Dongxi Liu and Shenlu Wang. Programmable Order-Preserving Secure Index for Encrypted Database Query. In *CLOUD*, 2012.

[LW13]   Dongxi Liu and Shenlu Wang. Nonlinear order preserving index for encrypted database query in service cloud environments. *Concurrency and Computation: Practice and Experience*, 2013.

[LW16]   Kevin Lewi and David J. Wu. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In *CCS*, 2016.

[lwI]   lwIP: A lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip/.

[LWN⁺15]   Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A Programming Framework for Secure Computation. In *IEEE S&P*, 2015.

[MAB⁺13]   Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[MADCK16]   Luca Melis, Hassan Jameel Asghar, Emiliano De Cristofaro, and Mohamed Ali Kaafar. Private Processing of Outsourced Network Functions: Feasibility and Constructions. In *SDN-NFV Security*, 2016.

[MAK⁺17]   Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security*, 2017.

[Maz]   MazuNAT. https://github.com/kohler/click/blob/master/conf/mazu-nat.click.

[Mbe20]   Mbed TLS. 2020. https://tls.mbed.org/.

[MC2]   MC^2 Platform. https://github.com/mc2-project/mc2.

[MDC16]   Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient Private Statistics with Succinct Sketches. In *NDSS*, 2016.

[Mer79]   Ralph Merkle. *Secrecy, authentication and public key systems / A certified digital signature*. PhD thesis, Stanford University, 1979.

[MGC⁺16]   Benjamin Mood, Debayan Gupta, Henry Carter, Kevin R. B. Butler, and Patrick Traynor. Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter. In *EuroS&P*, 2016.

[Mica]   Microsoft. Law Enforcement Requests Report. https://www.microsoft.com/en-us/about/corporate-responsibility/lerr.

[Micb]   Microsoft Azure Confidential Computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[Micc]   Microsoft Azure Media Analytics. https://azure.microsoft.com/en-us/services/media-services/media-analytics/.

[Micd]   Microsoft Project Rocket. https://aka.ms/Rocket.

[Mice]   Microsoft Rocket Video Analytics Platform. https://github.com/microsoft/Microsoft-Rocket-Video-Analytics-Platform.

[Micf]   Microsoft SQL Server. Always Encrypted Database Engine. https://goo.gl/51LwQ9.

[MIE17]   Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[Mir]   Mirage TCP/IP stack. https://github.com/mirage/mirage-tcpip.

[MLS+13]   Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *CCS*, 2013.

[MNPS04]   Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay – A Secure Two-party Computation System. In *USENIX Security*, 2004.

[MOG+20]   Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *IEEE S&P*, 2020.

[MPC+18]   Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An Efficient Oblivious Search Index. In *IEEE S&P*, 2018.

[MT09]   Di Ma and Gene Tsudik. A New Approach to Secure Logging. *Trans. Storage*, 2009.

[MWES18]   Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations. In *CT-RSA*, 2018.

[MZ19]   Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2019.

[Nav15]   Muhammad Naveed. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. Cryptology ePrint Archive, Report 2015/668, 2015. http://eprint.iacr.org/2015/668.

[Net]   NetBricks. http://netbricks.io.

[Net17]   Netty Project, 2017. http://netty.io/.

[NFR+17]   Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. HOP: Hardware makes Obfuscation Practical. In *NDSS*, 2017.

[NH12]   Arjun Narayan and Andreas Haeberlen. DJoin: Differentially Private Join Queries over Distributed Databases. In *OSDI*, 2012.

[NKAG17]   Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. Constructing and Characterizing Covert Channels on GPGPUs. In *MICRO*, 2017.

[NKW15]   Muhammad Naveed, Seny Kamara, and Chares V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS*, 2015.

[NNOB12]   Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO*, 2012.

[NNQAG18]   Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks are Practical. In *CCS*, 2018.

[NO09]   Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In *TCC*, 2009.

[Nod]   NodeBB. https://nodebb.org/.

[Nou]   Nouveau: Accelerated open source driver for NVIDIA cards. https://nouveau.freedesktop.org/wiki.

[NPG14]   Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. Dynamic Searchable Encryption via Blind Storage. In *IEEE S&P*, 2014.

[NSV+15]   David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *SIGCOMM*, 2015.

[NT01]   Moni Naor and Vanessa Teague. Anti-persistence: History Independent Data Structures. In *STOC*, 2001.

[NVI]   NVIDIA GPU Instruction Set Reference. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref.

[NWI+13]   Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *IEEE S&P*, 2013.

[NWI+15]   Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S&P*, 2015.

[OKKM13]   Wakaha Ogata, Keita Koiwa, Akira Kanaoka, and Shin'ichiro Matsuo. Toward Practical Searchable Symmetric Encryption. In *IWSec*, 2013.

[OLMS17]   Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exit-Less OS Services for SGX Enclaves. In *EuroSys*, 2017.

[Ope]   OpenCV. https://opencv.org/.

[Ope20]   Open Enclave SDK, 2020. https://openenclave.io.

[ÖSC03]   Gultekin Özsoyoglu, David A. Singer, and Sun S. Chung. Anti-Tamper Databases: Querying Encrypted Databases. In *DBSec*, 2003.

[OSF+16]   Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security*, 2016.

[OTK+18]   Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *ATC*, 2018.

[Pai99]   Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[Pala]   Palo Alto Networks. https://www.paloaltonetworks.com/.

[Palb]   Palo Alto Networks. Virtualization Features. https://goo.gl/ezntv6.

[PAS+20]   Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *USENIX Security*, 2020.

[PBC+16]   Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big Data Analytics over Encrypted Datasets with Seabed. In *OSDI*, 2016.

[PBP16]   Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An Encrypted Database using Semantically Secure Encryption. Cryptology ePrint Archive, Report 2016/591, 2016. http://eprint.iacr.org/2016/591.

[PBP19]   Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An Encrypted Database using Semantically Secure Encryption. In *VLDB*, 2019.

[PCHF18]   Alex Poms, William Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient Video Analysis at Scale. In *SIGGRAPH*, 2018.

[Pen]   PencilBlue. https://goo.gl/SS4biS.

[Per14]   Nicole Perlroth. Security Experts Expect 'Shellshock' Software Bug in Bash to Be Significant, 2014. https://www.nytimes.com/2014/09/26/technology/security-experts-expect-shellshock-software-bug-to-be-significant.html.

[PHJ+16]   Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *OSDI*, 2016.

[PJ20]   Raluca Ada Popa and Anila Joshi. 3 ways to train a secure machine learning model. 2020. https://www.ericsson.com/en/blog/2020/2/training-a-machine-learning-model.

