Adopting and Scaling Secure Systems with Distributed Trust

Vivian Fang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-106
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-106.html

May 16, 2025

Adopting and Scaling Secure Systems with Distributed Trust

by

Vivian Fang

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 Natacha Crooks
Professor Christopher Fletcher
Professor Aurojit Panda

Spring 2025

Adopting and Scaling Secure Systems with Distributed Trust

Abstract

Adopting and Scaling Secure Systems with Distributed Trust

by

Vivian Fang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Raluca Ada Popa, Chair

Distributed trust is an emerging design pattern that enables building secure systems with strong privacy and integrity guarantees. Yet, adopting these systems in practice poses significant hurdles, from requiring specialized cryptographic expertise to scaling performance and materializing independent trust among parties. To this end, we introduce a suite of works aimed at making secure systems with distributed trust more practical, scalable, and ultimately deployable. On usability, we develop CostCO, an automatic cost-modeling framework for secure multi-party computation protocols, and LegoLog, a configurable transparency log system that automatically generates logs tailored to specific application workloads. We explore scaling oblivious storage in Snoopy, a system capable of scaling throughput linearly with the number of machines without sacrificing security. Finally, we present SVR3, a practical solution for secret key recovery that distributes trust across heterogeneous hardware enclaves in order to protect secrets at scale for hundreds of millions of users.

To my grandparents.

Contents

List of Figures

List of Tables

Acknowledgments

Structured chronologically.

Year -20. I am born. Thanks **mom** and **dad**! And thanks to **Ethan** for the company.

Years -3 to 0. Members of the NetSys lab at UC Berkeley take me under their wing. Thanks to **Ethan J. Jackson** for introducing me to systems research, **Justine Sherry** for introducing me to networking research, and **Aurojit Panda** for introducing me to everything research.

Year 1. I pivot into security research, despite having never taken a security class. Thank you to my advisor **Raluca Ada Popa** for taking a chance on me, for camping out in 465E Soda with me when I was writing my first security paper, and for instilling within me a duty of thoroughness in research. I will take these lessons with me far, far beyond graduate school.

Years 1.5 to 2. The world turns upside down. **Catherine Han**, I am so grateful we became friends through those (and these!) trying times. From mundanities like café hunting and peacefully existing within each others' presence to dropping everything to go to South Korea on a 48-hour notice, our friendship is one of the dearest and closest to my heart. **Hong Jun Jeon**, it was a privilege descending through many rabbit holes together, and I look back fondly on the times when we would screen-share and collectively stare at anything from balls-into-bins bounds to the live price of various assets. **Emma Dauterman**, I am also very glad we started graduate school together—even if I am a tad late to finish!

Year 3. Emma and I commandeer 719/721 Soda, and convince Raluca to buy a couch for the office. We are joined by many other office occupants and (Po)pals throughout the years: **Chawin Sitawarin**, **Darya Kaviani**, **Deevashwer Rathee**, **Jean-Luc Watson**, **Jinhao Zhu**, **Mayank Rathee**, **Norman Mu**, **Sam Kumar**, **Samyu Yagati**, **Sijun Tan**, and **Zhanhao Hu**. Thank you all for the excellent company, snack hours, Tahoe trips, and memories.

Year 4. I have an existential crisis. In a pivotal Zoom call, Justine tells me that this is what the literature refers to as "the doldrums" and that what matters is not how bad it gets, but that it will eventually end. She was right. Over an Aperol spritz at ETH Zurich, **Anwar Hithnawi** renews my faith in academia. **Michele Orrù** introduces me to many wonderful applied cryptographers, and makes me feel like I am a part of the community. Working with **Rolfe Schmidt** at Signal Messenger marks a turning point in my graduate school experience. Thank you for always being generous with your time, for taking me on as your intern, and for letting me teach a bunch of middle and high schoolers lambda calculus. I always look forward to our conversations, and I am so grateful for your mentorship. In the midst of my existential crisis, I also pick up climbing. **Narek Galstyan** and **Rishabh Iyer**, thank you for being my climbing buddies!

Year 5. I undergo the Longest Year Ever. Raluca, Panda, **Natacha Crooks**, and **Christopher Fletcher** agree to be on my committee and start providing feedback that shapes this dissertation. I would also like to give special shout-outs to **Shishir G. Patil** for teaching me the ways of "ML", **Tianjun Zhang** for his infinite humor and the cool scar below my right eye, **Qijing (Jenny)**

Huang for her excellent company and yummy desserts, and **Justin Wong** for the delicious dinners. I am indebted to **Laura Power** for teaching me and **Jiwon Park** how to drive. **Julien Piet**, I have so many things to thank you for: teaching me how to ski, picking me up from the emergency room, rescuing me from car troubles, and providing countless adventures. **Noelle Davis** and **Tess Despres**, our sewing circle was a much-needed respite from the grind of grad school.

Year 6. I exist in purgatory, plotting my escape. Thanks to **Scott Shenker** and Panda for fielding my yearly crises. I would not know what to do next, had it not been for **Amin Tootoonchian**. I also start running to pass the time. **Kevin Zhang**, thank you for pacing my first half, even when my race pace is your recovery pace. Hong, thank you for accompanying all of my long runs in the South Bay, and for all of the late-night calls where we figured out what to do after the PhD.

* * *

Over the years, I have had the privilege of collaborating with many bright and talented individuals: Akshay Ravoor, Akshit Dewan, Chawin Sitawarin, Emma Dauterman, Graeme Connell, Julien Piet, Lloyd Brown, Norman Mu, Shishir G. Patil, Tamás Lévai, Tianjun Zhang, and William Lin. I have also been fortunate to be advised on research projects by Aisha Mushtaq, Aurojit Panda, Barath Raghavan, David Wagner, Ethan J. Jackson, Ioannis Demertzis, Ion Stoica, James (Murphy) McCauley, Justine Sherry, Natacha Crooks, Raluca Ada Popa, Rolfe Schmidt, Sangjin Han, Scott Shenker, Sylvia Ratnasamy, Wenting Zheng, and Yotam Harchol. Thank you all for making research more fun.

The students in the Sky Lab (formerly known as the RISE Lab) also made for a vibrant community: Altan Haan, Alvin Wan, Audrey Cheng, Charles Packer, Chris Douglas, Conor Power, Dacheng Li, David Chu, Eyal Sela, Frank Luan, Gabe Fierro, Jaewan Hong, Lianmin Zheng, Lisa Dunlap, Manish Shetty, Micah Murray, Michael Luo, Naman Jain, Neil Giridharan, Peter Schafhalter, Reggie Frank, Rolando Garcia, Sarah Wooders, Shadaj Laddad, Shangyin Tan, Shreya Shankar, Shu Liu, Simon Mo, Siyuan (Ryans) Zhuang, Soujanya Ponnapalli, Stephanie Wang, Sukrit Kalra, Tyler Griggs, Vinamra Benara, Wei-Lin Chiang, Wenshuo Guo, Xiaoxuan (Lily) Liu, Zhuohan Li, Ziming Mao, and Zongheng Yang. I am also grateful to our lab's amazing administrative staff: Boban Zarkovich, Carlyn Chinen, Dave Schonenberg, Ivan Ortega, Jon Kuroda, and Kattt Atchley. Carissa Caloud, Jean Nguyen, Judy Smithson, and Shirley Salanio are the underlying reason I (and many other graduate students) are actually able to graduate.

This disseration is in part supported by the NSF Graduate Research Fellowship. Support was also provided by gifts to the RISE/Sky Lab from the Sloan Foundation, Accenture, Alibaba, AMD, Amazon, Amazon Web Services, Ant Group, Anyscale, Cisco, Ericsson, Facebook, Futurewei, Google, IBM, Intel, Intesa Sanpaolo, Microsoft, Lambda, MBZUAI, Microsoft, NVIDIA, Samsung SDS, SAP, Scotiabank, Splunk, Uber, and VMware.

* * *

Finally, this dissertation would not exist without the unconditional love and support from **Wen Zhang**, who is also the best cat parent to **Toto** and **Mini**. Without you telling me to "wait a bit and think about it," I certainly would have dropped out of graduate school years ago.

Co-authored material

Parts of this dissertation are based on previously published material co-authored with others:

→ Chapter 2 is based on the following publication:

[98] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. CostCO: An automatic cost modeling framework for secure multi-party computation. In *EuroS&P*, 2022.

→ Chapter 3 is based on the following publication:

[99] Vivian Fang, Emma Dauterman, Akshat Ravoor, Akshit Dewan, and Raluca Ada Popa. LegoLog: A configurable transparency log. In *EuroS&P*, 2025.

→ Chapter 4 is based on the following publication:

[76] Emma Dauterman*, Vivian Fang*, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the bottlneck of scalable oblivious storage. In *SOSP*, 2021.

→ Chapter 5 is based on the following publication:

[64] Graeme Connell*, Vivian Fang*, Rolfe Schmidt*, Emma Dauterman, and Raluca Ada Popa. Secret key recovery in a global-scale end-to-end encryption system. In *OSDI*, 2024.

* denotes equal contribution

Chapter 1

Introduction

Many systems today *distribute trust* across multiple parties such that the system provides certain security properties if a subset of the parties are honest. A distributed-trust system is deployed across n parties (in this dissertation, we will subsequently refer to these as trust domains) where, if there are no more than f independent corruptions, the system provides certain security, privacy, and/or integrity properties. In recent years, we have seen an explosion of academic and industrial cryptographic systems built on distributed trust, spanning private search [77, 78, 249, 317], private analytics [14, 32, 68, 91, 153], private media delivery [129], private blocklist lookups [169], private DNS [289], anonymous messaging [54, 69, 92, 178, 179, 323], cryptocurrency wallets [100, 107, 168, 244, 266, 278, 308]), key recovery [173, 303, 316], and private storage [94].

We start by examining existing distributed-trust systems (§1.1), the challenges organizations face in deploying distributed-trust systems today (§1.2), and then we present our contributions to address these challenges (§1.3).

1.1 Existing distributed-trust systems

Multi-party computation. Secure multi-party computation (MPC) is now efficient enough for several real-world use cases [44, 89, 148, 223, 224, 226, 282, 307]. However, because there is no MPC protocol that wins for all workloads, researchers have designed a number of *hybrid protocols* [31, 106, 112, 133, 160, 197, 221, 265] that combine different MPC protocols to bring orders of magnitude performance improvement over using just one MPC protocol to run the entire workload. While promising for performance, designing a hybrid protocol requires deep cryptographic expertise and is often bespoke to the workload.

Privacy-preserving analytics. Prio [68] splits trust across two servers to compute aggregate statistics without revealing individual users' data and has been deployed for Firefox telemetry [91] and COVID-19 exposure notification analytics [14]. For Firefox telemetry, Firefox runs one server and the ISRG (the public-benefit corporation behind Let's Encrypt, which offers free TLS certificates) runs the other. The COVID-19 exposure notification system computes statistics across iOS

and Android users where the ISRG and the National Institute of Health each run a server. To enable other organizations to easily run a Prio system, the ISRG has announced Divvi Up, a service where the ISRG acts as the second trust domain for a Prio deployment (the organization building the application acts as the "first trust domain") [1, 153]. The challenges ISRG faced in setting up Divvi Up illustrate just how hard it is to set up a distributed-trust system correctly [108, 109]. For example, debugging and running integration tests now must take place across organizations that don't have a common release process or deployment system.

Financial custody. Users transfer cryptocurrency by signing transactions, and so transaction signing keys can secure millions of dollars. Many financial custody companies deploy solutions where the signing key is split across hardware security modules (HSMs), and the HSMs run a multi-party computation to generate a signature on a transaction [100, 107, 168, 244, 266, 278, 308]. In this way, no HSM ever holds the entire signing key. A limitation of the financial custody companies that deploy these solutions is that they are themselves centralized and only provide security if the company is honest at setup time. One company deploys and maintains all of the secure hardware, and the end-user cannot check that the system is set up and distributes trust in the way that the company claims. Furthermore, if the company locks itself out of its machines to defend against post-setup compromise, there is no way to patch bugs or push updates.

Key recovery. In applications like end-to-end encrypted messaging [80, 214, 287], users often lose access to their secret keys, and the service provider needs a way to recover the secret key without compromising security of its users. This is a usability-security trade-off, as the service provider must balance the need to recover keys with the need to protect users' data. We can imagine two extremes: one where the user is responsible for their own key recovery (e.g., by physically storing their secret key somewhere) and one where the service provider directly stores the user's secret key. On one hand, users are notoriously bad at storing long-term secrets [262], and if the user loses their secret key, they may lose access to their data. On the other hand, if the service provider stores the user's secret key, the service provider now has access to the user's data, thus undermining the security guarantees of end-to-end encryption. Current deployed systems [13, 173, 200, 303, 316, 320] prevent brute-force attacks by using secure hardware to limit the number of PIN guesses and provide strong protection against service provider administrators and cloud providers. While these systems all represent significant advances in password-based key recovery, they rely on the security guarantees of a *single* type of secure hardware.

Private storage. Organizations increasingly outsource sensitive data to the cloud for better convenience, cost-efficiency and availability [101, 176, 279]. While end-to-end encryption (E2EE) can protect the *contents* of the data, it does not protect against *access patterns* from leaking sensitive information [45, 82, 124, 152, 161, 171]. Oblivious RAM (ORAM) is a cryptographic primitive that hides a user's access patterns from untrusted storage providers, and there has been significant work in building efficient private storage systems based on ORAM [29, 47, 72, 116, 240, 263, 269, 292–294, 322]. However, these systems often have poor scalability, with throughput bottlenecked by centralized components that cannot be efficiently and securely parallelized [72, 269, 322].

1.2 Challenges in deploying distributed-trust systems

Below, we summarize the types of challenges that organizations face when adopting and scaling distributed-trust systems:

→ **Cryptographic expertise (C1).** Many distributed-trust systems require deep cryptographic expertise to set up and run. They require an intimate understanding of cryptographic protocols and their security properties, often resulting in a bespoke protocol design.

→ **Materializing trust domains (C2).** In practice, achieving truly independent trust domains proves challenging. We find that in many deployed distributed trust systems, the trust domains are either not independent or require a challenging, application-specific setup that often required cross-organization coordination.

→ **Scaling (C3).** Due to the security guarantees of distributed-trust systems, they often have performance limitations that can render them impractical for real-world workloads with plaintext counterparts.

1.3 Contributions and dissertation outline

This dissertation aims to address these challenges by making secure systems with distributed trust more practical for adoption, scalable to large workloads, and deployable in real-world environments. We focus on systems challenges and present solutions that bridge the gap between theoretical security and practical deployment. This dissertation is organized in two parts: Part I focuses on addressing usability hurdles in adopting distributed-trust systems, and Part II focuses on addressing their scalability and deployability challenges in real-world environments. In total, we present four major contributions.

Part I: Adopting distributed-trust systems

Chapter 2: CostCO (C1). The efficiency of MPC solutions depends significantly on protocol selection. Different MPC protocols excel in different contexts, but determining which protocol to use requires deep cryptographic expertise. CostCO addresses this challenge by automatically generating empirical cost models for different MPC protocols. By profiling performance characteristics of MPC protocols in the deployment environment, CostCO enables non-experts to leverage appropriate protocols without manual performance tuning. This automation enables hybrid-protocol MPC, where different parts of a computation use different protocols based on their characteristics, significantly improving performance compared to single-protocol approaches.

Chapter 3: LegoLog (C1). Transparency logs provide verifiability in many security-critical applications, from validating web certificates to software distribution. However, selecting or designing an appropriate transparency log for a specific application is challenging. LegoLog introduces the concept of a configurable transparency log that automatically generates specialized transparency logs for given workloads. Rather than requiring developers to analyze trade-offs be-

tween different transparency log designs, LegoLog takes a workload description and constraints, then outputs a tailored transparency log implementation. This approach democratizes the use of transparency logs, enabling non-experts to deploy them effectively.

Part II: Scaling distributed-trust systems

Chapter 4: Snoopy (C3). Oblivious storage systems hide access patterns from untrusted storage providers but suffer from scalability bottlenecks. Existing systems either rely on a trusted proxy (which becomes a bottleneck) or run on a single machine (limited by hardware resources). Snoopy introduces a scalable oblivious storage system that distributes both computation and storage across multiple machines without sacrificing security. Snoopy achieves linear throughput scaling with additional hardware resources, similar to plaintext storage systems, but with strong security guarantees.

Chapter 5: SVR3 (C2, C3). End-to-end encrypted systems need a means to recover users' secret keys without compromising users' security. Existing approaches rely on a single type of secure hardware, creating a single point of security failure. SVR3, developed in collaboration with Signal Messenger [287], introduces a key recovery system that distributes trust across different types of secure hardware (TEEs) from different vendors running in different cloud providers. This approach ensures that a vulnerability in any single hardware platform does not compromise user secrets. SVR3 demonstrates the practical deployment of distributed trust in a production system serving tens of millions of users, with careful consideration of the security guarantees that real-world TEEs provide today.

Finally, in Chapter 6, we conclude with a summary of contributions and discuss future research directions.

Part I

Adopting

Chapter 2

CostCO: Automatic cost modeling of secure multi-party computation protocols

2.1 Introduction

Secure collaborative computation [44,89,148,223,226,282,307], is an increasingly popular paradigm where multiple organizations with sensitive data run an analysis over their *aggregate* data, without revealing their individual sensitive data to each other. Computing on multiple parties' data is both advantageous and necessary in many cases ranging from finance to healthcare. For example, money laundering—where criminals transfer assets across financial institutions to mask their activities—is illegal in most jurisdictions and banks are required to detect and report such activity. However, detecting money-laundering is challenging because it requires banks to share sensitive customer transaction data with each other [276], and they are unwilling to do so because of business competition.

Over the past three decades, researchers have made impressive progress towards a cryptographic approach to this problem: *secure multi-party computation* (MPC) [23, 73, 145, 162, 163, 194, 229, 319, 326]. At a high level, MPC allows n parties $p_1, \ldots, p_n$ with corresponding inputs $x_1, \ldots, x_n$ to learn the output of a public function $f(x_1, \ldots, x_n)$ without revealing each party's x_i to other parties. Due to these efforts, MPC is now efficient enough for several real-world use cases [44, 89, 148, 223, 226, 282, 307]. Because there is no single MPC protocol that wins for all workloads, researchers have begun to design a number of *hybrid protocols* [31, 106, 112, 133, 160, 197, 221, 265] that combine different MPC protocols to bring orders of magnitude performance improvement over using just one MPC protocol to run the entire workload.

Consequently, *hybrid-protocol compilers* [42, 51, 151, 243] emerged in an effort to automate the manual process of optimally combining different MPC protocols for a given application. Figure 2.1 depicts the standard workflow of a hybrid-protocol compiler. In order to construct a hybrid protocol for a given program, the compiler partitions the program and uses a cost model to select the MPC protocol to run on each partition. The resulting hybrid protocol can then be executed by the parties who want to securely compute the program.

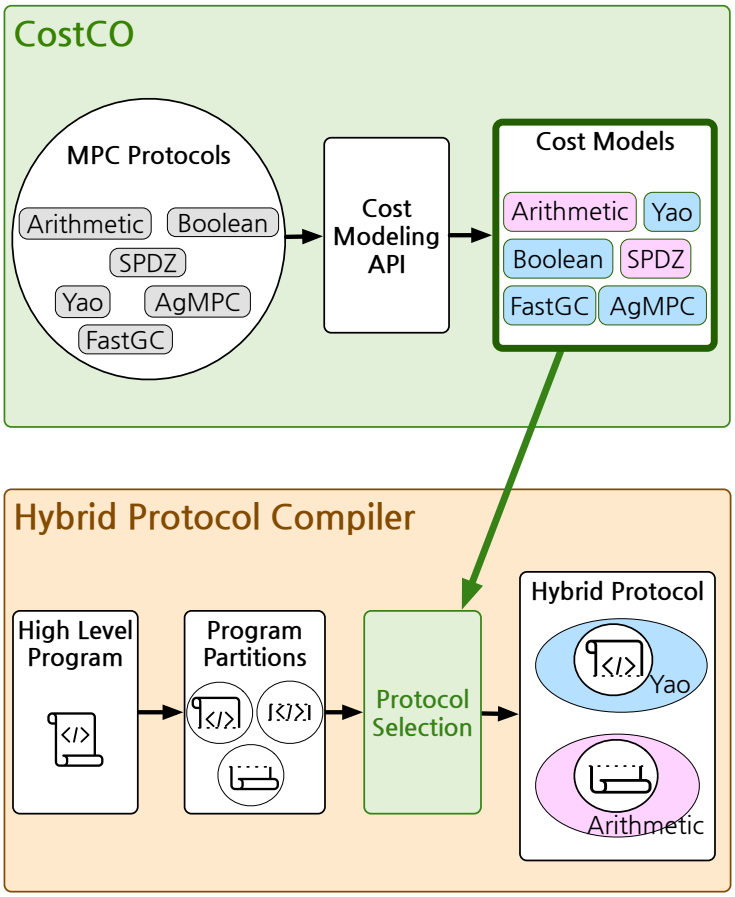


Figure 2.1: The standard workflow of hybrid-protocol compilers is drawn on the bottom (orange box). As shown on the top (green box), CostCO generates cost models that the hybrid-protocol compiler queries during protocol selection.

A hybrid-protocol compiler crucially depends on its *cost model*, which it queries to determine the concrete costs of different options for protocol assignment. The main problem with existing cost models is the manual effort needed to derive them. Multiple issues arise as a result:

- *Burdensome effort.* Deriving a cost model for a specific MPC protocol requires a deep understanding of its runtime complexity. For example, EzPC [51] relies on distilling the properties of ABY [84] into hard coded heuristics specialized to a deployment and Kerschbaum, et. al [164, 273] manually derive cost models for every building block in their two supported MPC protocols. Others [42, 151, 243] set up experiments by hand and fix parameters like the data size.
- *Lack of extensibility.* Existing compilers base their cost models on certain assumptions about the deployment environment and the supported MPC protocols. EzPC [51] relies on rigid

heuristics while newer compilers [42, 151, 243] use a cost model specific to the set of protocols they initially support. Both approaches make it difficult to integrate new MPC protocols into the compiler; new heuristics need to be manually found and MPC protocols with cost functions that differ from ABY (e.g., [41, 104, 319]) require nontrivial effort to support. Furthermore, existing MPC protocols are actively developed on and frequently branch into different versions, e.g., ABY [84, 222, 242] and SPDZ [73, 162, 163]. Adapting existing cost models to account for new versions of protocols necessitates human effort, impeding the wider adoption and availability of MPC.

– *Limited decision-making capabilities.* In addition to being unable to support more MPC protocols, existing compilers are limited by their cost models which predict only one type of cost (runtime). As a result, they cannot make more sophisticated decisions like choosing between an MPC protocol [146] and its memory-optimized version [135].

In this chapter, we present CostCO, a cost modeling framework for MPC capable of automatically deriving an accurate cost model for a given MPC protocol. At a high level, CostCO takes as input an MPC protocol with a specification, builds a statistical model indicating which programs to test on the MPC protocol, measures the protocol running on these samples, and uses the results to compute a cost model. The computation of cost models in CostCO leverage the common properties of a class of MPC protocols. Specifically, security requirements of MPC imply that computation time and execution are deterministic and input-data independent, which in turn means that all branch arms are evaluated during execution (we refer to this property as non-branching). Additionally, many recent MPC protocols [22, 23, 73, 84, 97, 163, 319, 326] have quadratic communication, computation and memory complexity. In combination these insights allow CostCO to efficiently generate cost models *without* requiring additional user input such as representative workloads, etc. In contrast to prior cost modeling works [42, 151, 243], CostCO is:

– *Automatic.* A protocol provider supplies an MPC protocol implementation together with a short specification, and CostCO automatically infers a cost model, both in terms of a symbolic equation and an empirical cost for each metric. Subsequent versions of MPC protocols require less effort to generate cost models and integrate into the hybrid protocol compiler. CostCO can also automatically generate cost models for different deployment settings the secure computation takes place on, which frees the compiler from deployment assumptions (e.g., network conditions) imposed by hard-coded heuristics.

– *Extensible.* CostCO's API (§2.3) is rich enough to allow it to be used with a range of different MPC protocols, running in a wide variety of different deployment settings. We have successfully used CostCO to generate cost models for 7 different MPC protocols which run the gamut from ones that use arithmetic and Boolean secret sharing [84] to ones based on garbled circuits [319] and homomorphic encryption [97]. Using CostCO in each of these cases required less than 200 lines of code, demonstrating CostCO's expressivity.

– *Versatile.* In addition to generating cost models that predict an MPC protocol's runtime, CostCO also generates cost models that predict network communication and peak memory consumption, which enables compilers to additionally consider memory-optimized MPC protocols.

Improving existing compilers. We build a compiler that uses cost models generated by CostCO in order to reduce the effort that is otherwise required for manually developing cost models for an MPC protocol. At the same time, when compared to existing frameworks and their benchmarks, our compiler still produces comparable protocol assignments. By considering the different types of costs CostCO models, our compiler is also able to make more sophisticated decisions, e.g., taking into account performance degradation from memory pressure. We believe that the research on hybrid-protocol compilers can now focus on the problem of effectively searching the complex space of combinations of MPC protocols, without expending effort on cost modeling.

2.1.1 Techniques summary

In order to generate cost models for a variety of MPC protocols, CostCO first needs a way to interface with each protocol. Such a task is challenging because different MPC protocol implementations have widely varying APIs. CostCO manages to provide a common interface in a simple API that is still rich enough to express different MPC protocols (described in §2.3).

Once CostCO is able to interface with a given MPC protocol, the main challenge lies in working out the relations between the primitives for the total cost. MPC protocols typically publish their asymptotic complexity, which do not contain lower-order terms. CostCO needs to first recover lower-order terms that influence the real cost to obtain the set of primitive features that influence performance. Requiring users to derive the correct set of features is both burdensome, and error prone since feature importance might itself depend on factors such as the deployment scenario. Instead, CostCO *automatically* identifies the set of important features by leveraging techniques from response-surface methodology [138] so the user only need provide the set of features that potentially influence performance (described in §2.4).

Next, CostCO uses the observation that the performance and resource requirements for MPC protocols are independent of input data in order to automatically generate an empirical cost model. Determining the set of important features, and generating an empirical cost model requires empirical experiments, however running an MPC circuit can be expensive in terms of runtime and resource costs. Therefore, CostCO draws techniques from the experiment design literature [35, 138, 174] and runs in two stages to reduce the overall number of empirical observations required.

Evaluation summary. We implement and port 7 MPC protocols to CostCO (§2.5). Each protocol takes less than 200 LOC to use CostCO (§2.6.1) and CostCO generates cost models with greater accuracy than prior cost modeling approaches [42, 151, 243] (§2.6.2). In order to demonstrate the utility of the cost models generated by CostCO, we implement a hybrid-protocol compiler along with 6 applications and find that our compiler makes comparable, if not better (up to 41% faster) protocol assignments than the current state-of-the-art hybrid-protocol compiler which uses manually derived cost models (§2.6.3).

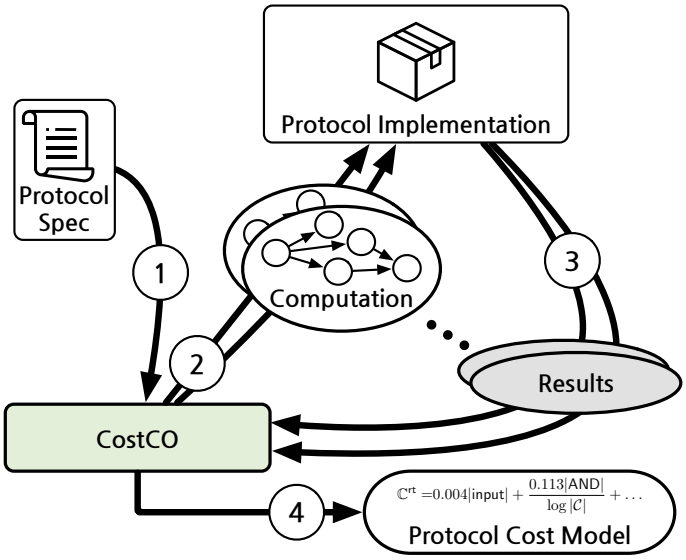


Figure 2.2: CostCO's workflow. CostCO is run on the deployment environment where the secure computation is planned to be executed.

2.2 CostCO overview

CostCO is designed to automatically generate cost models that provide an estimate of the execution time of running a given computation using a specific MPC protocol.

2.2.1 CostCO architecture

CostCO's system architecture consists of a pipeline that automatically synthesizes a cost model from a list of user-provided features for a given MPC protocol executed in a fixed deployment environment. Figure 2.2 depicts CostCO's workflow. At a high level, ① CostCO receives a protocol specification for the MPC protocol and uses it to generate the exact computation experiments to run. ② CostCO then runs the experiments on the MPC protocol's implementation and ③ determines the next set of experiments to run. ④ CostCO uses the experiment results to infer the lower order asymptotic terms and finally output an empirical cost model.

CostCO is run on the deployment environment which consists of the machine or cluster at each party where the secure computation will be run and the network environment among them. For example, some banks who wish to collaborate on anti-money laundering [276] can run CostCO on their setups. For some, this means their on-premise cloud, while for others, a major cloud provider. This is the same deployment on which hybrid-protocol compilers [42, 151] are designed to run on. CostCO also computes cost models for fixed field sizes and security parameters, i.e., CostCO should be rerun if the field sizes or the security parameters change.