[PKV+14]   Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind Seer: A scalable private DBMS. In *IEEE S&P*, 2014.

[PKY+20]   Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. Cryptology ePrint Archive, Report 2020/1350, 2020. https://eprint.iacr.org/2020/1350.

[PKY+21]   Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. Senate: A Maliciously Secure MPC Platform for Collaborative Analytics. In *USENIX Security*, 2021.

[PLD+11]   Bryan Parno, Jay Lorch, John Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical State Continuity for Protected Modules. In *IEEE S&P*, 2011.

[PLPR18]   Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *NSDI*, 2018.

[PLZ13]   Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. An Ideal-Security Protocol for Order-Preserving Encoding. In *IEEE S&P*, 2013.

[PNH17]   Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. DStress: Efficient Differentially Private Computations on Distributed Data. In *EuroSys*, 2017.

[Pop14]   Raluca Ada Popa. *Building Practical Systems that Compute on Encrypted Data*. PhD thesis, MIT, 2014.

[Pou08]   Kevin Poulsen. Five IRS employees charged with snooping on tax returns, 2008. https://www.wired.com/2008/05/five-irs-employ/.

[Pri]   Privacy Rights Clearinghouse. Chronology of Data Breaches. http://www.privacyrights.org/data-breach.

[PRZB11]   Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, 2011.

[PSTY19]   Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient Circuit-Based PSI with Linear Communication. In *EUROCRYPT*, 2019.

[PVC18]   Christian Priebe, Kapil Vasawani, and Manuel Costa. EnclaveDB: A Secure Database Using SGX. In *IEEE S&P*, 2018.

[Ras11]   Fahmida Y. Rashid. Salesforce.com Acquires SaaS Encryption Provider Navajo Systems, 2011. https://goo.gl/MKiF2b.

[RBC13]   Joel Reardon, David Basin, and Srdjan Capkun. SoK: Secure Data Deletion. In *IEEE S&P*, 2013.

[RDGF16]   Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *CVPR*, 2016.

[Red]   Redux. https://goo.gl/AWZy6z.

[RHH14]   Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *IEEE S&P*, 2014.

[Riz12]   Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *ATC*, 2012.

[RLT15]   Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security*, 2015.

[RMR+21]   Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CROSSTALK: Speculative Data Leaks Across Cores Are Real. In *IEEE S&P*, 2021.

[Roe99]   Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *LISA*, 1999.

[RP66]   Azriel Rosenfeld and John L. Pfaltz. Sequential Operations in Digital Picture Processing. *J. ACM*, 1966.

[RRK18]   Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable Provably-secure Deep Learning. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018.

[RWT+18]   M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *AsiaCCS*, 2018.

[SA85]   Satoshi Suzuki and Keiichi Abe. Topological Structural Analysis of Digitized Binary Images by Border Following. *Comput. Vis. Graph. Image Proc.*, 1985.

[Sca]   SCALE-MAMBA Framework. https://homes.esat.kuleuven.be/~nsmart/SCALE/.

[SCF+15]   Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE S&P*, 2015.

[SCNS16]   Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *AsiaCCS*, 2016.

[Sec]   Secure XGBoost. https://github.com/mc2-project/securexgboost.

[Sel]   Selenium. http://www.seleniumhq.org/.

[SGF18]   Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *NDSS*, 2018.

[Sha]   ShareLaTeX. https://www.sharelatex.com/.

[SHS+12]   Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *SIGCOMM*, 2012.

[SKKG16]   Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV States by Using SGX. In *SDN-NFV Security*, 2016.

[Sky]   Skyhigh. Skyhigh Security Cloud. https://www.skyhighnetworks.com/.

[SLKP17]   Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[SLM+19]   Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[SLPR15]   Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *SIGCOMM*, 2015.

[Sno]   Snort Community Rulesets. https://www.snort.org/downloads.

[SP16]   Raoul Strackx and Frank Piessens. Ariadne: A Minimal Approach to State Continuity. In *USENIX Security*, 2016.

[SPS14]   Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, 2014.

[SRS17]   Rohit Sinha, Sriram Rajamani, and Sanjit A. Seshia. A Compiler and Verifier for Page Access Oblivious Computation. In *FSE*, 2017.

[SS15]   Reza Shokri and Vitaly Shmatikov. Privacy-Preserving Deep Learning. In *CCS*, 2015.

[SST17]   Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *USENIX Security*, 2017.

[STTS17] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *NDSS*, 2017.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *CCS*, 2013.

[SvHA+19] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. Secure multiparty PageRank algorithm for collaborative fraud detection. In *FC*, 2019.

[SWG+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[SWP00] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *IEEE S&P*, 2000.

[TB19] Florian Tramer and Dan Boneh. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *ICLR*, 2019.

[Tel] The Telegraph. How retailers make shoppers stand out from the crowd. https://www.telegraph.co.uk/business/open-economy/how-retailers-make-shoppers-stand-out/.

[TGS+18] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference. *arXiv:1810.00602*, 2018.

[The] The Fast Data Project. Vector packet processing. https://www.fd.io/technology.

[TKG+18] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *SOSR*, 2018.

[TKMZ13] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing Analytical Queries over Encrypted Data. *PVLDB*, 6(5):289–300, 2013.

[TPCa] TPC-C Transation Processing Benchmark. http://www.tpc.org/tpcc/.

[TPCb] TPC-H Benchmark. http://www.tpc.org/tpch/.

[TPV17] Chia-che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *ATC*, 2017.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX Security*, 2017.

[TZJ⁺16] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security*, 2016.

[UNC] UNCAP: Ubiquitous iNteropable Care for Ageing People. http://www.uncap.eu/.

[VBMS⁺20] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE S&P*, 2020.

[Ver] Verkada. https://verkada.com.

[Vis] Vision Zero. https://visionzeronetwork.org.

[Viv] Vivotek. Smart Stream II. https://www.vivotek.com/website/smart-stream-ii/.

[VP9] VP9 Codec. https://www.webmproject.org/vp9/.

[VSG⁺19] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: Secure Multi-Party Computation on Big Data. In *EuroSys*, 2019.

[vSMK⁺20] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. https://cacheoutattack.com/, 2020.

[vSMO⁺19] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE S&P*, 2019.

[VVB18] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In *OSDI*, 2018.

[WCP⁺17] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[WGSW18] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating Timing Side-Channel Leaks Using Program Repair. In *ISSTA*, 2018.

[WR19] Ke Coby Wang and Michael K. Reiter. How to end password reuse on the web. In *NDSS*, 2019.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-Scale Secure Multiparty Computation. In *CCS*, 2017.

[XCP15]   Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE S&P*, 2015.