Similar to existing MPC literature [42, 84, 272, 319], we expect that MPC computations are expressed as *circuits*, a directed acyclic graph (DAG) where vertices represent an MPC protocol's *primitive operations*, also known as *gates* (e.g., AND/XOR in garbled circuits [319] and sum/product in arithmetic circuits [84]). The edges of the DAG represent dataflow between the gates. The *depth* of a circuit is the path length from the computation's start to end, where edges are weighted by the number of communication rounds the edge's source vertex (corresponding to a gate) requires. For example, consider a circuit for a round-based arithmetic MPC protocol [84] that consists of an addition followed by a product. In this case, the addition requires no rounds while the product requires 1 round and thus the depth of the circuit is 1.

To compute the cost model of a protocol π, CostCO requires the following inputs:

1. *Protocol specification* S_π. In order to be able to interface with π, CostCO requires a short specification for π. At a high level, a specification consists of the units of computation and parameters of π that CostCO needs in order to derive π's cost model. We describe the specification in more detail in §2.3.
2. *Protocol implementation.* This is a specific implementation of π, which implements the interface described in §2.4.2. CostCO uses this implementation to synthesize empirical cost models.

Note that, unlike prior work [42, 151, 243], CostCO only requires a list of parameters that the cost model depends upon. The list may include parameters that do not appear in the final cost model as CostCO can automatically perform parameter selection and filter out extraneous parameters. Crucially, CostCO can derive a detailed cost model from *shallow parameters*, that is, parameters that can be easily identified without a deep understanding of the protocol. For example, in order to generate the detailed cost model for AG-MPC [319], a Boolean-circuit MPC protocol, the user inputs:

$$\left\{|\mathsf{AND}|, |\mathsf{XOR}|, |\mathsf{input}|, |\mathsf{output}|, \mathsf{depth}, |\mathcal{C}|, \frac{1}{\log(|\mathcal{C}|)}\right\},$$

where $|\mathcal{C}| = |\mathsf{AND}| + |\mathsf{XOR}|$ is the total number of gates in the circuit, and depth is circuit depth as described earlier. Of these terms $\{|\mathsf{input}|, |\mathsf{XOR}|, |\mathsf{AND}|, \mathsf{depth}, |\mathsf{output}|\}$ come from how computation in AG-MPC is represented (as a Boolean circuit) and the terms $\{|\mathcal{C}|, 1/\log(|\mathcal{C}|)\}$ appear in the asymptotic $O(\cdot)$ complexity reported in Table 1 of the original paper [319]. From the list of parameters, CostCO automatically generates the following empirical cost model for execution time in milliseconds:

$$\hat{\mathbb{C}}^{\mathsf{RT}}_{\mathsf{AGMPC}} = .004|\mathsf{input}| + .033|\mathsf{AND}| + \frac{.113|\mathsf{AND}|}{\log|\mathcal{C}|} + 56.3 \tag{2.1}$$

The protocol specification also only needs to be written once for an MPC protocol and can be reused for different deployment settings. The party who writes this specification can be the designer of the MPC protocol or the developer who wants to use the protocol in a hybrid-protocol compiler. We further elaborate on the shallow parameters the protocol specification writer needs to provide in §2.3.

After the inputs are specified, CostCO can use these inputs to generate the appropriate cost models. At a high level, CostCO's cost modeling workflow is as follows:

1. (§2.4.1) CostCO first carefully chooses a set of profiling experiments using experiment design. The user may optionally provide CostCO with a maximum experiment size (2^{13} operations by default) if they want to control CostCO's total execution time.
2. (§2.4.2) Generates computations for the profiling experiments, executes them on the implementation I_π, and collects performance measurements.
3. (§2.4.3) Automatically synthesizes and outputs both the *abstract* and *empirical cost models*, denoted as $\mathbb{C}_\pi$ and $\hat{\mathbb{C}}_\pi$, respectively. $\mathbb{C}_\pi$ and $\hat{\mathbb{C}}_\pi$ each contain cost models for computation, memory, and network consumption.

Compared to plaintext cost modeling systems, MPC cost modeling does not require knowledge of the input data. This is because, MPC's privacy properties mean that a program's performance *cannot* depend on data content – and generally, the overhead depends only on computation size and the input data size(s). As a result, executions are deterministic and non-branching, making cost modeling CostCO feasible with only a few additional assumptions. We elaborate on the assumptions CostCO makes and their limitations in §2.2.3 and §2.2.4.

2.2.2 Usage in a hybrid-protocol compiler

As previously explained in §2.1 and Figure 2.1, a hybrid-protocol compiler mixes MPC protocols $\pi_1, \ldots, \pi_n$. Parties invoke a hybrid-protocol compiler on the same deployment where they plan to run their secure computation. Since transitioning from a protocol π_i to π_j needs a special MPC conversion in order to maintain security, the compiler also needs to support *conversion protocols* $\pi_{i \to j}$, which CostCO can automatically generate cost models for.

At a high level, the hybrid-protocol compiler divides the user program into pieces and tries to determine the most cost effective way of dividing and assigning pieces of computation to MPC protocols. The exponential search space makes it impractical to directly run and measure the cost of every possible protocol assignment. Instead, the compiler infers the runtime, memory, and network requirements of each assignment by querying each of the cost models returned by CostCO.

Each empirical cost model $\hat{\mathbb{C}}_\pi = (\hat{\mathbb{C}}_\pi^{\mathsf{RT}}, \hat{\mathbb{C}}_\pi^{\mathsf{MEM}}, \hat{\mathbb{C}}_\pi^{\mathsf{NW}})$ takes as input a *circuit file* representing a computation to be run (described in §2.4.2), and outputs the inferred execution time, peak memory usage, or network cost of executing the computation using π on the parties' deployment environment. The cost for executing any program can be computed by summing up the cost of executing each piece as well as the cost of conversions, which is given by plugging the data size to be converted into the cost model for the relevant conversion $\hat{\mathbb{C}}_{\pi_i \to \pi_j}$. We describe the integration of CostCO in a compiler in §2.5.

2.2.3 System assumptions

The security properties of MPC lend themselves to appealing computational properties, namely, the execution is non-branching (all branch arms are evaluated), input-data independent, and deterministic. This means CostCO does not require additional inputs that are normally required by generic cost modelers, e.g., representative workloads to run [117, 315, 325]. CostCO also makes domain-specific assumptions that let it support automatically cost modeling many MPC protocols.

Quadratic approximation

As mentioned in §2.1, CostCO approximates a cost model as polynomial with monomials of maximum degree = 2. For many MPC protocols' cost models [22, 23, 73, 84, 86, 97, 163, 225, 277, 326], a quadratic approximation is sufficient. This is due to the computation time and the number of communication rounds of an MPC circuit normally being bounded by the size of the circuit ($|\mathcal{C}|$), and in the worst case no gate requires more than a constant number of all-to-all communication rounds, bounding the number of messages and hence bandwidth requirements to $|\mathcal{C}|^2$. While CostCO can approximate non-polynomial cost terms as second degree polynomials, for some protocols such as AG-MPC [319] explicitly specifying the asymptotic terms can improve accuracy, especially when they contain non-polynomial factors (e.g., $1/\log|\mathcal{C}|$). As a result, for such protocols, CostCO allows users to provide non-polynomial terms appearing in a protocol's asymptotic complexity as input and uses this information to improve cost model accuracy by including them in the set of features considered when synthesizing cost models.

This assumption also implies that the cost model is smooth (differentiable everywhere). That is to say, the protocol's empirical cost model is not piecewise; the constant factors are expected to not arbitrarily change with respect to computation size. However, CostCO can still enable the compiler to make decisions on piecewise costs. For example, CostCO can extrapolate the performance degradation from running out of memory by artificially limiting the system's memory and measuring the difference in runtime of a circuit. By using the memory model generated by CostCO in tandem with the performance under memory pressure, our compiler can account for a protocol assignment's performance degradation when its peak memory usage exceeds the available system memory (§2.5).

Fixed parameters

CostCO runs on the deployment that the users plan to run their secure computation on, which is the same deployment existing hybrid compilers [42, 151] run on. Cost models are created for fixed field sizes and security parameters, which allows the cost models to be viewed as a function of computation size and input data size. CostCO should be rerun if the field sizes or security parameters change.

2.2.4 Limitations

Circuit structure

CostCO only considers the number of gates in the circuit and its depth. For the MPC protocols we consider, circuit depth corresponds to the number of communication rounds required when evaluating the computation. We show in §2.6 that considering these factors is sufficient for producing relatively accurate cost models, despite not considering round-level parallelism in protocols with non-constant communication rounds.

Circuit optimization

The cost models generated by CostCO are derived from mapping several unmodified circuits (§2.4.2) to the costs experienced during their execution. In practice, some protocols [163] can reorder the evaluation order of the units comprising the circuit to improve performance. Significant modifications could render analysis on the unmodified circuit useless to extrapolate to the modified circuit which the protocol actually evaluates. In order to produce accurate models for these protocols, CostCO would need a way to execute a circuit without optimizations from the protocol. The protocol would need to implement an interface that takes the computation and outputs the optimized circuit that can be used as input to the cost model.

Scheduling

MPC protocols may use multiple threads throughout their execution. The runtime of these threads is at the mercy of the scheduler while CostCO is profiling costs. Scheduling decisions may influence the overall runtime performance of the protocol, but CostCO makes no attempt to understand such scheduling decisions.

2.3 CostCO specification

The first challenge that CostCO needs to address is how to interface with a given MPC framework. MPC frameworks differ significantly from each other both in how computation is expressed and in how it is executed. CostCO manages to provide a common interface using a simple API. We observe that many MPC protocols express computation as a sequence of *primitive operations*, also known as *gates*. A primitive operation is either a *computational gate* or a *data gate*. For example, the primitive operations of AG-MPC are the AND and XOR gates (computational units), and the input and output gates (data units). CostCO assumes the cost model can be computed as a function of the MPC protocol's computational primitive operations. Intuitively, because MPC requires the execution to be non branching, deterministic, and input-data independent, the cost of an MPC protocol can be viewed in terms of the computation itself. The specification for an MPC protocol π captures the primitive operations in π.

Given the specification, CostCO will automatically generate experiments to evaluate the MPC framework, collect the results, and derive the abstract and the empirical cost models.

A protocol specification S_π for protocol π is a file that details the following:

1. A set of *gates*, each of which is a computational gate or data gate and has the form $\mathsf{g_n} = (\mathsf{i_n}, \mathsf{o_n}, d_n)$, where n is the name of the gate, $\mathsf{i_n}$ is the number of inputs, $\mathsf{o_n}$ is the number of outputs, and d_n is the depth. Depth here indicates the number of communication rounds required to run this computational unit. We define $\mathcal{G}$ to be the entire set of π's gates. Inputs and outputs are assumed to be gates, i.e., $\mathcal{G}_{\mathsf{io}} = \mathsf{input}, \mathsf{output}$ and $\mathcal{G}_{\mathsf{io}} \subset \mathcal{G}$. For a given computation, we define $\mathcal{S} = \{s_{\mathsf{g}}\}$ to be the counts of every gate $\mathsf{g} \in \mathcal{G}$ in that computation.
2. A set of *asymptotic terms*, each of which is some function $h : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}$ that expresses parameters in the cost model as a function of some subset of $\mathcal{S}$ and appears as a term in the worst-case asymptotic complexity bound $O(\cdot)$ of π. We define $\mathcal{H}$ to be the entire set of π's asymptotic terms. The counts of gates $\mathcal{S}$ and the asymptotic terms $\mathcal{H}$ evaluated on $\mathcal{S}$ make up the set of features $\mathcal{F} = \{f_1, \ldots, f_{|\mathcal{F}|}\} = \mathcal{S} \cup \{h(\mathcal{S}) | h \in \mathcal{H}\}$ that CostCO uses to derive a cost model. The cost model is expected to have the form:

$$\mathbb{C}_\pi(f_1, \ldots, f_{|\mathcal{F}|}) = \mathbf{r}^\top \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{|\mathcal{F}|} \end{bmatrix} + \mathbf{s}^\top \begin{bmatrix} f_0 f_0 \\ f_0 f_1 \\ \vdots \\ f_{|\mathcal{F}|} f_{|\mathcal{F}|} \end{bmatrix},$$

   where $\mathbf{r}_i, \mathbf{s}_i \in \mathbb{R}^+$.

Example. To make the specification writing process more concrete, we will elaborate on the specification for AG-MPC [319] introduced in §2.2.1. Boolean MPC requires the computation to be specified as a Boolean circuit composed of XOR and AND operations. Note that since CostCO assumes that the deployment is fixed, variables such as the number of parties and network bandwidth are also fixed and thus integrated into the empirical constants of the cost model. Both operations have two inputs and one output. AG-MPC is a constant-round protocol, thus both operations have a depth of 0. Therefore, AG-MPC has the following gates:

$$\mathcal{G} = \{(\mathsf{XOR}, 2, 1, 0), (\mathsf{AND}, 2, 1, 0)\} \cup \mathcal{G}_{\mathsf{io}}.$$

Table 1 of AG-MPC's original publication [319] lists an asymptotic complexity of $O(|\mathcal{C}|/\log|\mathcal{C}|)$. Because CostCO is capable of automatically inferring quadratic parameters (monomial degree = 2),

$$\mathcal{H} = \left\{ |\mathcal{C}|, \frac{1}{\log|\mathcal{C}|} \right\}.$$

2.4 Synthesizing cost models

Given an MPC protocol specification and implementation (§2.3), the goal of CostCO is to compute a cost model for that protocol. When designing CostCO's cost modeling algorithm (Algo-

Algorithm 2.1 Cost model synthesis

Input: $\mathcal{F}, \mathcal{H}, d$
Output: A cost model $(\mathcal{F}^*, \beta^*)$ for π
$\tilde{\mathcal{F}} \leftarrow \emptyset$
$X_1 \leftarrow \text{PBD}(|\mathcal{F}|)$
$Y_1 \leftarrow \textsc{RunCircuitFiles}(X_1)$
$\beta \leftarrow \textsc{LeastSquares}(X_1, Y_1)$
for $f_i \in \mathcal{F}$ **do**
    if $0 \notin \textsc{ConfInterval}(\beta_i)$ **then**
        $\tilde{\mathcal{F}} \leftarrow \tilde{\mathcal{F}} \cup \{f_i\}$
    end if
end for
$X_2 \leftarrow \text{CCD}(|\tilde{\mathcal{F}}|)$
$Y_2 \leftarrow \textsc{RunCircuitFiles}(X_2)$
$\mathcal{G} \leftarrow \tilde{\mathcal{F}} \cup \mathcal{H}$
$\mathcal{G}^2 \leftarrow \{g_i \cdot g_j \mid g_i, g_j \in \mathcal{G}\}$
$(\mathcal{F}^*, \beta^*) \leftarrow \textsc{FoBa}(X_2, Y_2, \mathcal{G} \cup \mathcal{G}^2)$

rithm 2.1), we initially considered using techniques from optimal experiment design (OED) [151], which provides a methodology for choosing samples while minimizing some desired metric (e.g., metrics related to expected error of the estimator). Existing cost modelers like Ernest [315] leverage OED to predict the performance of large-scale data analytics workloads. However, OED requires a known model which, in the case of AG-MPC above, would require knowing the abstract cost model *a priori*:

$$\mathbb{C}^{\mathsf{rt}}_{\mathsf{AGMPC}} = x_0 + x_1|\mathsf{input}| + x_2|\mathsf{AND}| + x_3\frac{|\mathsf{AND}|}{\log|\mathcal{C}|} \tag{2.2}$$

Generating such a model is challenging for a user because it requires a non-trivial understanding of the protocol and in some cases might rely on implicit assumptions about the deployment (e.g., network latency). While it is hard for the user to specify the exact abstract cost model equation, it is easier for the user to come up with factors that affect performance. For example, as previously mentioned in §2.2.1, MPC protocols commonly report their asymptotic $O(\cdot)$ complexity. Thus, CostCO asks the user to only specify such factors (the user can also provide more than is necessary) and does not require the user to figure out how to combine the factors. CostCO blends techniques from experiment design and statistics to cull important features (§2.4.1), generate and run a set of experiments (§2.4.2), and perform regression analysis to construct the abstract and empirical cost models (§2.4.3).