[XYH12]   Liangliang Xiao, I-Ling Yen, and Dung T. Huynh. Extending Order Preserving Encryption for Multi-User Systems. Cryptology ePrint Archive, Report 2012/192, 2012. http://eprint.iacr.org/2012/192.

[Yah]   Yahoo! Transparency Report. https://transparency.yahoo.com/.

[Yao82]   Andrew C. Yao. Protocols for secure computations. In *Symposium on Foundations of Computer Science (SFCS)*, 1982.

[Yao86]   Andrew C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.

[YGH16]   Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. In *CHES*, 2016.

[YKK+11]   Dae Yum, Duk Kim, Jin Kim, Pil Lee, and Sung Hong. Order-Preserving Encryption for Non-uniformly Distributed Plaintexts. In *WISA*, 2011.

[YWLW16]   Xingliang Yuan, Xinyu Wang, Jianxiong Lin, and Cong Wang. Privacy-preserving Deep Packet Inspection in Outsourced Middleboxes. In *INFOCOM*, 2016.

[ZAB+17]   Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Victor Bahl, and Michael Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *NSDI*, 2017.

[ZAM11]   Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive Mitigation of Timing Channels in Interactive Systems. In *CCS*, 2011.

[ZDB+17]   Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*, 2017.

[Zet10]   Kim Zetter. Ex-Googler allegedly spied on user emails, chats, 2010. https://www.wired.com/2010/09/google-spy/.

[Ziv04]   Zoran Zivkovic. Improved Adaptive Gaussian Mixture Model for Background Subtraction. In *International Conference on Pattern Recognition (ICPR)*, 2004.

[ZJR+18]   Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. AWStream: Adaptive Wide-area Streaming Analytics. In *SIGCOMM*, 2018.

[ZKP15]   Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *CCS*, 2015.

[ZLL⁺19]   En Zhang, Feng-Hao Liu, Qiqi Lai, Ganggang Jin, and Yu Li. Efficient Multi-Party Private Set Intersection Against Malicious Adversaries. In *CCSW*, 2019.

[ZPGS19]   Wenting Zheng, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. Helen: Maliciously Secure Coopetitive Learning for Linear Models. In *IEEE S&P*, 2019.

[ZRE15]   Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates. In *EUROCRYPT*, 2015.

[ZSB13]   Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: A General-purpose Compiler for Private Distributed Computation. In *CCS*, 2013.

[Zsc]   Zscaler. https://www.zscaler.com/.

[ZvdH06]   Zoran Zivkovic and Ferdinand van der Heijden. Efficient Adaptive Density Estimation per Image Pixel for the Task of Background Subtraction. *Pattern Recognition Letters*, 2006.

Appendix A

Joins over Multisets in Senate

We now describe an extension to the set intersection circuit that enables the evaluation of joins on multisets. Our extension requires Senate to know the *multiplicity c* of values in the joined columns. That is, each value in the column can have no more than c occurrences.

Consider an example where parties P_1 and P_2 wish to compute a join over their respective columns T_1 and T_2 each of size n. Let c_1 and c_2 be the multiplicity of values in the two columns. Then, one way to express the join as a set intersection is to encode the values in a columns based on the other column's multiplicity, as proposed by prior work [KS05, NH12]. Suppose an element a occurs $k_1 \leq c_1$ times in T_1, and $k_2 \leq c_2$ times in T_2. Then, each instance a_i of the element a (for $1 \leq i \leq k_1$) is replaced with the set of elements $\{a||i||j\}$ for $1 \leq j \leq c_2$. Similarly, each instance a_i of an element a in T_2 is replaced with the set of elements $\{a||j||i\}$ for $1 \leq j \leq c_1$. The intersection thus contains $k_1 \times k_2$ instances of a, as required by the join operation. As a consequence, however, the size of P_1's input set increases from n to $c_2 \times n$, and the size of P_2's input set increases to $c_1 \times n$. In Senate, this affects the size of the 2-SI circuit, which increases as a result. .

Senate therefore employs an alternate approach. We note that broadly, there are two possible query types: (i) those in which to the input to the join operation is the joined column alone; or (ii) the input contains additional columns. To handle the former, for every distinct element a each party replaces its instances a_i with a single tuple (a, k), where k is the number of occurrences of a. The 2-SI circuit is then computed over the lists of tuples instead of singleton elements, outputting values of the type (a, k_1, k_2) for elements in the intersection.

For the latter case where there are additional columns along with the joined column, the inputs can be represented as tuples of the form (a, b), where the first element corresponds to the column to be joined. In such situations, for every distinct element a, each party replaces its instances with a single tuple $(a, b_1, \ldots, b_c)$ instead, where c is the maximum multiplicity of the column. If a party only has $k < c$ instances of a, then the values $b_{k+1} \ldots b_c$ are zeros. The 2-SI circuit then outputs values of the type $(a, b_1^1, \ldots, b_c^1, b_1^2, \ldots, b_c^2)$, which can be demultiplexed to obtain the result of the join.

Appendix B

Invertibility of SQL Operators in Senate

To verify the results of intermediate operations in the query tree, Senate's compiler needs to deduce the range of values that the output of a node must satisfy. As the query tree is executed, constraints implied by operations lower in the tree accumulate upwards and may impact the outputs of later operations. We now show how Senate's compiler deduces the constraints imposed by various relational operations (*i.e.*, what needs to be verified). Then, we show the invertibility of relational operations given these constraints. This ensures that the resulting tree is admissible, and satisfies the requirements of Senate's MPC protocol.

For simplicity, Senate considers operations that constrain the output in one of three ways: (i) for a column, each element a_i in the output can be constrained to belong to one or more ranges of the type: $n_1 \leq a_i \leq n_2$, where n_1 and n_2 are constants; (ii) the records in the output are ordered by a single column (*e.g.*, due to a sort); or (iii) the elements in the column are distinct (*e.g.*, primary keys). If an operation in the tree results in an output that violates these constraints, then it is marked as unverifiable. Note that equality is a special case of the range constraint. Further, we can check whether or not a column has distinct elements by sorting them first and then checking that adjacent elements are different (*i.e.*, by reducing it to an ordering constraint).

Filters. For filtering operators σ_f that filter a column a based on the predicate f, we need to verify that each record in the output relation satisfies the applied function f (which can be of type $=$, $<$, $>$). All original constraints on the input columns are preserved and propagated to the output; in addition, an extra range constraint is added in accordance with f.

The invertibility of filtering operations is straightforward. Let C be the set of constraints on the input; then, the set of constraints induced by σ_f is $C' = C \cup f(a)$. Given an output R' that satisfies all constraints in C', one can generate an input $R \supseteq R'$ that forms a valid pre-image and satisfy all constraints in C. In particular, the relation R' is itself a valid pre-image.

Joins. We consider equijoin operations $\bowtie$ over columns that are sets. In this case, all range constraints on the input columns are preserved in the output, and the operation only requires that the joined column in the output is also a set. If some column in the input contains an ordering constraint, then the constraint is discarded (*i.e.*, it doesn't propagate upwards to the next node) because join operations make no guarantees towards preserving order. Joins that are not based

on equality testing are marked as unverifiable; this is because such joins constrain the values of a column to be dependent on the values in other columns, violating our requirements above.

As regards invertibility, let C' be the constraints after the join operation, as described above. Given an output relation R' that satisfies C' and the constraint sets C_1, C_2 on relations R_1, R_2 respectively, one can craft relations R'_1, R'_2 that satisfy constraints sets C_1, C_2 as follows: A row in R_3 can be separated to two parts, one with columns of relation R_1 are added to relation R'_1 and the other with columns of relation R_2 are added to relation R'_2. The values under the joined column are duplicated to these two parts. The relations R'_1, R'_2 clearly satisfy constraints sets C_1, C_2, respectively.

Order-by. For order-by operations τ over the column a, if the input contains ordering constraints on any other column, then the operation is marked as unverifiable. Otherwise, we need to verify that the column a in the output relation is sorted. All range / distinctness constraints on the input columns are preserved and propagated upwards.

It is easy to see the invertibility of order-by operations. In particular, the output itself is a valid input.

Group-by aggregates. For group-by operations γ that group by a column c while performing a single aggregate Σ on the column a, suppose the output columns are (c', a'), where c' represents the groups and a' the aggregates per group. Then, all range constrains on c are preserved in c'; additionally, c' now includes a distinctness constraint. As regards a', if Σ is max or min, then the range constraints on a apply to a' as well. If Σ is count, then a' is unconstrained. If Σ is sum, then no constraints apply on a' only if a is also unconstrained; however, if a has a range constraint, then the operation is marked as unverifiable. This is because in the presence of range constraints, it may be hard to deduce the possible values the sum can take, requiring a constraint solver in many cases.

Given a relation R' output from a group-by node which satisfies the above constraints, one can find a pre-image relation R as follows. Since columns other than a are not constraints, ignore them (we could set any value for them). Then, for every row (c', a') in R', add rows to R such that the values in a are at most a' if the aggregate is a max, or the values in a are at least a' if the aggregate is a min. If the aggregate is a count, then the node can be easily inverted by generating the requisite number of rows for every value of c.

Appendix C

Security Proofs and Pseudocode for Visor

We now provide detailed pseudocode along with proofs of security for our algorithms. We start by providing a formal definition of data-obliviousness.

Let $\mathsf{trace}_{\mathcal{A}}(x)$ be the trace of observations that the adversary can make during an execution of an algorithm $\mathcal{A}$, when given some input x, *i.e.*, the sequence of accessed memory addresses, along with the (encrypted) data that is read or written to the addresses. To prove that the algorithm is data-oblivious, we show that there exists a simulator program that can produce a trace T indistinguishable from $\mathsf{trace}_{\mathcal{A}}(x)$, when given *only* the public parameters for the algorithm, and regardless of the value of x. Since T does not depend on any private data, the indistinguishability of T and $\mathsf{trace}_{\mathcal{A}}$ implies that the latter leaks no information about the private data to the adversary, and only depends on the public parameters. The following definition captures the definition formally.

Definition 7 (Data-obliviousness). *We say than an algorithm $\mathcal{A}$ is data-oblivious if there exists a polynomial-time simulator* Sim *such that for any input x*
$$\mathsf{trace}_{\mathcal{A}}(x) \equiv \mathsf{Sim}(\mathcal{L}(\mathcal{A}))$$
where $\mathcal{L}(\mathcal{A})$ is the leakage function and represents the public parameters of $\mathcal{A}$.

We now prove the security of each of our algorithms with respect to Definition 7 in the following subsections. Figure C.1 summarizes the public parameters across Visor oblivious vision modules that are leaked to the attacker.

C.1 Oblivious video decoding

Algorithm 1 provides detailed pseudocode for oblivious decoding the bitstream into pixel coefficients during video decoding, as described in Section 6.6.2. We first explain the pseudocode in more detail, following the high-level description of Section 6.6.2.

In our implementation, we model the prefix tree as a finite state machine (FSM). While traversing the tree, we decode a single bit at each state (*i.e.*, each node in the tree) using the ENTROPYDECODE subroutine. This subroutine takes in a pointer `ptr` to the bitstream, and decodes a single bit from

Component	Public parameters
Video decoding	(*i*) Metadata of video stream (format, frame rate, resolution); (*ii*) Number of bits used to encode each (padded) row of blocks; (*iii*) Maximum number of bits encoded per 2-byte chunk.
Background sub.	–
Bounding box det.	(*i*) Maximum number of objects per image; (*ii*) Maximum number of different labels that can be assigned to pixels (an object consists of all labels that are adjacent to each other).
Object cropping	Upper bounds on object dimensions.
Object tracking	(*i*) An upper bound on the intermediate number of features; (*ii*) An upper bound on the total number of features.
CNN Inference	CNN architecture.
Overall	Modules and algorithms used in the pipeline.

Figure C.1: Summary of public parameters in Visor's oblivious vision modules observable by the attacker. These consist of the input parameters provided to Visor, along with information leaked by Visor (such as frame rate and resolution).

the bitstream via arithmetic operations. If no more bits can be decoded at the current position, it outputs null; otherwise, it outputs the decoded bit 0 or 1.

To enable decoding and traversal, each state S (*i.e.*, each node in the tree) is a structure containing four fields: $(\text{prob}, \text{next}_0, \text{next}_1, \text{type})$. Here, prob is the probability that the bit to be decoded at S is 0 (as defined in the VP8 specifications [BKQ+11]); and next_0 and next_1 are the indices of the next state based on whether the decoded bit is a 0 or 1. Some states in the FSM are end states, *i.e.*, states that complete reconstructing a data value; for these states, type is set to 'end'. States that are not end states (*i.e.*, decode intermediate bits) have type set to 'mid'. The FSM also contains a dummy state S_{dummy} that performs dummy bit decodes by invoking the entropy decoder with isDummy set to true; for the dummy state, type is set to 'dummy'.