	Features		
Experiment	f_1	f_2	f_3
1	1	1	1
2	1	0	0
3	0	1	0
4	0	0	1

Table 2.1: PBD for three features and two levels. After running every experiment in PBD, CostCO can analyze which features f_i to drop before the second phase of more intensive experiments.

2.4.1 Design of experiments

Given a set of potential features, CostCO still needs to figure out which experiments to run. As mentioned in §2.4, we cannot use OED because we do not know the abstract cost model (e.g., eq. (2.2)) *a priori*. Other techniques that predict execution time using random sampling [144, 284] do not provide a principled rule on the amount of samples to collect and ended up needing hundreds of samples to train a model. CostCO's approach is inspired by ideas from response-surface methodology (RSM) [138], a technique used in several disciplines to explain how different features influence an observed value. We remain sample efficient by leveraging two techniques in the experiment design literature used in tandem with RSM.

CostCO initially starts with a set of user-provided potential features $\mathcal{F}$. In our running AG-MPC example,

$$\mathcal{F} = \{|\mathsf{AND}|, |\mathsf{XOR}|, |\mathsf{input}|, \\ |\mathsf{output}|, \mathsf{depth}, |\mathcal{C}|, 1/\log(|\mathcal{C}|)\}.$$

In this subsection, we will be working on $\mathcal{F}_M$, the monomial subset of $\mathcal{F}$. That is,

$$\mathcal{F}_M = \{|\mathsf{AND}|, |\mathsf{XOR}|, |\mathsf{input}|, |\mathsf{output}|, \mathsf{depth}\}\,.$$

In order to identify the important features $\tilde{\mathcal{F}}$, CostCO carries out a screening step (Lines 4–11), that generates a set of experiments used to compute the importance of each potential feature. A naïve way to find $\tilde{\mathcal{F}}$ is via a full factorial experiment [110], which is a design of size $2^{|\mathcal{F}_M|}$ that enumerates all permutations where each feature is either at its highest value or lowest value. Instead, CostCO uses Placket-Burman design (PBD) [247] for this step because it produces a set of experiments that is small but still capable of finding features that have a significant effect on the runtime. PBD is a fractional factorial design requiring only $4\lceil(n+1)/4\rceil \approx n+1$ measurements for n features, under the assumption that interactions between features (second-order) are confounded with main (first-order) effects. A PBD for $n = 3$ is shown in Table 2.1. The samples collected from PBD are fit and features are filtered according to the confidence intervals computed with eq. (2.3). After the screening step in our running example,

$$\tilde{\mathcal{F}} = \{|\mathsf{AND}|, |\mathsf{input}|\}\,.$$

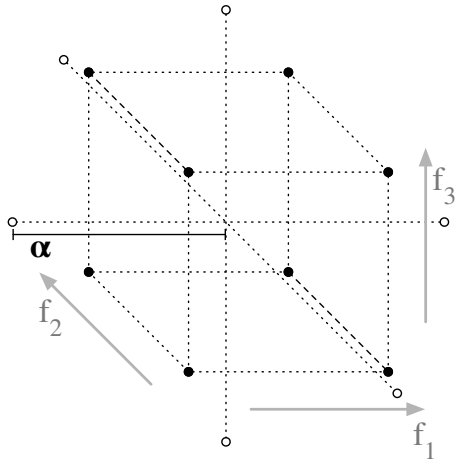


Figure 2.3: CCD for three features. Star points (unshaded points) are α away from the center of the factorial design (shaded points). Note that α can be a different maximum value for each feature.

The screening step using PBD indicates that depth does not have a statistically significant impact on performance, and we drop that from the list of features. This is because AG-MPC is a constant-round protocol, and circuit depth does not impact runtime. |XOR| is dropped because XOR requires no additional communication to compute and |output| is dropped because its effect is both small and limited by the circuit size.

CostCO then carries out a more intensive experiment step using the culled set of features (Lines 12-13). CostCO uses central composite design (CCD) [35], an experiment design methodology that is more expensive than PBD but capable of capturing interactions between features in $\tilde{\mathcal{F}}$ (second-order effects). A CCD is composed of a factorial design augmented with star points that are a distance α from the center [35]. Figure 2.3 shows a CCD for $|\tilde{\mathcal{F}}| = 3$. The extra sampling points help build a second-order model for the features without having to carry out a complete three-level factorial experiment ($3^{|\tilde{\mathcal{F}}|}$ samples). CostCO is also able to reuse data from the PBD experiments (e.g., PBD forms a subset of the cube's vertices in Figure 2.3) instead of re-running those experiments. CostCO then runs these experiments (§2.4.2) to obtain data for computing the abstract and empirical cost models (§2.4.3).

2.4.2 Generating and running experiments

In the previous section, we explained CostCO's two-phase experiment design procedure. Given the set of features and their values each experiment, CostCO still needs to use the user-provided protocol specification (described in §2.3) and the implementation to run one or more profiling experiment instances. In order to do so, given a profiling experiment CostCO synthesizes and runs computational circuits with the required features (Lines 4-5 and 12-13).

Circuits

Similar to many MPC frameworks, CostCO represents computation as a *circuit*. Each circuit is a DAG, where the vertices, which we call gates, correspond to the π primitive operations; and edges represent data flow between gates. An edge $g_i \rightarrow g_j$ means that the output from gate g_i is used as an input to g_j. In addition, CostCO needs to ensure that generated circuits can capture properties like network round-trips, etc. Protocols that are not constant-round, e.g., arithmetic and Boolean ABY [84], have performance that can depend on the sequence of operations that are run. CostCO models this dependency in the circuit as the depth d_n of a primitive operation; this information is supplied as a part of the protocol specification. If g_i represents a gate with depth d_i, then all outgoing edges from g_i are assigned the weight d_i. The depth of the entire circuit is the number of round trips needed to run the computation, which is the equivalent to the length of the longest path in the circuit.

Generating circuits

CostCO generates representative computation in the form of a circuit. The circuit generation component needs to take as input the number of each gate (computational gates and input gates) and the desired circuit depth. Given these parameters, CostCO constructs the circuit by first putting in input gate according to the input size. While building the circuit, CostCO keeps track of: (1) gates that still need to be realized on the circuit and (2) gates with available outputs that can be used as an input to another gate. CostCO selects a gate from the first list and places it on the circuit. Using the second list, inputs are selected for the new gate. CostCO repeats this process until all gates have been realized in the circuit. CostCO achieves the desired circuit depth by keeping track of path lengths and preventing edges that would result in a path length exceeding the desired depth. Finally, CostCO ensures that the output of every gate is either fed into another gate as input or is evaluated as an output.

Note that the input to CostCO's circuit generation component (number of gates and desired circuit depth) are not specified by the user but are automatically generated and varied by CostCO's experiment design component (§2.4.1).

Running circuits

CostCO assumes that the MPC framework can accept inputs in the circuit format described above. CostCO therefore expects that the protocol provider implements the following function:

> `RunCircuit(CircuitFile cf)` Execute the circuit represented by `cf` with the protocol π and report the total execution time and network communication.

CostCO generates circuits and calls the protocol's implementation of `RunCircuit` to profile its performance with circuits generated from the methodology described above. Empirically, implementing `RunCircuit` took less than 200 LOC for every MPC protocol we evaluated.

Types of cost measured

CostCO generates cost models for three types of execution costs: runtime, peak memory usage, and network communication. CostCO opts to measure memory in addition to runtime and network communication because π can experience a significant performance degradation if the peak memory of an execution exceeds available memory. Recent protocols have started trading off more communication rounds for using less memory [333].

2.4.3 Deriving cost models

After choosing and running profiling experiments, CostCO selects terms from the monomial culled features $\tilde{\mathcal{F}}$, asymptotic terms $\mathcal{H}$, and their pairwise products in order to form the final abstract cost model (Lines 15–16). In our running example,

$$\mathcal{G} = \tilde{\mathcal{F}} \cup \mathcal{H} = \{|\mathsf{AND}|, |\mathsf{input}|, |\mathcal{C}|, 1/\log(|\mathcal{C}|)\}$$
$$\mathcal{G}^2 = \left\{ |\mathsf{AND}|^2, |\mathsf{AND}||\mathsf{input}|, |\mathsf{AND}||\mathcal{C}|, \frac{|\mathsf{AND}|}{\log(|\mathcal{C}|)}, \ldots \right\}.$$

We make the observation that only a handful of the terms in $\mathcal{G}$ and $\mathcal{G}^2$ contribute significantly to the cost. Intuitively, this is because many of the pairwise products do not appear in the asymptotic $O(\cdot)$ cost. Thus, the problem becomes one of selecting a *sparse* polynomial out of $\mathcal{G} \cup \mathcal{G}^2$. CostCO uses ordinary least squares regression and leverages the adaptive forward-backward (FoBa) greedy algorithm described in [329] to select a sparse polynomial. FoBa first greedily adds terms to the candidate polynomial in the forward step, halting when the reduction in mean error of adding a feature is less than ϵ. To correct for incorrect features added in the forward step, FoBa's backward step removes a feature if the increase in mean error is less than a factor δ than the last decrease in mean error.

To test how well the polynomial produced by FoBa generalizes, CostCO runs cross-validation on the outputted model. Specifically, CostCO uses leave-one-out-cross-validation where one data point is left out and the leftover data is re-fit to the polynomial. This is repeated for all points in the data. Since FoBa is parameterized by the stopping threshold ϵ, CostCO uses the mean error from the cross-validation runs as a metric when binary searching for the optimal ϵ.

In order to further prevent selecting a polynomial that overfits the data, we only consider a model (polynomial outputted by FoBa) if all of its features have regression coefficients with a positive 95% confidence interval. Under the least squares assumption that observation errors are normally distributed, i.e., $\varepsilon \sim \mathcal{N}(0, \sigma^2)$, the estimated coefficients $\hat{\beta}_i$ from samples X have a $100(1-\alpha)\%$ confidence interval of:

$$\hat{\beta}_i \pm t_{\alpha/2, n-(d+1)} \sqrt{\hat{\sigma}^2 [(X^\top X)^{-1}]_{i,i}}, \tag{2.3}$$

where $\hat{\sigma}^2$ is the mean squared error (estimate of σ^2), t is the Student's t-distribution, and $n-(d+1)$ is the degree of freedom. Note that we make the least squares assumption because MPC protocols are usually deterministic in their execution—dependence on data content would otherwise cause privacy leakage—so errors stem from the execution environment.

2.5 Implementation

We implemented a prototype of CostCO and a hybrid-protocol compiler that uses the generated cost models in ≈5000 lines of code (LOC) of Python. Porting MPC protocols to CostCO took less than 200 LOC per protocol (§2.6.1). Lines of code were counted with `cloc`. Similar to the SPDZ [163] compiler, our compiler takes programs written in a subset of Python. It currently supports six MPC protocols: Arithmetic, Boolean, Yao (and their share conversion protocols) [84]; FastGC [146]; AG-MPC [319]; and SPDZ [73].

We use a program decomposition approach and a protocol conversion assignment approach based on OPA [151]. However, instead of treating the optimal protocol assignment problem as a linear program, we use a randomized greedy algorithm that reduces the search space by avoiding conversions with some probability p if the cost of converting shares to a different protocol outweighs the immediate cost reduction from using a different protocol. This allows us to mix all three protocols (Arithmetic, Boolean, and Yao) whereas the linear program relaxation, though integral, only allows two protocols to be mixed at a time [151]. We leave finding better search algorithms for the optimal program assignment problem for future work.

The cost models provided by CostCO also enable our compiler to consider peak memory consumption when assigning protocols for a program and select between variants of an MPC protocol like FastGC [146], which is a memory-optimized version of Yao [84].

2.6 Evaluation

We evaluate CostCO on 7 MPC protocols. We first report on the ease of porting MPC protocols to CostCO (§2.6.1). We then evaluate the accuracy of our generated models (§2.6.2) and how they compare to current work in cost modeling (§2.6.2). We conclude with results on the cost of running CostCO (§2.6.2).

We ran all evaluations on AWS `c5.4xlarge` instances (16 vCPUs and 32GB RAM) in the same region. All protocols used a deployment with 2 parties except AG-MPC, which used 3 parties. All protocols, when applicable, had their field sizes set to 32 bits. We set the maximum number of a feature per experiment to be the default value of 2^{13}. This value determines the concrete value to set for a feature at the "high" point in a factorial design (described in §2.4.1).

Additionally, we implemented our own hybrid-protocol compiler for ABY based off the techniques in [151]. We replace the cost models used by [151] with the cost models automatically generated by CostCO and find that it arrives at the same hybrid protocol assignments for the applications we tested (GCD, biometric matching, and non-parallelized k-means). In all cases we found that the automatic models generated by CostCO allowed the compiler to perform as well as a compiler that uses hand-tuned, human generated cost models.

	Existing	Using CostCO	
Protocol	Implementation	`RunCircuit`	Spec
ABY [84]	13722	161	62
A	—	141	14
B	—	4	14
Y	—	4	14
A→Y, B→Y, B→A, Y→A, A→B, Y→B	—	12	20
FastGC [146]	9905	139	14
AG-MPC [319]	7269	100	14
SPDZ [73]	39511	185	166
BFV [277]	8854	164	14

Table 2.2: Lines of code to generate cost models using CostCO for each MPC protocol, split between implementing `RunCircuit` and writing the MPC protocol specification. Using CostCO requires additional code that is 1-2% of the existing MPC protocol's implementation size.

2.6.1 Ease of use

We integrated the following MPC protocols to work with CostCO: the three in ABY [84] (Arithmetic (A), Boolean (B), and Yao (Y)), their conversions (A→Y, B→Y, B→A, Y→A, A→B, Y→B), FastGC [135], AG-MPC [319], SPDZ [163], and BFV homomorphic encryption [97]. Table 2.2 shows the lines of code needed to implement the required functionality for the protocol using CostCO (§2.3), as well as the (existing) lines of code in the protocol's implementation.

Running circuits. In order to interface with CostCO, the MPC protocol needs to implement the `RunCircuit` API described in §2.4.2. Implementing `RunCircuit` took about 1-2% of a protocol's implementation size.

Protocol spec. The gates for these protocols were simple to enumerate (Boolean vs. arithmetic operations), resulting in a compact specification of 20 lines or less. The degree bounds on runtime and communication complexity were straightforward to compute from the asymptotic complexity results published for each protocol. The only protocol with a non-polynomial asymptotic term in its cost model was AG-MPC, and its term $|\mathcal{C}|/\log|\mathcal{C}|$, which appears in its reported asymptotic complexity (see Table 1 of [319]), can be input into CostCO in a straightforward manner ($|\mathcal{C}|$ and $1/\log|\mathcal{C}|$). In contrast, we found that manually deriving the cost model to be demanding and error-prone due to requiring further analysis of the protocol. While the term $|\mathcal{C}|/\log|\mathcal{C}|$ is read-

	CostCO				CheapSMC [243]				Opt. Mix [151]			
Protocol	p5	p50	p95	RMSE	p5	p50	p95	RMSE	p5	p50	p95	RMSE
A	1.01	15.0	31.1	37.5	224.7	1163.2	1559.6	2952.4	84.3	238.4	408.8	549.8
B	0.6	10.6	33.5	150.6	19.0	387.8	566.7	4771.5	71.1	314.7	612.8	3632.1
Y	1.2	5.3	13.5	172.9	2.1	28.0	62.9	394.4	313.4	495.2	719.7	5732.3

Table 2.3: Comparison of each cost modeling framework's runtime prediction error (%) and $\sqrt{MSE}$ (ms) on ABY [84]. The percentiles are computed from prediction errors of a set of 100 randomly generated circuits.

ily available in the reported asymptotic complexity, users cannot uncover additional terms in its cost model (such as $|\mathsf{AND}|/\log|\mathcal{C}|$) without more detailed analysis of the protocol.

2.6.2 Microbenchmarks

Model accuracy

To evaluate how well the models produced by CostCO extrapolate, we generate cost models for runtime, network usage, and peak memory usage; and determine their accuracy in predicting each metric on a set of 100 circuits with features of size up to $8\times$ larger (2^{16}) than the largest feature size in the samples used to build the cost model. We sample the test circuits randomly with a fixed seed.

The Root Mean Squared Error (RMSE) as well as the 5th, 50th, and 95th percentile of the relative error of the runtime cost models produced by CostCO for the 3 MPC protocols in ABY [84] are reported in Table 2.3. The relative error as a percent value is calculated as:

$$\left|\frac{\mathsf{actual}-\mathsf{predicted}}{\mathsf{actual}}\right|\times 100.$$

Comparison to existing cost models

CheapSMC [243]. CheapSMC computes cost models by constructing a separate circuit with 1000 operations for every operation in ABY. CheapSMC then records the runtime of each circuit and divides the average over 10 runs by 1000 to get the average per-operation cost. The cost of running a circuit becomes the sum of its individual operation costs. This approach, while efficient and easy to reason about, sacrifices model accuracy because the structure of the circuit is ignored. This can be seen in Table 2.3, where the median accuracy of Yao is $28\,\%$ because while the circuits are evaluated in topological order, the depth of the circuit does not affect the number of communication rounds. In Arithmetic and Boolean, however, the median relative error degrades to $388\,\%$ due to the circuit depth affecting the number of communication rounds.

	CostCO			CheapSMC [243]		Opt. Mix [151]		HyCC [42]	
		Runtime	Memory		Runtime		Runtime		Runtime
Protocol	(#)	(sec)	(sec)	(#)	(sec)	(#)	(sec)	(#)	(sec)
A	34	143.1	62.9	2	14.7	20	35.1	1	0.16
B	34	120.2	69.1	2	8.8	20	80.0	1	0.16
Y	27	111.4	51.0	2	7.6	20	82.1	1	0.16

Table 2.4: Number of experiments (#) and total time (sec) taken by each cost modeling framework to collect samples.

Optimal Mixing (OPA) [151] & HyCC [42]. The cost model used by OPA builds on the methodology of CheapSMC by also considering circuit structure. This is accomplished by running circuits of each gate with different levels of parallelism ($n \in \{1, 2, 5, 10, 25, 50, 100, 200, 300, 500, 800\}$). The cost of a circuit is calculated by finding the number of gates running at each level and taking the appropriate per-gate cost at that level of parallelism. This method suffers from compounding errors introduced from the overhead of executing an actual circuit, e.g., to run a circuit with one gate, the circuit also needs two inputs and one output. HyCC uses a similar cost modeling approach, except with only 4 levels of parallelism. Note that *relative* accuracy matters for protocol selection, which we further investigate in §2.6.3.

Cost of model generation

We now characterize the economic cost of generating our cost models. Our average runtime to generate the cost models (displayed in Table 2.4) for our set of circuits for each of our profiled protocols was in general less than two and a half minutes with an approximate worst-case addition of one minute to profile the memory usage. This leads to a total cost (in our `r5.xlarge` deployment) of model generation of less than $\$0.005$ for each protocol. Although other hybrid-protocol compilers are comparatively cheaper, CostCO is able to automatically generate the experiments to run, saving on manual effort.

2.6.3 Application benchmarks

Comparison to related work

In order to demonstrate the utility of the cost models automatically generated by CostCO, we compared our compiler to prior benchmarks from HyCC and ABY. Below, we describe and discuss each application. The assignments made by each compiler are summarized in Table 2.5 and the runtimes of each application are illustrated in Figure 2.4.

Benchmark (LAN)	Best Known	CostCO	HyCC*	HyCC†	ABY (Manual)
Modular exponentiation	A+B+Y	A+B+Y	—	—	A+B+Y
Biometric matching	A+Y	A+Y	A+B	A+Y	A+Y
Private set intersection	Y	Y	B	Y	—
k-means	A+Y	A+Y	A+B	A+Y	—
DB-Merge	A+Y	A+Y	A+B	A+Y	—
MiniONN MNIST	A+Y	A+B+Y	*	A+Y	—

Table 2.5: Benchmark comparisons with HyCC [42] (automatically generated) and ABY [84] (manually written). HyCC* uses their cost model and protocol selection algorithm, which did not finish running after 2 weeks for MiniONN. HyCC† uses their heuristic approach, which selects the best out of 5 default protocol assignments. The "Best Known" assignment is determined by measuring the runtimes of each assignment and taking the assignment with the minimum runtime.

Modular exponentiation. Two parties want to compute $x^y \mod m$, where one party holds x and the other holds y (all values are 32-bit integers). CostCO computes an assignment similar to the manual ABY [242] assignment.

Biometric matching. One party holds a list L containing m n-dimensional vectors and the other holds an n-dimensional query vector q. The two parties come together to compute the vector $l \in L$ with the shortest Euclidean distance from q. For $(m = 256, n = 4)$, CostCO computes an assignment that is in total $\approx 40\,\mathrm{ms}$ slower than HyCC. This difference is due to HyCC's implementation of biometric matching, which structures the computation of the minimum as a tree, minimizing the depth of the computation.

Private set intersection. One party holds a set S_1 and the other holds a set S_2. The two parties want to compute the intersection of S_1 and S_2 using the standard $O(n^2)$ algorithm. CostCO outperforms the automatic protocol selection of HyCC by $84\,\mathrm{ms}$ but is $595\,\mathrm{ms}$ slower than the hand-selected HyCC protocol. This is due to HyCC running a circuit-level minimization tool which CostCO does not run, which results in a circuit that is 41% smaller.

Database merge. Both parties hold a database with two columns and want to compute the aggregate mean and variance of their merged databases. CostCO outperforms both the automatic HyCC assignment (by $1.28\,\mathrm{s}$) and hand-picked HyCC assignment (by $560\,\mathrm{ms}$). This is due to CostCO also assigning the arithmetic MPC protocol when computing the squares of differences needed to calculate the variance. HyCC only assigns the arithmetic MPC protocol for addition operations when computing the mean and variance.

k-means. Both parties hold datapoints and want to identify centroids in their data using a textbook k-means algorithm [198]. CostCO outperforms the automatic HyCC assignment (by

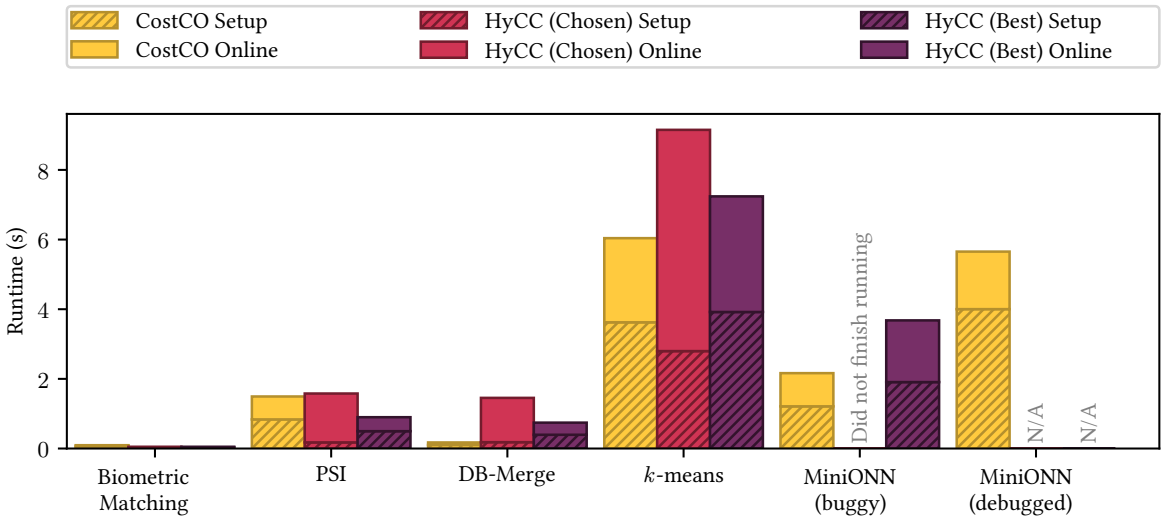


Figure 2.4: Runtime breakdown and comparison of hybrid protocols generated by CostCO and HyCC [42] for various applications. Each bar is broken up into Setup (shaded) and Online runtimes.

3.1 s) and hand-picked HyCC assignment (by 2.5 s). Note that the implementation of k-means in CostCO is more straightforward than HyCC's implementation, which decomposes into multiple inner and outer loops.

Secure prediction. The last application is a machine learning (secure prediction) workload where one party holds the model and the other party holds an input for the model. The two parties want to compute the input's corresponding prediction from the model. We implemented the convolutional neural network described in MiniONN [197] which performs a secure prediction with a neural network trained on the MNIST dataset. HyCC's performance is affected by a bug where certain repetitive calls to a function result in empty circuits (i.e., less work and lower runtime compared to CostCO's debugged assignment). We accounted for this by removing the computations that were removed by HyCC. In this context, HyCC's solver did not finish running and CostCO was able to produce an assignment that was 41% faster than HyCC's hand-selected assignment. The rightmost bar in Figure 2.4 shows the runtime of debugged MiniONN (it does not lose any work), which has a runtime 40% higher than HyCC's buggy MiniONN.

Memory

The cost models produced by CostCO enable our compiler to make more sophisticated decisions by considering each MPC protocol's peak memory usage. To emulate a resource-constrained device, we set an artificial memory budget of 200MB using `cgroupv2` and informed the compiler of the memory limit. This caused CostCO to choose the memory-optimized version of FastGC

MPC Protocol	CostCO	HyCC [42]	OPA [151]
ABY [84]	✓	✓	✓
FastGC [146]	✓	✗	✗
FastGC-Mem [135]	✓	✗	✗
AG-MPC [319]	✓	✗	✗
SPDZ 2.0 [163]	✓	✗	✗

Table 2.6: MPC protocol support across hybrid-protocol compilers.

for our DB merge application. Note that other compilers like HyCC do not model memory consumption, and hence cannot make the correct decision in choosing between an MPC protocol implementation [146] and its memory-optimized version [135].

Extensibility

The most recent hybrid-protocol compilers only support the 3 protocols in ABY [242], which provides MPC for 2 parties and semi-honest security (no party deviates from the protocol). Our compiler is able to use CostCO's cost models to incorporate additional MPC frameworks (SPDZ, AG-MPC, FastGC), which lets it choose MPC protocols for a variety of usage settings, e.g., more than 2 parties and malicious security. Table 2.6 lists the MPC protocol support across hybrid-protocol compilers.

2.7 Related work

2.7.1 MPC compilers

MPC compilers originated as a means to enhance the accessibility of secure computation outside of expert users. Developers often lack the domain-specific knowledge required to employ MPC. These compilers allow developers to describe secure computation through high-level languages, obfuscating the underlying details of the protocols. Fairplay [24] represented the first attempt at compilation, proposing a domain specific language that compiles to a garbled circuit. This work launched the compiler space leading to a wealth of future work. Subsequent single protocol compilers have focused on optimizing performance or scalability (e.g. Sepia [40], Sharemind [30], TinyGarble [291] FastGC [15]), and improvements upon the high-level abstraction (e.g. Picco [330], CMBC-GC [103], Obliv-c [328], ObliVM [196], Wysteria [260]).

As MPC protocols began to specialize, performing especially well for a single type of computation, the need arose for hybrid compilers that leverage each of these categories of protocol. These hybrid protocol compilers have the additional task of partitioning the program and choosing the

optimal MPC protocol for each piece of the computation. As a result, each compiler either relies on hard-coded heuristics or on cost models to guide their decision. Tasty [134] was the first such hybrid protocol compiler and allowed the generation of secure protocols combining homomorphic encryption and garbled circuits. In this framework, the developer hard-codes the protocol to be used for each operation. After Tasty, there have been a number of different approaches to protocol selection. Some frameworks, such as EZPC [51] include hard-coded heuristics for their currently supported protocols and require coming up with new heuristics when integrating new protocols. Other recent works profile the cost of the gates and model the total circuit cost as a sum of the gates (HyCC [42], CheapSMC [243], OPA [151]).