Next, we represent the FSM as an array—Nodes in Algorithm 1. We set Nodes[0] to be S_{dummy}, and Nodes[1] to be the starting state. This enables us to implement transitions to any state S_j by simply fetching the state at index j from the array using the oaccess primitive. As a result, the current state of the FSM remains hidden across transitions. Each transition passes four items of information to the next state: (i) the state that made the transition S_{parent}; (ii) an integer pos that denotes the position in the bitstream of the current bit being decoded; (iii) the (partially) constructed data value data, and (iv) a counter ctr that counts the number of bits decoded at each position. Note that the structure of the prefix tree (and hence the array Nodes) is public information since it is constructed per the VP8 specifications [BKQ+11].

Theorem 3. *The bitstream decoding algorithm in Algorithm 1 is data-oblivious per Definition 7, with public parameters N_{bits}, N_{chunk}, and the size of the prefix tree array Nodes (which is a known constant).*

Algorithm 1 Bitstream decoding

1: **Constants:** Upper bound on number of encoded bits per 2-byte chunk in bitstream N_{chunk}; total number of bits in bitstream N_{bits}; array representation of the prefix tree for decoding `Nodes`

2: **Globals:** The data value being decoded `data`; counter `ctr` that counts the number of bits decoded per chunk in the bitstream

3: **Input:** Bitstream B

4: **procedure** DECODEBITSTREAM(B)

5:     $\text{ptr} = B.\text{start}$

6:     $S = \text{Nodes}[1]$

7:     $S_{\text{parent}} = \text{null}, \text{data} = 0, \text{ctr} = 0, \text{pos} = 0$

8:     $O = [\,]$

9:     **while** $\text{ptr} < B.\text{start} + N_{\text{bits}}$ **do**

10:         isDummy = (S.type == 'dummy')

11:         $b = \text{ENTROPYDECODE}(\text{isDummy}, \text{ptr}, S.\text{prob})$

12:         isDummy = (b == null)

13:         $\text{data} = \text{UPDATEDATA}(\text{isDummy}, \text{data}, b)$

14:         $\text{pos} += 1$

15:         $\text{ctr} = \text{oassign}(\text{ctr} == N_{\text{chunk}}, 0, \text{ctr} + 1)$

16:         isEnd = (S.type == 'end')

17:         $o_1 = \text{oassign}(\text{isEnd}, \text{pos}, 0)$

18:         $o_2 = \text{oassign}(\text{isEnd}, \text{data}, \text{null})$

19:         $\text{parent} = \text{oassign}(\neg\text{isDummy}, S.\text{index}, S_{\text{parent}}.\text{index})$

20:         $\text{next} = \text{oassign}(b == 0, S.\text{next}_0, S.\text{index})$

21:         $\text{next} = \text{oassign}(b == 1, S.\text{next}_1, \text{next})$

22:         $\text{next} = \text{oassign}(\text{isDummy}, 0, \text{next})$

23:         $\text{next} = \text{oassign}(\text{isDummy} \wedge$
                $\text{ctr} == N_{\text{chunk}}, \text{parent}, \text{next})$

24:         $O.\text{APPEND}((o_1, o_2))$

25:         $S_{\text{parent}} = \text{oaccess}(\text{Nodes}, \text{parent})$

26:         $S = \text{oaccess}(\text{Nodes}, \text{next})$

27:         $\text{ptr} = \text{oassign}(\text{ctr} == N_{\text{chunk}}, \text{ptr} + 2, \text{ptr})$

28:     **end while**

29:     $\text{osort}(O)$

30:     **return** O

31: **end procedure**

Proof. The simulator starts be generating a random bitstream B of length N_{bits}, and then simply executes Algorithm 1. It outputs the trace produced by the algorithm.

Lines 5–8 have fixed access patterns.

The loop in line 9 runs a fixed number of times: `ctr` increments by 1 in each run of the loop on line 15 until it becomes equal to N_{chunk}, at which point the loop variable `ptr` is incremented by 2 in

line 27. Thus, the loop makes exactly $N_{\texttt{bits}} \times N_{\texttt{chunk}}/2$ iterations.

Within the loop, line 10 has fixed access patterns. In line 11, the function ENTROPYDECODE dereferences `ptr` and decodes a single bit from the dereferenced value using simple arithmetic operations; if `isDummy` is true it performs dummy operations instead, using `oassign`. Its access patterns thus only depend on the location pointed to by `ptr` within the bitstream B, and not the contents of the bitstream. Further, the value of the loop variable `ptr` is incremented per a fixed schedule (as described above), and thus only depends on the value of $N_{\texttt{bits}}$.

Line 12 has fixed access patterns. In line 13, the function UPDATEDATA updates the value of `data` with b using arithmetic operations implemented using `oassign`. Its access patterns are thus independent of the `data` or b.

Lines 14–24 have fixed access patterns. The access patterns of lines 25–26 only depend on the length of `Nodes`, which is fixed and public. Line 27 also has fixed access patterns.

Finally, line 29 uses the `osort` primitive to sort the array O; its access patterns thus depend on the length of O. Since a single tuple is appended to O per iteration of the loop, the length of O is equal to the number of iterations, which only depends on $N_{\texttt{bits}}$ and $N_{\texttt{chunk}}$ as described above.

Thus, the trace produced by the algorithm can be simulated only using the values of $N_{\texttt{bits}}$ and $N_{\texttt{chunk}}$. □

C.2 Oblivious image processing

In this section, we provide pseudocode along with proofs of security for the image processing algorithms described in Section 6.7. For each algorithm, we first briefly describe its pseudocode, and then prove its security with respect to Definition 7.

C.2.1 Background subtraction

As described in Section 6.7.1, the background subtraction algorithm (Algorithm 2) maintains a mixture of M Gaussian components per pixel.

Let $\vec{x}^{(t)}$ denote the value of a pixel in RGB at time t. The algorithm uses the value of the pixel to update each Gaussian component via a set of arithmetic operations (lines 5–8 in the pseudocode) along with their weights π_m such that, over time, components that represent background values for the pixel come to have larger weights, while foreground values are represented by components having smaller weights.

Then, it compute the distance of the sample from each of the M components. If no component is sufficiently close, it adds a new component, increments M, and if the new $M > M_{\texttt{max}}$, discards the component with the smallest weight π_m (lines 9–21).

Finally, it uses the B largest components by weight to determine whether the pixel is part of the background (lines 22–30). Note that $M_{\texttt{max}}$ and B are algorithmic constants, independent of the input video streams.