2.7.2 Cost modeling

There have been a number of works on cost modeling outside of secure computation that still hold parallels to the MPC environment. Recently, with the increasing distributed nature of computation, many works have tried to quantify the performance of jobs as a function of a cloud configuration while minimizing the individual experiments required. Ernest [315] aims to enable efficient performance prediction based on the assumption that the related jobs have a predictable structure. The system leverages optimal experiment design [254] to minimize the number of samples required to develop such a predictor. Generic empirical cost modeling frameworks like `trend-prof` [117] receive a representative workload as input from the user and fit to a predetermined type of model (linear or powerlaw). CostCO leverages the security properties of MPC to produce cost models that are independent of data content.

2.7.3 Statistics

Experiment design [174] is a widely applicable statistical tool to determine the features responsible for a value. It allows the user to maximize the understanding gained per-experiment of the relationship between the response variable and different features. Optimal experiment design [254] allows a value to be estimated with minimum variance and bias. In a multi-parameter setting the variance of the parameter estimator is a matrix. Its inverse is denoted as the information matrix. There are different categorizations of optimality based upon minimizing different values related to the information matrix.

There are several instantiations of optimal experiment design. One such category of implementations is iterative experimentation which concerns the development of sequential experiments. Response-surface methodology [138] is an iterative experimentation used to optimize the response by exploring the surface of the response curve. First, in order to explore the curve when far from the optimum, the experimenter uses the method of steepest ascent on a first order model to find the most efficient direction to move. When closer to the optimum, the experimenter switches to a second order model that is more expensive to compute (because more factors equates to more runs), but more accurate. In order to minimize the number of runs required to fit the second-order model, RSM employs the uses of factorial designs such as CCD [35]. A factorial design is a design that enumerates all permutations of feature settings where each feature is

either at its highest value or lowest value. An example of a fractional factorial design is Plackett-Burman design (PBD) [247], which has been shown to be particularly useful in unconstrained configuration spaces [284].

Sparse representations in functional relationships prevent overfitting to the dataset. Traditionally, sparse learning has been performed by using one of three approaches: lasso regularization; forward greedy algorithms, which choose a feature to add to maximize the reduction in the cost function; and backward greedy algorithms, which choose features to minimize the increase in the cost function. FoBa [329] employs the uses of both greedy algorithms showing reduced training error to the three classic choices making sure backward steps don't erase the gain made in forward steps. They ensure that backward steps are only taken when the cost functions increase is less than or equal to half of the decrease of the cost function in earlier forward steps.

2.8 Conclusion

The recent growth in the development of MPC protocols has significantly improved the performance and feasibility of MPC. However, it has led to a zoo of MPC protocols that a prospective user must reason about when choosing a protocol for their workload. CostCO helps compute accurate cost models for different protocols and does so in an automated way. Accurate cost models can aid in the synthesis of efficient hybrid MPC protocols, which could enable the realization of significantly increased employment of practical secure computation.

Chapter 3

LegoLog: A configurable transparency log

3.1 Introduction

Transparency logs are a core building block of many security-sensitive applications across domains such as encrypted messaging [11, 143, 183, 202, 208], Internet of Things (IoT) [7], encrypted backups [74], and code binary distribution [4, 142, 219, 230]. For example, incorrectly issued web certificates undermined the security of TLS traffic for many years because there was no way to verify if an attacker issued a web certificate for a website they did not control. Certificate transparency addressed this problem by providing transparency: the certificate transparency infrastructure deployed in browsers like Chrome and Safari does not prevent certificates from being incorrectly issued, but it does provide visibility into and consistency across all issued certificates, which can be used to detect misbehavior [182]. Key transparency solves a similar problem for end-to-end encrypted messaging [143, 202, 208]: finding the correct public key for a user before sending an encrypted message. Key transparency allows clients to check that the server is advertising the correct key. If the server publishes the wrong key, the transparency log provides incontrovertible evidence of misbehavior.

A transparency log is, at its core, a *verifiable* key-value store. If a client writes a value, other clients querying the server should see the write. If the server is compromised, clients will either still read the correct value or obtain a proof of misbehavior. Transparency logs leverage the common client-server architecture with a single powerful server and many lightweight clients, and augment it by adding a small number of lightweight auditors to validate that the server correctly maintains system state. This makes transparency logs a useful tool for building verifiable key-value stores.

However, choosing a suitable transparency log for an application is not straightforward. Different transparency logs distribute computation and communication differently, leading to diverging performance and security properties. This forces a developer to sift through many transparency log designs to find one fitting her application. For example, specialized transparency log designs exist for web certificates [182], end-to-end encrypted messaging [143, 202, 208], IoT [7], code binary distribution [4, 142, 230], and encrypted backups [74], assuming different client,

server, and auditor device profiles. To complicate matters further, the developer might need a transparency log where no specialized design exists for an application and workload. To achieve good performance, the developer must design a new log specific to the workload and the computational and communication constraints of different entities in their system. Identifying the correct transparency log or, worse, designing a new one, for every application is a massive barrier to adoption, especially for developers without expertise in transparency logs.

To address this problem, we introduce the concept of a *configurable transparency log*, a system that takes as input a workload description and the computational resources of each party and automatically outputs a transparency log design that matches the specification. In this chapter, we present the first configurable transparency log system: LegoLog. Using LegoLog, a developer can describe her application workload without additional expertise and automatically obtain a tailored log.

LegoLog's design is guided by the observation that the vast majority of transparency log designs are built from the same key ingredients. Many designs are inspired by authenticated data structures [231, 296] and are built using data structures permitting efficient proofs that an item was appended to the log or that an item is included in the log. The variation in design then comes from how these ingredients are combined and how the work of verifying the transparency log state is distributed across different entities. We show how to combine these ingredients in a single generalized design that adjusts the underlying algorithm to meet application requirements.

To illustrate its generality, we show how LegoLog derives close approximations of existing, state-of-the-art transparency logs for several different workloads. For example, LegoLog can derive certificate transparency [182], binary transparency [28], multiple key transparency logs depending on application requirements—including CONIKS [208] and Merkle^2 [143]—and a transparency log for IoT [7].

The core challenges in designing LegoLog were threefold: (1) automatically generating a log design tailored to a workload, (2) creating an API for developers to easily specify workload details, and (3) exploring new design points representing different performance or security tradeoffs. When reasoning through a configurable design in LegoLog, we realized that parameters and tasks unnecessarily tied together in prior designs (e.g. update vs. monitoring intervals) could be decoupled. This allowed us to efficiently capture more workloads and, in some settings, to provide stronger security than existing transparency logs (e.g. preventing tampering vs. detecting misbehavior) as we explain below.

Decoupling the frequency of updates and monitoring (§3.3.3). In most existing transparency log designs, clients must come online to monitor for server misbehavior every time the server updates the log [208]. However, some applications require writes to propagate quickly with clients that are not regularly online (e.g., a user whose phone is stolen might want to update her messaging keys right away). We observe that we can decouple the update frequency and monitoring frequency. This insight allows us to explore new design points that allow for low update latency even if clients are not constantly online. In particular, we show how to trade off the frequency with which clients come online to perform checks for the cost of a lookup in the transparency log.

Allowing clients to detect misbehavior early (§3.3.3). In many existing transparency log designs [28, 143, 182, 184, 208], clients monitor for server misbehavior. If the server misbehaves, then the client obtains proof, but possibly only after an irreversible action has taken place due to an incorrect value (e.g., for a cryptocurrency directory or end-to-end encrypted messaging). In LegoLog, we explore a different security tradeoff by allowing clients who monitor regularly to detect server misbehavior *before* other clients use the incorrect values. This way, clients can not only detect server misbehavior, but also protect against its ill effects, provided that clients come online regularly. LegoLog achieves this while supporting low-latency updates, meeting both the performance and security requirements of an application.

Supporting offline clients (§3.3.4). In some applications, clients may disconnect for extended durations. When a client comes back online, it should be able to batch monitoring checks to perform them more cheaply. We show how to batch checks so that the client only has a small amount of work to do when it comes back online to detect misbehavior during the period it was offline. We draw inspiration from prior work on aggregating verification work [143]. We integrate these ideas in a way that is compatible with the other design parameters in our system without harming performance.

Designing an expressive developer API (§3.2.3). While LegoLog uses a set of internal system parameters to derive the correct transparency log construction, the developer has no intuition for how to set these parameters. We introduce an API that allows developers to specify the application workload and the computational and communication constraints of the clients and auditors. From that, the LegoLog's planner outputs the number of cores and bandwidth required for the central server along with the internal system parameters for running LegoLog for the specified application (§3.4).

Evaluation summary (§3.5). To see how LegoLog generalizes to different applications, we empirically evaluate it end-to-end for three specialized transparency logs [28,143,208]. We also show that LegoLog can express six different applications, and we compare their asymptotic complexities to existing specialized transparency logs. We find that configurability does not come at the cost of performance: LegoLog can capture a variety of applications while performing comparably to existing, special-purpose transparency logs.

Limitations. While LegoLog captures a wide variety of transparency log settings, it does not capture those based on heavyweight cryptographic primitives, such as SNARKs [58, 305, 306], bilinear accumulators [299], or vector commitments [191]; these tools improve asymptotic performance, but generally have comparatively high overheads in practice. Also while we output the amount of server storage required, we do not optimize for it, as in the case of SEEMless [53] and Parakeet [202]; we leave the problem of optimizing for server storage to future work.

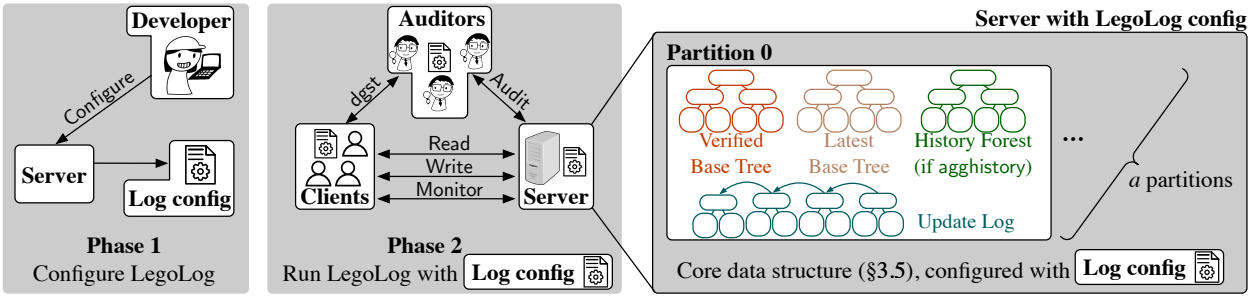


Figure 3.1: LegoLog overview. In Phase 1, the developer configures the LegoLog deployment for their application. The planner then generates a deployment configuration satisfying the requirements. In Phase 2, LegoLog runs with the configuration from Phase 1. Clients read, write, and monitor identifier-value pairs by interacting with the server. Auditors check the server correctly maintains the transparency log. The deployment configuration configures LegoLog's core data structure (§3.3.5), and enables different possible performance characteristics.

3.2 Overview

3.2.1 System architecture

In a LegoLog deployment (Figure 3.1), the central server provides an identifier-value store, and the clients and auditors enforce certain integrity guarantees on this store. We consider two types of clients: readers and writers. In some deployments (e.g. key transparency), clients are both readers and writers; in other deployments, clients perform only reads or only writes (e.g. in a producer-consumer model). Throughout this chapter, we refer to "identifier-value" pairs rather than "key-value" pairs to avoid confusion with cryptographic keys. We now describe the entities in a LegoLog deployment.

Readers. Lightweight clients that read identifier-value pairs.

Writers. Writers are clients that write identifier-value pairs. Writers monitor a subset of the identifiers (the identifiers they can write to) to ensure the server provides the correct value to readers for any identifier in the writer's set. Writers should be lightweight and may go offline for some time.

Auditors. Auditors check that the server maintains the transparency log correctly. Auditors also service client requests for the log digest, allowing clients to ensure that the server answers queries relative to the auditors' view of the log.

Server. The server maintains the transparency log and services read and write requests from clients. The server also sends data to the auditors that allow the auditors to ensure that the server is maintaining the log correctly.

3.2.2 System API

We now describe the API that LegoLog provides.
Readers can execute the following API call with the server:

- $\mathsf{Read}(\mathsf{config}, \mathsf{dgst}, \mathsf{id}, t) \to \{\mathsf{val}, \perp\}$: To read the value of id relative to a digest dgst at epoch t for a deployment with configuration config, the reader queries the central server for the value val and a proof. If the proof verifies relative to the auditors' digest, the reader outputs val; otherwise, it outputs $\perp$.

Writers then execute the following API calls with the server, as writers need to write values to identifiers, and then monitor the value of the identifiers they have write privileges for:

- $\mathsf{Write}(\mathsf{config}, \mathsf{sk}, \mathsf{id}, \mathsf{val}, t) \to \{0, 1\}$: To write an $(\mathsf{id}, \mathsf{val})$ pair in epoch t for a deployment with configuration config, the writer sends a request to the central server. The server updates its state and the client outputs "1" if $(\mathsf{id}, \mathsf{val})$ was correctly included and "0" otherwise. If this is the first write to id, the server associates id with the public key corresponding to sk. If this is not the first write to id, the write is only successful if sk matches the secret key used on previous successful writes to id. As in prior work [143], this secret key ensures that only the owner of the secret key can update the value corresponding to id.
- $\mathsf{Monitor}(\mathsf{config}, \mathsf{dgst}, \mathsf{id}, \mathsf{val}, t) \to \{0, 1\}$: To verify that id still maps to val relative to digest dgst at epoch t in a deployment with configuration config, the writer queries the central server for a proof. If the proof verifies relative to the auditors' digest and asserts that id maps to val, then the writer outputs "1"; otherwise, it outputs "0".

Auditors also run the following algorithm periodically:

- $\mathsf{Audit}(\mathsf{config}, \mathsf{dgst}_{\mathsf{old}}, \mathsf{dgst}_{\mathsf{new}}, \pi) \to \{0, 1\}$: The auditor takes in configuration config, an old digest $\mathsf{dgst}_{\mathsf{old}}$, a new digest $\mathsf{dgst}_{\mathsf{new}}$, and a proof π from the central server. The auditor outputs "1" if π certifies that $\mathsf{dgst}_{\mathsf{new}}$ correctly extends $\mathsf{dgst}_{\mathsf{old}}$, and "0" otherwise.

3.2.3 Developer API

LegoLog exposes the following API to developers:

- $\mathsf{Configure}(\mathsf{workload}, \mathsf{reader}, \mathsf{writer}, \mathsf{auditor}) \to (\mathsf{server}, \mathsf{config})$: Given a description of the workload, the reader specification reader, the writer specification writer, and the auditor specification auditor, output a server specification server, as well as the system parameters config that are used by the readers, writers, server, and auditors.

LegoLog allows a developer to set parameters to derive a transparency log suited to her workload (Table 3.1 lists the parameters). Given a workload specification, LegoLog outputs the necessary server capacity and the configuration.