Theorem 4. *The background subtraction algorithm in Algorithm 2 is data-oblivious per Definition 7, with public parameters $M_{\texttt{max}}$ and B (which are known constants).*

Algorithm 2 Background subtraction

1: **Constants:** Maximum number of Gaussian components M_{max}, number of components to count towards background decision B; threshold measures δ_{thr}, c_f, and c_{thr}
2: **Globals:** Actual number of Gaussian components M, array of Gaussian components GMM of size M_{max}
3: **Input:** Pixel x
4: **procedure** BACKGROUNDSUBTRACTION(x)
5:      **for** $m = 1$ **to** M_{max} **do**
6:          isDummy $= (m > M)$
7:          UPDATEGAUSSIAN(isDummy, GMM[m], x)
8:      **end for**

9:      SORTBYWEIGHT(GMM)
10:      isClose $=$ false
11:      **for** $m = 1$ **to** M_{max} **do**
12:          $\delta =$ GETDISTANCE(GMM[m], x)
13:          isClose $=$ isClose $\vee (\delta > \delta_{\text{thr}})$
14:      **end for**
15:      $M =$ oassign(isClose $\wedge (M < M_{\text{max}}), M + 1, M)$
16:      $G =$ GENERATEGAUSSIAN()
17:      GMM[$M_{\text{max}} - 1$] $=$ oassign(isClose, GMM[$M_{\text{max}} - 1$], G)
18:      **for** $m = M_{\text{max-1}}$ **to** 1 **do**
19:          toSwap $=$ (GMM[m].π $<$ GMM[$m + 1$].π)
20:          GMM[m] $=$ oassign(toSwap, GMM[$m + 1$], GMM[m])
21:      **end for**

22:      $c = 0$
23:      $p = 0$
24:      toInclude $=$ true
25:      **for** $m = 1$ **to** B **do**
26:          $c = c +$ GMM[m].π
27:          $p =$ oassign(toInclude, $p + c, p$)
28:          toInclude $=$ oassign($c > c_f$, false, toInclude)
29:      **end for**
30:      **return** $p > c_{\text{thr}}$
31: **end procedure**

Proof. The simulator chooses a random pixel value x and simply runs the algorithm. It outputs the trace produced by the algorithm.

Lines 6–7 are executed exactly M_{max} times. Here, the loop variable m is public information. Line 6 has fixed access patterns. The function UPDATEGAUSSIAN performs a set of arithmetic operations independent of the value of x, via oassign operations using the condition value isDummy. The function updates The access patterns of line 7 therefore only depend on m.

SORTBYWEIGHT in line 9 sorts the GMM array using the oblivious sorting primitive osort.

Hence, the access patterns of this step only depend on the length of GMM, which is $M_{\max}$.

Lines 12–13 are executed exactly $M_{\max}$ times. The function GETDISTANCE computes the distance of x from GMM$[m]$ via arithmetic operations, independent of the value of x. Thus, the access patterns of line 12 thus depends only on the loop variable m. Line 13 has fixed access patterns. Lines 15–16 have fixed access patterns, and the access patterns of line 17 depends only on $M_{\max}$. Lines 19–20 are executed exactly $M_{\max} - 1$ times. The access patterns of both lines depend only on the loop variable m.

Lines 22–24 have fixed access patterns. Lines 26–28 are executed exactly $M_{\max}$ times. The access patterns of line 26 depend only on the loop variable m; lines 27 and 28 have fixed access patterns.

Thus, the trace produced by the simulator is indistinguishable from the trace produced by a real run. □

C.2.2 Object detection

Algorithm 3 describes our algorithm for detecting bounding boxes of objects in an input image. The algorithm maintains a list L of tuples of the form (`parent`, `bbox`), where each tuple corresponds to a distinct "label" that will eventually be mapped to each blob. Initially, the list L is empty. The `parent` field identifies other labels that are connected to the tuple's label (explained shortly), and the `bbox` field maintains the coordinates of the bounding box of the label (or blob).

The algorithm first scans the image row-wise (lines 6–22). Whenever a white pixel is detected, the algorithm checks if any of its neighbors scanned thus far were also white (*i.e.*, pixel to the left and the three pixels above). In case at least one neighbor is white, the pixel is assigned the label of the neighbor with the smallest numerical value, $l_{\min}$. The algorithm records that all white neighbors are connected, by setting the `parent` fields for each neighboring label to $l_{\min}$ and updating the `bbox` field for $l_{\min}$. In case no neighbor is white, the pixel is assigned a new label, and a new entry is added to the list L, with its `parent` set to the label itself and `bbox` as the coordinates of the current pixel.

Next, the algorithm *merges* the bounding boxes of all connected labels into a single bounding box (lines 23–35). Specifically, for every label l in L, the algorithm first obtains the `parent` label of l (say l_{par}), and then updates the `bbox` of l_{par} to include the `bbox` of l. It repeats the process recursively with l_{par}, until it reaches a root label l_{root} whose `parent` value is the label itself. The process repeats for all labels in L, until only the root labels are left behind. Each root label corresponds to a distinct object in the frame.

Theorem 5. *The bounding box detection algorithm in Algorithm 3 is data-oblivious, with public parameters N, and the height and width of the input frame.*

Proof. The simulator generates a random frame F of the given height and width and runs the algorithm. It outputs the trace produced by the algorithm.

The access patterns of line 4 depends only on N. Line 5 has fixed access patterns.

The loops (lines 6–22) are executed a fixed number of times, equal to the height and width of the frame. The access patterns of line 8 depend only on the loop variables i and j, which are public

---

Algorithm 3 Bounding box detection

 1: **Constants:** Maximum number of labels N
 2: **Input:** Frame F
 3: **procedure** BOUNDINGBOXDETECTION(F)
 4:     Initialize list L of N tuples of type (parent, bbox),
    with $L[i]$.parent $= i$
 5:     ctr $= 1$
 6:     **for** $i = 1$ **to** F.height **do**
 7:         **for** $j = 1$ **to** F.width **do**
 8:             $p = F[i][j]$
 9:             isWhite $= (p \neq 0)$
10:             $(p_1, p_2, p_3, p_4) = $ GETNEIGHBORS(F, i, j)
11:             $(l_1, l_2, l_3, l_4) = $ GETNEIGHBORLABELS(F, i, j)
12:             isNew $= (p_1 == p_2 == p_3 == p_4 == 0) \wedge$ isWhite
13:             $l_{\min} = $ GETMINLABEL(l_1, l_2, l_3, l_4)
14:             $l_{\min} = $ oassign(isNew, ctr, $l_{\min}$)
15:             ctr $= $ oassign($\neg$isNew, ctr $+ 1$, ctr)
16:             **for** each label l in $\{l_1, l_2, l_3, l_4\}$ **do**
17:                 UPDATEPARENT($L, \neg$isNew, $l, l_{\min}$)
18:             **end for**
19:             UPDATEBBOX($L,$ isWhite, $l_{\min}, i, j$)
20:             SETLABEL($F, i, j, l_{\min}$)
21:         **end for**
22:     **end for**