If a developer makes a mistake when specifying the workload or there is an unexpected change in the workload or client connectivity, LegoLog still provides the standard transparency log security guarantees, but the performance and additional security of active prevention may be affected. For example, performance could be affected by incorrect server requirements or a suboptimal update period or number of partitions. Security could be affected if clients cannot

		INPUTS
Workload	N	Number of users
	t_W	Max time for writes to propagate (s)
	t_R	Max time for reads to be processed (s)
	burst_R	Max number of reads that can be handled immediately in t_R (reads/t_R)
	burst_W	Max number of writes that can be handled immediately in t_W (writes/t_W)
Read	bw_R	Reader bw utilization (bytes/hr)
	cores_R	Reader CPU utilization (% of CPU cycles, averaged over 1 hour)
Write	bw_W	Writer BW utilization (bytes/hr)
	cores_W	Writer CPU utilization (% of CPU cycles, averaged over 1 hour)
	t_{refresh}	Background refresh rate, frequency at which writer queries server (s)
	t_{offline}	Max writer offline time staying in comm and utilization bounds (s)
Audit	bw_A	Auditor-server BW (bytes/hr)
	cores_A	Number of auditor cores
		OUTPUTS
Server	bw_S	Server BW utilization (bytes/s)
	cores_S	Number of server cores
	$\mathsf{storage}_S$	Server storage (bytes)
Sys params	t_v	Length of verification period (s)
	t_u	Length of update period (s)
	a	Number of partitions
	verifier	Party responsible for verification period checks (client or auditor)
	agghistory	Whether or not to aggregate base trees for efficient delayed verification (T/F)

Table 3.1: Inputs and outputs to the LegoLog configuration algorithm exposed to the developer.

come online every verification period: in this case, they can only detect server misbehavior after the fact.

Example usage: Key transparency

To build intuition for how a developer would characterize a workload, we take key transparency as a running example [165, 208]. In this application, the transparency log holds a mapping of usernames to public keys (e.g. for end-to-end encrypted messaging), and clients monitor the log to ensure that the mapping of usernames to public keys is maintained correctly.

Say that a developer wants to configure LegoLog for key transparency in Signal. We now show an example of how the developer can specify the workload to LegoLog. Signal reportedly

has roughly 40 million active users as of January 2021 [298], with roughly 125 million downloads in 2021 for an average of 342K downloads a day [288]. Therefore we can set the number of users N =125M. When a user moves to a new device, it is important for security to update their public key quickly, and so we can set the maximum time for writes to propagate t_W to 15 seconds. Reads must be processed quickly so that users can message other users right after downloading Signal or importing a new contact, and so we set t_R to 30 seconds (it will take the user some time to type a message before trying to send). The number of downloads is roughly equivalent to the number of key updates (i.e. the number of writes). The average number of key updates every 15 seconds is 59, and so we set our application to handle up to a burst of up to 75 writes every 15 seconds by setting $\mathsf{burst}_W = 75$. The choice of how large of a burst to accommodate is somewhat arbitrary and depends on how much the developer wants to minimize the cost of the service vs. how important write latency is in the event of a burst (if there are more requests than can fit in the burst, the client can simply try again later).

The average Android user has roughly 5,000 contacts synced to their device [285], and so we conservatively assume that every contact uses Signal and that users need to query the transparency log for a user's public key roughly once a year (assuming that users buy new phones roughly once a year). This results in 200B read requests over a year, or on average 190K read requests every 30 seconds. If we want to accommodate a burst of 300K read requests every 30 seconds (again, the choice of burst relative to the average number of requests is somewhat arbitrary), we set $\mathsf{burst}_R = 300{,}000$.

In key transparency, every client is both a reader and a writer (clients read public keys for their contacts, and write their own public key). We can then set the reader and writer parameters in the same way based on the acceptable communication and computation costs for LegoLog in a Signal deployment. We use these communication and computation constraints to set the max bandwidth bw_R and bw_W, CPU utilization cores_R and cores_W, and background refresh rate t_{refresh}. Based on how long clients typically go offline, we can set the max offline time t_{offline}. We also provide as input the computation and communication that the auditors can devote to LegoLog (cores_A and bw_A, respectively). Based on these inputs, LegoLog outputs the required server bandwidth (bw_S), number of cores (cores_S), and storage ($\mathsf{storage}_S$), as well as the internal parameters for running the system.

3.2.4 Security guarantees

We now describe the security guarantees that LegoLog provides. Like other transparency logs, LegoLog defends against a malicious attacker that has compromised the server, any number of clients, and all but one auditor that can communicate with the clients and the server. Against such an attacker, LegoLog ensures that the client can *detect* if the attacker tampered with any $(\mathsf{id}, \mathsf{val})$ pairs the client is monitoring.

Furthermore, if the client comes online at regular intervals, the client can *prevent* this tampering. Transparency logs traditionally only allow a client to detect when the server has misbehaved. In LegoLog, if the client comes online periodically, the client can detect if the server will advertise the wrong value for a given identifier before any clients use this incorrect value. In this event, the

client can report this misbehavior before the wrong values are used, which is valuable for applications where clients take irreversible actions based on values in transparency logs (e.g. by sending cryptocurrency transactions or sending sensitive information via encrypted email or messaging). Note that the client can only report this misbehavior if it has access at the time to some public forum of sorts where it can notify other clients of the server's misbehavior in a timely manner.

Like other transparency logs [53, 143, 208], we do not guarantee availability: the server can refuse to provide service. Some auditors can refuse to provide service, but as long as clients can continue contacting at least one honest auditor, they can make requests and have strong integrity guarantees.

If a client does not monitor its identifier-value pairs, there are no integrity guarantees for this client's data. Also, a malicious client (who could collude with the server) cannot affect the identifier-value pairs of an honest client.

Security game. We define security for LegoLog via a security game (Experiment 3.5 in §3.9.2). At a high level, the challenger plays the role of honest clients and an honest auditor. The adversary chooses the identifier-value pairs that the clients can write and monitor each round. At the end of the game, the challenger monitors and then reads all identifier-value pairs that it has written. The adversary wins if all auditing and monitoring checks pass while ensuring one of the reads at the end of the game outputs a value that does not match the client's latest write. This game captures log tampering detection and prevention. For detection, the adversary can choose which identifier-value pairs are not monitored and the challenger must still detect any tampering when it monitors all identifier-value pairs at the end of the game. For prevention, the client does not monitor after its last read at the end of the game, so any misbehavior must be caught by prior monitoring or auditing checks. Note that the game also captures correctness as client writes and reads values according to a reference dictionary.

Definition 3.1. Let Π be a configurable transparency log protocol and $\mathcal{C}$ be the set of all possible configurations. Then we say that a configurable transparency log protocol Π is secure if for all non-uniform PPT adversaries $\mathcal{A}$ in Experiment 3.5 with advantage $\mathsf{TLogAdv}$ and for all possible configurations $\mathsf{config} \in \mathcal{C}$ with sufficiently large security parameter λ,

$$\mathsf{TLogAdv}[\mathcal{A}, \Pi](\mathsf{config}) \leq \mathrm{negl}(\lambda)$$

Theorem 3.1. *When instantiated with a signature scheme that provides existential unforgeability under chosen-message attack (EUF-CMA) and a collision-resistant hash function, LegoLog is secure according to Definition 3.1.*

Proof sketch. We must reduce the advantage of the adversary $\mathcal{A}$ in our security game to the advantage of an adversary against an EUF-CMA signature scheme or of an adversary against the collision-resistant hash function. We prove this by first constructing a hybrid game where the adversary wins if the challenger outputs an old identifier-value pair as a result of the reads at the end of the game, and prove that the difference in adversary advantage $\mathcal{A}_\triangle$ between the two games is negligible when using a secure signature scheme by constructing an adversary $\mathcal{B}$ that uses $\mathcal{A}_\triangle$ to win in the EUF-CMA game. We then prove that an adversary $\mathcal{A}$ that wins this hybrid

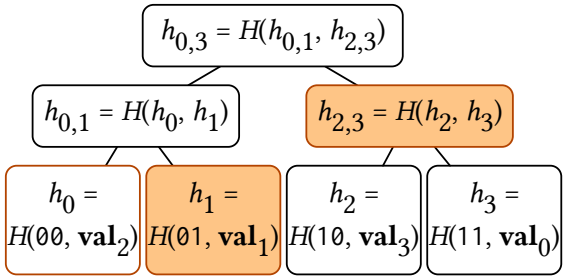


Figure 3.2: An example prefix Merkle tree containing 4 leaf nodes in lexicographic order. $H(\cdot)$ is a hash function and val_i is a node added at time i. An inclusion proof for the identifier-value pair ($\texttt{00}$, $\mathbf{val}_2$) consists of the nodes shaded in orange, which are sufficient to reconstruct the root hash $h_{0,3}$.

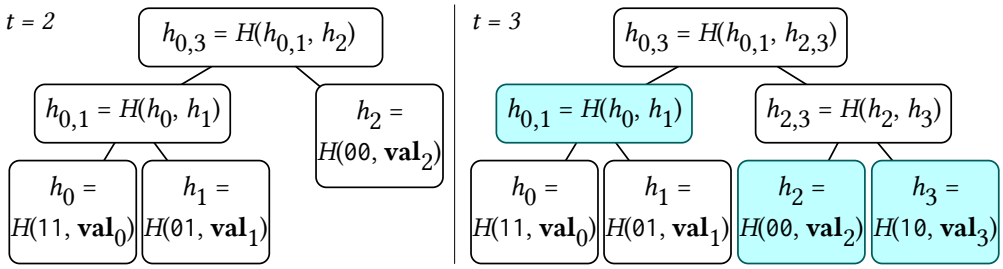


Figure 3.3: An example chronological Merkle tree that contains 3 leaf nodes at time $t = 2$ and 4 leaf nodes at time $t = 3$. $H(\cdot)$ is a hash function and val_i is a node added at time i. The leaf nodes are ordered by the time the node was added. The extension proof to prove that the $t = 3$ tree extends the $t = 2$ tree consists of the nodes shaded in blue, which commit to the same values in the $t = 2$ tree and are sufficient to calculate the root hash $h_{0,3}$ in the $t = 3$ tree.

game can be used to construct an adversary $\mathcal{C}$ that can find a collision in the hash function. The full proof can be found in §3.9.3.

3.3 System design

3.3.1 Building blocks

Merkle trees. LegoLog uses binary Merkle trees [210], which are hash-based data structures that enable efficient verification of contents in the tree. Each leaf node contains a hash of the data it holds, and every non-leaf node contains a hash of its children. In transparency logs, it is common for the leaves of a Merkle tree to be either organized in prefix (lexicographic) order or chronological (time of append) order.

Prefix Merkle tree. organize their leaves in lexicographic order. This organization enables efficient lookups because a client knows the exact leaf to request an inclusion proof for when looking up an identifier-value pair. An example prefix Merkle tree and inclusion proof (shaded in orange) is depicted in Figure 3.2. A proof of exclusion for some j is a proof of inclusion to the longest-prefix match of j currently in the tree. That node will either be a leaf node $i \neq j$ that matches the first few bits of j or an empty node that is a prefix of j [208]. While prefix Merkle trees provide efficient lookups, it is inefficient to prove that a newer prefix tree did not remove any contents of the old one. In existing transparency logs based on prefix Merkle trees, clients must monitor their identifiers every epoch to check for server equivocation [208].

Chronological Merkle trees. have leaves organized by time the leaf was appended to the tree. Unlike prefix Merkle trees, chronological Merkle trees have inefficient lookups because a client may need to search through all the leaves to find a given identifier-value pair. Figure 3.3 shows a chronological tree at two points in time. It is efficient to prove that a chronological Merkle tree extends an older tree (i.e., is append-only). In Figure 3.3, the extension proof consists of the nodes shaded in blue. A detailed algorithm for generating the extension proof is found in §2.1.4.1. of [182].

3.3.2 Sharding chronological trees

Chronological and prefix Merkle trees have complimentary properties that are both desirable. Chronological trees have efficient extension proofs that prove the append-only property of the tree [182], but do not support efficient lookups. On the other hand, prefix trees support efficient lookups but require users to monitor their identifier-value pairs every epoch to ensure that the server has not equivocated [208]. Ideally, we would like a system that supports *both* efficient lookups and extension proofs so that users do not have to monitor their own identifier-value pairs every epoch.

Chronological trees already support efficient extension proofs and do not require clients to monitor their key every epoch. Due to its leaves being ordered by append time, a lookup requires the client to look through all the leaves in the tree. We observe that we can reduce the cost of lookups by reducing the size of the chronological tree by sharding identifier-value pairs across multiple chronological trees (Figure 3.4). A lookup for id would entail hashing id to find its corresponding chronological tree and linearly scanning through all of its leaves for relevant updates to id. Adding another chronological tree shard means that auditors have another extension proof to check every epoch.

If u is the number of updates to identifier-value pairs in the transparency log, a is the number of partitions, and l is the number of lookups/epoch, the work done by an auditor is equal to the work of checking an extension proof times the number of chronological trees of size u/a:

$$\text{work}_A = a \max\left(1, \log_2\left(u/a\right)\right)$$

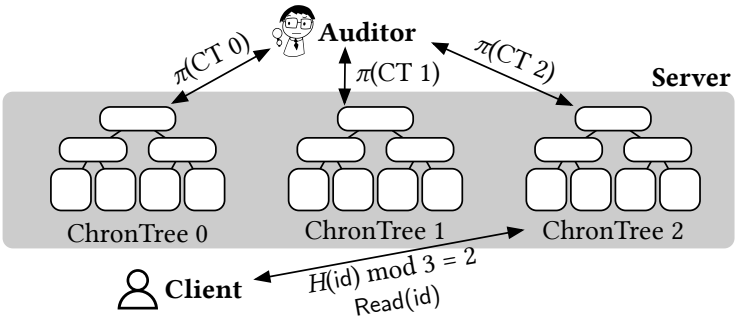


Figure 3.4: Sharding chronological trees. When looking up an id, the client calculates the partition the id belongs to and looks up id in the corresponding chronological tree. Auditors are responsible for verifying extension proofs for all of the chronological trees.

and the work done by clients for lookups is equal to the number of lookups times the work of checking each leaf node in the chronological tree of a partition (size u/a):

$$\text{work}_C = lu/a$$

The total work is the sum of auditor and lookup work:

$$\text{work}_T = a \max\left(1, \log_2\left(u/a\right)\right) + lu/a$$

Sharding chronological trees provides a configurable system parameter—the number of partitions—that affects client and auditor work. This parameter can be tuned for different objectives. For example, the optimal number of shards that minimizes total work can be found by taking the partial derivative $\partial/\partial a$ of work_T and solving for its roots. By sharding chronological trees, we are a step closer to achieving a system where clients do not have to monitor their keys (due to efficient chronological tree extension proofs) and where lookups are efficient (requiring u/a rather than u work).