23:     **for** $i = 1$ **to** N **do**
24:         par $= L[i]$.parent
25:         toMerge $= $ (par $< i$)
26:         **for** $j = i$ **to** 1 **do**
27:             $L[i]$.parent $= $ oassign(toMerge $\wedge$ (par $== j$),
                $L[j]$.parent, $L[i]$.parent)
28:         **end for**
29:     **end for**
30:     **for** $i = 1$ **to** N **do**
31:         **for** $j = 1$ **to** N **do**
32:             toMerge $= (L[j]$.parent $== i$)
33:             MERGEBBOX(toMerge, $L[i]$.bbox, $L[j]$.bbox)
34:         **end for**
35:     **end for**
36:     **return** L
37: **end procedure**

---

information. Line 9 has fixed access patterns. In line 10, the function GETNEIGHBORS returns

---

Algorithm 4 Object cropping

1: **Constants:** Upper bounds on object dimensions `height`, `width`
2: **Input:** Frame F, bounding box coordinates bbox
3: **procedure** CROPOBJECT(F, bbox)
4:     Initialize an empty buffer buf with width $= F$.width and height $=$ `height`
5:     **for** $i = 1$ **to** F.height **do**
6:         cond $= (i ==$ bbox.top$)$
7:         COPYROWS(cond, i, F, buf)
8:     **end for**

9:     Initialize an empty buffer obj with width $=$ `width` and height $=$ `height`
10:     **for** $i = 1$ **to** F.width **do**
11:         cond $= (i ==$ bbox.left$)$
12:         COPYCOLS(cond, i, buf, obj)
13:     **end for**
14: **end procedure**

---

the four pixels neighboring the input coordinates (i and j), and its access patterns thus depend only on i and j. Similarly in line 11, GETNEIGHBORLABELS looks up the labels assigned to the neighboring pixels, and thus has access patterns that only depend on i and j. Line 12 has fixed access patterns. In line 13, GETMINLABEL selects the minimum of the input values using `oassign` operations, and thus has fixed access patterns. Lines 14–15 have fixed access patterns. In line 17, UPDATEPARENT uses `oaccess` combined with `oassign` to update $L[l]$.parent to l_{min}; it thus has access patterns that only depend on the length N of the array L. In line 18, UPDATEBBOX similarly uses `oaccess` combined with `oassign` to update $L[l_{min}]$.bbox with the current coordinates i and j; its access patterns therefore only depend on L's length N. In line 19, SETLABEL sets the label of the pixel at $F[i][j]$ to l_{min}; its access patterns depend only on the loop variables i and j.

Lines 24–28 are run N times. The access patterns of line 24 depend only on the loop variable i. Line 25 has fixed access patterns. Line 27 is executed i times, which is public information; also, the access patterns of this line only depend on the loop variables i and j.

Lines 32–33 are run N^2 times. The access patterns of line 32 only depend on the loop variable j. In line 33, the function MERGEBBOX uses `oassign` operations to update $L[i]$.bbox with $L[j]$.bbox; it therefore has fixed access patterns.

Thus, the trace produced by the simulator is indistinguishable from the trace produced by a real run of the algorithm. $\square$

C.2.3   Object cropping

Algorithms 4 and 5 together describe Visor's oblivious cropping algorithm. Visor crops out images of a fixed upper bounded size using Algorithm 4, and then scales up the ROI within the cropped image using Algorithm 5 (as described in Section 6.7.3). The pseudocode is self-explanatory.

Algorithm 5 Object resizing

 1: **Input:** Object buffer O, bounding box coordinates bbox
 2: **procedure** RESIZEOBJECT(O, bbox)
 3:     RESIZEHORIZONTALLY(O, bbox, false)
 4:     TRANSPOSE(O)
 5:     RESIZEHORIZONTALLY(O, bbox, true)
 6:     TRANSPOSE(O)
 7: **end procedure**

 8: **procedure** RESIZEHORIZONTALLY(O, bbox)
 9:     **for** $i = 1$ **to** O.height **do**
10:         **for** $j = 1$ **to** O.width **do**
11:             $p = $ PIXELOFINTEREST(j, bbox)
12:             $a = $ oaccess(O[i], p)
13:             $b = $ oaccess(O[i], $p + 1$)
14:             $O[i][j] = $ LINEARINTERPOLATE(a, b)
15:         **end for**
16:     **end for**
17: **end procedure**

Theorem 6. *The object cropping algorithm in Algorithm 4 is data-oblivious, with public parameters equal to the dimensions of the input frame, and the upper bounds on the object dimensions* height *and* width.

Proof. The simulator generates a random frame of the given dimensions, along with a bounding box bbox with random coordinates. It then runs the algorithm, and outputs the produced trace.

The access patterns of line 4 depend only the frame's width, and the parameter width, both of which are known to the simulator. Lines 6–7 run a fixed number of times, equal to the height of the frame. Line 6 has fixed access patterns. In line 7, COPYROWS uses oassign to copy pixels from F into buf; its access patterns thus only depend on the loop variable i, the width of the frame, and the parameter height.

The access patterns of line 9 depend only the parameters width and height. Lines 11–12 run a fixed number of times, equal to the width of the frame. Line 11 has fixed access patterns. In line 12, COPYCOLS uses oassign to copy pixels from buf into obj; its access patterns thus only depend on the loop variable i, and the parameters height and width.

Thus, the trace produced by the simulator is indistinguishable from the trace produced by a real run of the algorithm. □

Theorem 7. *The object resizing algorithm in Algorithm 5 is data-oblivious, with public parameters equal to the dimensions of the input object O.*

Proof. The simulator generates a random object buffer with the given dimensions, along with a bounding box bbox with random coordinates. It then runs the algorithm, and outputs the produced trace.

The function Transpose transposes the object buffer, and thus its access patterns only depend on the dimensions of *O*. The function ResizeHorizontally works as follows. The loops (lines 9–15) are executed a fixed number of times, equal to the dimensions of *O*. Line 11 computes the location of the pixels to be used for linearly interpolating the current pixel, using a set of arithmetic operations; it thus has fixed access patterns. Lines 12 and 13 have access patterns that only depend on the width of *O*. In line 14, LINEARINTERPOLATE linearly interpolates the current pixel's value using a set of arithmetic operations; the access patterns of this line thus depend only on the loop variables.

Thus, the trace produced by the simulator is indistinguishable from the trace produced by a real run of the algorithm. □

C.2.4 Object tracking

Algorithm 6 describes the feature detection phase of the object tracking. We omit a description of feature matching since it is oblivious by default.

The algorithm first creates a set of increasingly blurred versions of the input image (line 5). Then, it identifies a set of candidate *keypoints* in these blurred images, *i.e.*, pixels that are the maximum and minimum of all their neighbors (lines 6–14). This set of keypoints is further refined to identify those that are robust to changes in illumination (*i.e.*, have high intensity), or represent a "corner" (lines 15–18). Mathematically, these tests involve the computation of derivatives at the candidate point, and then a comparison of the results against a threshold. Candidates that fail these tests are discarded.

Finally, for each keypoint, the algorithm constructs a *feature descriptor*. It calculates the "orientation" of the pixels around the keypoint (within a 16×16 neighborhood) based on pixel differences, and then constructs a histogram over the computed values (lines 20–14). The histogram acts as the keypoint's descriptor.

Theorem 8. *The feature detection algorithm in Algorithm 6 is data-oblivious, with public parameters equal to the dimensions of the input image O, and upper bounds N_{temp} and N.*

Proof. The simulator generates a random image buffer with the given dimensions, and then runs the algorithm. It outputs the trace produced by the algorithm.

Line 5 performs Gaussian blurring operations on the input image *O*, which perform a convolution of the input image with a specified *kernel* (*i.e.*, a small matrix). The access patterns of these matrix multiplications are fixed, and independent of the values of the matrices.

The loop (lines 6–14) runs a fixed number of times, the value of which depends on the resolution of the input image, which is public. Line 7 fetches the neighbors of the current pixel; its access patterns are therefore dependent only on coordinates of the loop variable, which is public. Line 8 checks the value of the current pixel with the obtained nbrs using oassign operations, and thus

---

Algorithm 6 Feature detection

1: **Input:** Object buffer O, maximum number of candidate keypoints N_{temp}, maximum number of actual keypoints N
2: **procedure** DETECTFEATURES(O, N_{temp}, N)
3:     Initialize an empty list L of size N_{temp} for candidate keypoints, and a list H of size N for features of final keypoints
4:     ctr $= 0$
5:     images $=$ GETDIFFERENCEOFGAUSSIANS(O)
6:     **for** each pixel p in images **do**
7:         nbrs $=$ GETNEIGHBORS(p)
8:         isExtrema $=$ CHECKEXTREMA(p, nbrs)
9:         $k = (p, \text{nbrs})$
10:         **for** $i = 1$ to N_{temp} **do**
11:             $L[i] =$ oassign(isExtrema $\wedge\, i ==$ ctr, $k, L[i]$)
12:         **end for**
13:         ctr $=$ oassign(isExtrema $\wedge$ ctr $< N_{\text{temp}}$, ctr+1, ctr)
14:     **end for**

15:     **for** $i = 1$ to N_{temp} **do**
16:         isRobust $=$ CHECKROBUSTNESS($L[i]$)
17:         $L[i] =$ oassign(isRobust, $L[i]$, null)
18:     **end for**
19:     osort(L) such that non-null values move to the head of L

20:     **for** $i = 1$ to N **do**
21:         bbox $=$ CALCNEIGHBORHOODBBOX($L[i]$)
22:         roi $=$ CROPOBJECT(images, bbox)
23:         $H[i] =$ CALCORIENTATIONHIST($L[i]$, roi)
24:     **end for**
25:     **return** H
26: **end procedure**

---

has fixed access patterns, independent of the values. Line 9 has fixed access patterns. The loop in lines 10–12 executes a fixed N_{temp} number of times. The access patterns of line 11 depend only on the public loop variable. Line 13 has fixed access patterns.

The loop in lines 15–18 executes a fixed N_{temp} number of times. Line 16 has fixed access patterns. The access patterns of line 17 depend only on the public loop variable. Line 19 has fixed access patterns that depend only on the size N_{temp} of the array L.

The loop in lines 15–18 executes a fixed N number of times. The function CALCNEIGHBOR-HOODBBOX in Line 21 computes the bounding box of the 16×16 neighborhood of the current keypoint using arithmetic operations, and has fixed access patterns. The access patterns of the function CROPOBJECT depend only on the dimensions of the input image O (which is public) and the resolution of the bounding box, which is fixed (from Theorem 6). The function CALCORIENTA-

TIONHIST performs arithmetic operations, and the access patterns of line 23 depend only on the public loop variable.

Thus, the trace produced by the simulator is indistinguishable from the trace produced by a real run of the algorithm. □

Appendix D

Impact of Video Encoder Padding on Visor

In Visor, the source video streams are padded at the camera to prevent information leakage due to variations in bitrate of the encrypted network traffic. However, it may not always be possible to modify legacy cameras to incorporate padding. This security guarantee also comes at the cost of performance and increased network bandwidth.

While we recommend padding the video streams for security, we studied the impact of disabling video encoder padding on Visor so as to aid practitioners in taking an informed decision between security and performance. Disabling padding has two implications on Visor.

First, the encoded stream may also contain interframes in addition to keyframes (see Section 6.6.1). Thus, we have devised an oblivious routine for interframe prediction, which is described in Appendix D.1. Second, the performance overhead of Visor ($\sim 2\times$–$6\times$) reduces to a range of $\sim 1.6\times$–$2.9\times$. This is due to lower interframe decoding latency and smaller number of decoded bits per row of blocks (which are obliviously sorted).

D.1   Inter-prediction for interframes

Inter-predicted blocks use *previously decoded frames* as reference (either the previous frame, or the most recent keyframe). Obliviousness of inter-prediction requires that the reference block (which frame, and block's coordinates therein) remains private during decoding. Otherwise, an attacker observing access patterns during inter-prediction can discern the motion of objects across frames. Furthermore, some blocks even in interframes can be *intra*-predicted for coding efficiency, and oblivious approaches need to conceal whether an interframe block is inter- or intra-predicted. A naïve, but inefficient, approach to achieve obliviousness is to access *all blocks in possible reference frames* at least once—if any block is left untouched, its location its leaked to the attacker.

We leverage properties of video streams to make our oblivious solution efficient: (*i*) Most blocks in interframes are inter-predicted ($\sim 99\%$ blocks in our streams); and (*ii*) Coordinates of reference blocks are close to the coordinates of inter-predicted blocks (in a previous frame), *e.g.*, 90% of blocks are radially within 1 to 3 blocks. These properties enable two optimizations. First, we assume every block in an interframe is inter-predicted. Any error due to this assumption on

intra-predicted blocks is minor in practice. Second, instead of scanning all blocks in prior frames, we only access blocks within a small distance of the current block. If the reference block is indeed within this distance, we fetch it obliviously using `oaccess`; else, (in the rare cases) we use the block at the same coordinates in the previous frame as reference.