3.3.3 Compacting chronological trees

As time goes on and the number of updates increases, client lookup work continues to increase because a lookup entails a linear scan through all the leaves in the chronological tree. In order to reduce lookup time, we can routinely roll up the updates into a prefix tree after some time. We call this prefix tree the *base tree* and the chronological tree holding updates the *update log*. In each partition, LegoLog keeps a core data structure (CDS) consisting of a base tree and an update log.

The client must now periodically monitor its own identifier-value pairs and ensure that its updates are properly incorporated into the latest base tree. Our key insight is decoupling when a value is updated and when a value should be monitored (verified) by the client. The frequency

the client needs to monitor its values is controlled by the *verification period* t_v and the update propagation time is controlled by the *update epoch* t_u. During each verification period t, the client verifies its values in the latest base tree. The previous base tree, verified during period $t-1$, and the update log containing updates since period $t-1$ are used to look up values during verification period t. A verification epoch can contain multiple update epochs: longer verification periods correspond to more update epochs (↑ lookup work) and less frequent monitoring (↓ monitoring work).

Update log optimization. To further decrease lookup cost, the server can roll its updates every update epoch into a prefix tree and append the root hash of the prefix tree to the update log. If the client made an update, the client should check that its update appears in the prefix tree for that update epoch. A lookup for id will consist of 1) looking up id in the base tree and 2) looking up id in every update epoch's prefix tree. Figure 3.5 shows a server partition that compacts chronological trees and optimizes the update log.

Preventing tampering vs. detecting misbehavior. Clients look up values in the base tree from the previous verification period and the tree in the current verfication period can be thought of as being in 'staging'. If the client comes online periodically, the client can detect if the server will advertise the wrong value for a given identifier before any clients use this incorrect value. In this event, the client can report this misbehavior before the wrong values are used. Depending on the security guarantees and performance properties that the developer desires for her application, the planner will set t_v accordingly (§3.4.2) to achieve the desired configuration.

3.3.4 Supporting offline clients

Ideally, clients come online every verification period and monitor to detect server equivocation. However, in workloads with clients that are offline for multiple verification periods, clients will need to check the base tree in every verification period it was offline for. To support *efficient* delayed verification for clients that are offline for multiple verification periods, we take inspiration from the ideas of Merkle^2 [143] and show how to incorporate them into LegoLog, which is general-purpose and configurable for different workloads.

History forest. We construct a chronological forest of base trees, which we call a *history forest*. This data structure is related to the chronological forest in Merkle^2 [143] where leaves are identifier-value pairs arranged in historical order and each intermediate node contains a prefix tree representing the subtree rooted at that node. Our data structure differs in that the leaves are base trees instead of identifier-value pairs and intermediate nodes contain the most recent base tree of their child nodes (i.e., the right child node). Nodes are added to the forest in an immutable manner; once added, a node cannot be changed. As a result, intermediate nodes can only be created when their subtree is full, so the data structure is a *forest* of trees rather than a single tree. Figure 3.6 shows an example history forest with 7 base trees (7 verification periods) and 3 history trees. Adding the next base tree BT_8 to this history forest first involves creating a leaf

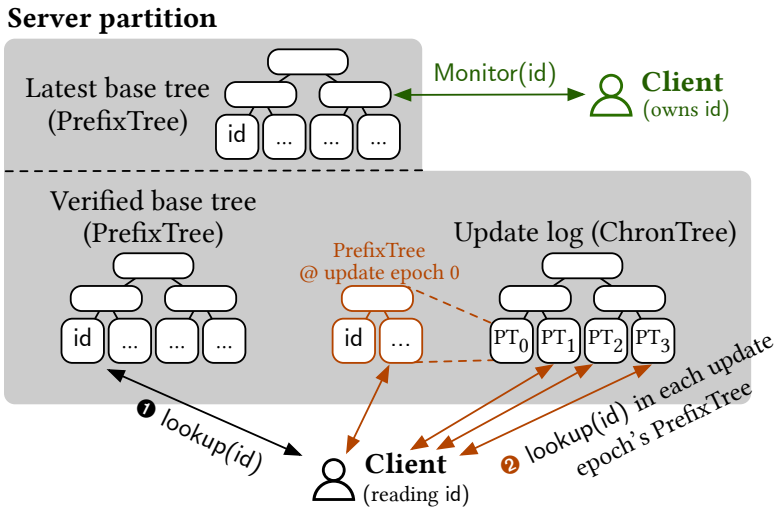


Figure 3.5: A server partition that compacts chronological trees contains a verified base tree, the latest base tree, and an update log. A base tree is a prefix tree that routinely rolls up updates. The update log tracks changes after the verified base tree, and is a chronological tree of the root hashes of prefix trees for each update epoch. A client looks up id ❶ in the verified base tree and ❷ in each update epoch's tree. The client that owns id verifies that the latest base tree contains the correct value.

node representing BT_8. Then, an intermediate node can be created as the parent of the leaf nodes BT_7 and BT_8. This results in HT_3 being a full history tree with 2 leaves. Because HT_2 is also a full history tree with 2 leaves, we can create an intermediate node as the parent of HT_2 and HT_3, effectively merging HT_2 and HT_3 into one full history tree HT'_2 with 4 leaves. Finally, since HT_1 is a full history tree with 4 leaves, we can create an intermediate node as the parent of HT_1 and HT'_2, resulting in a full history tree with 8 leaves where the root contains BT_8. Note that adding BT_8 did not modify any existing nodes.

A monitoring proof in the history forest during verification period t_m for update u made during verification period t_u consists of the path from t_u to the history tree root containing t_u in the history forest at time t_m. For each node in the path, the client looks up the value of u in the corresponding base tree. Because the history forest is append-only and its nodes do not change after being added, the client need only verify a given verification period's base tree once. Figure 3.6 shows a monitoring proof consisting of nodes (green) for an update made during verification period 1 (corresponding to BT_1) and monitoring during verification period 7. The client ensures that the update has propagated to BT_1, BT_2 and BT_4.

The monitoring proof alone is insufficient to detect whether the server has removed an update, as it does not check every prefix tree. For example, in Figure 3.6, the monitoring proof for an update made in BT_1 does not check the base trees BT_3 and BT_{5-7}. Instead, during a *lookup*,

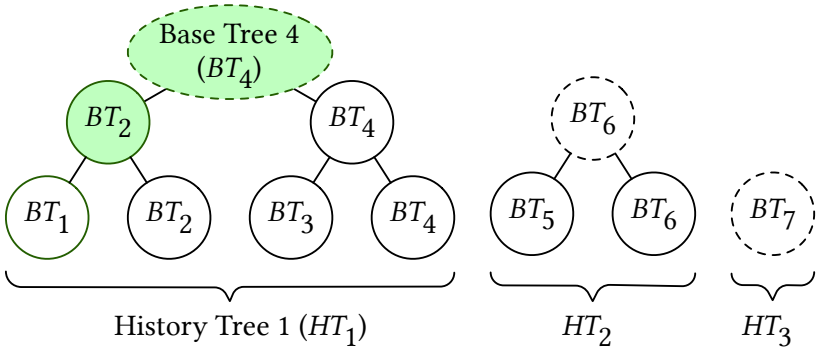


Figure 3.6: A history forest of base trees from 7 verification periods. There are 3 history trees, and each history tree's leaves are arranged in chronological order. A monitoring proof for an update in BT_1 consists of the nodes shaded in green. A lookup consists of looking up a value in each history tree's root (nodes with dotted outline). The lookup and monitoring proof both include the root (BT_4) of HT_1.

our system must ensure that at least one base tree that the client looks up overlaps with a base tree verified in a monitoring proof. LegoLog achieves this by having the client look up the value in *every* history tree's root base tree. Figure 3.6 shows a lookup during verification period 7 consisting of each history tree's root (nodes with dotted outline): BT_4, BT_6, and BT_7. Observe that BT_4 overlaps with the monitoring proof (nodes shaded in green), so the client can detect that the server removed the update if it is present in BT_4 but missing in BT_6 or BT_7.

Our history forest data structure lends itself to an efficient monitoring proof for clients that are offline for multiple verification periods and trades off a higher lookup cost for this increased monitoring efficiency. A similar $\log n$ (where n is the number of verification periods) checking scheme is proposed in a design draft of Google's new KT system [165], but mapping the version checks to checking a path in the history tree is not stated explicitly in the document.

The optionality of using a history forest allows LegoLog to be configured for different workloads. For a workload where clients are offline for long periods of time, LegoLog can be configured to use the history forest, resulting in a lookup cost of $O(\log^2 n)$ and a monitoring cost of $O(\log^2 n)$. However, for a workload where clients are regularly online, LegoLog can be configured to not use the history forest, resulting in a lookup cost of $O(\log n)$ and a monitoring cost of $O(\log n)$.

Signing updates. While monitoring proofs prevent the server from omitting values, they do not prevent it from maliciously inserting values. To prevent this, we use signatures, a solution inspired by previous work [143]. For a given id, and verification period t, the writer first registers a verifying key on the log. In order to ensure that this is the first time a verifying key is being registered for this id, the writer must verify a proof that id is not present in the base tree at verification period $t-1$ (BT_{t-1}). Upon verifying this proof, the writer inserts the verifying key for id into the log. Then, for each update to id, the writer attaches a signature. When reading id,

the client can look up the verifying key in the base tree and use it to verify the signature.

3.3.5 Putting it together: Core data structure

LegoLog partitions identifiers and in each partition, uses a configurable core data structure (CDS). A CDS is comprised of a digest for the previous verification period (forming a hash chain between verification periods) and *three* Merkle trees (Figure 3.5): (1) a verified based tree, (2) the latest base tree (that is being verified), and (3) an update log. Base trees are prefix Merkle trees mapping identifiers to values. The update log is a chronological Merkle tree tracking updates to the verified base tree. The leaves of the update log are prefix Merkle trees containing the updates of an update epoch.

LegoLog's CDS can be configured to support efficient delayed verification by aggregating the base trees into the history forest described in §3.3.4. When the agghistory parameter is true, LegoLog's CDS contains a history forest of base trees instead of a verified base tree and the latest base tree.

3.3.6 LegoLog protocol

We now describe the LegoLog protocol. We include a full description of the protocol in §3.8.

Reading. In order to read the value of id relative to a digest dgst obtained from an auditor, ❶ the client first computes the partition for the queried id: $H(\mathsf{id}) \mod a$. ❷ Then, the client retrieves the root of the verified base tree for that partition from dgst. ❸ If the client does not have the verifying key for id, it first looks up the verifying key in the verified base tree (via inclusion proofs). ❹ The client then looks up id in the verified base tree (via inclusion proofs). ❺ For each update epoch since the start of the verification period, the client receives an inclusion or exclusion proof for id in the update log from the server. The client can retrieve exclusion proofs from the most recent update epochs until hit the most recent inclusion proof, or get all of the exclusion proofs if it wants the full history of changes to id. ❻ The client outputs $(\mathsf{id}, \mathsf{val})$ where val is the latest value with a signature that successfully verifies with the verification key.

History forest case. If $\mathsf{agghistory} = \mathsf{true}$, the client follows ❶–❸ as previously described, except in ❷, instead of retrieving a single base tree root, it gets the base prefix tree corresponding to the root of each history tree in the history forest. ❹ The client then looks up id in each base prefix tree (via inclusion proofs). The client follows ❺–❻ as previously described.

Writing. To update a value, the client ❶ signs the update with its signing key and sends the $(\mathsf{id}, \mathsf{val})$ pair and signature to the server. ❷ The server appends to the $(\mathsf{id}, \mathsf{val})$ pair to the partition (determined by $H(\mathsf{id}) \mod a$). ❸ The client then checks that the update was correctly incorporated into the update log in the next update epoch (via inclusion proofs).

Monitoring. We describe the cases where the client or the auditor performs monitoring.

Clients perform monitoring. If the client is responsible for monitoring values it has written (verifier $=$ client), ideally the client comes online every verification period, finds the partition corresponding to the id it wants to monitor by computing $H(\mathsf{id}) \mod a$, and queries the latest base tree in that partition to make sure the value is correct. Clients have until the next verification period—when the latest base tree becomes the next verified base tree—to check this value.

History forest case. If agghistory $=$ true, then when the client comes online, ❶ it queries the auditors for the roots of the history forest in the partition corresponding to id. ❷ In each update period where the client made an update to id, it retrieves an inclusion proof from the LegoLog server of the base tree root for each node on the path from the update epoch to the root in the corresponding history tree. An example path is shown in Figure 3.6. ❸ For each base tree root the client receives that the client has not checked in the past, the client will check the value of id in that tree by requesting an inclusion proof from the server.

Auditors perform monitoring. If the auditor is responsible for monitoring values on behalf of the users (verifier $=$ auditor), then in each verification period, the auditors check that the latest base tree in each partition corresponds to the contents of the update log applied to the previous base tree.

Auditing. In each update epoch, the auditors check that the update log in each partition is append-only by requesting extension proofs from the server (a total update logs are checked). In each verification period t_v, the auditors keep track of the digest, check it amongst themselves, and serve it to clients.

History forest case. If agghistory $=$ true, auditors check the history forest of base trees in each partition is append-only.

3.4 LegoLog planner

LegoLog's configurability comes from an automatic planner that generates the transparency log design for a given application workload description. The planner takes as input a workload description and specifications of the readers, writers, and auditor and outputs internal system parameters and a central server specification (Table 3.1). By automatically tuning the internal system parameters, it can customize the log for the specified performance and security needs.

3.4.1 Specifying entities

Requiring the developer to directly specify device properties (e.g., CPU, target CPU utilization, network bandwidth) can be difficult or insufficient for determining the optimal configuration. For example, the developer may not know a mobile client's exact CPU utilization because battery life is often the priority [12]. They often want to optimize for an objective more complex than a single target metric. Sometimes this means minimizing client-side CPU utilization for mobile clients; sometimes it means reducing total system workload.

Instead, we allow developers to specify the *type* of device that the readers, writers, and auditors run on. The device type translates to a heuristic about the device's properties and a corresponding optimization objective. For example, if a developer specifies that readers run on mobile phones, the planner will minimize client-side work.

The planner must also know the auditor's compute budget. It determines this by running a benchmark on a representative auditor device, which measures the device hash rate. This information and the developer's target auditor CPU utilization sets a constraint on the maximum auditor work.

3.4.2 Determining system parameters

We now describe how LegoLog's planner maps developer-specified parameters to internal system parameters (Table 3.1).

Update period. The update period t_u directly corresponds to t_W, the maximum time for writes to propagate.

Verification period. The verification period t_v is lower-bounded by the update period t_u and upper-bounded by t_{refresh}, the background refresh rate of writers t_{refresh} (i.e., the frequency at which writers query the server). In order to find the optimal t_v for a fixed number of partitions a, we need to consider all possible verification periods t_v where $t_u \leq t_v \leq t_{\mathsf{refresh}}$. The cost of a lookup ($\mathsf{work}_{\mathrm{L}}$) without the history forest ($\mathsf{agghistory} = \mathsf{false}$) is defined in terms of the number of hashes the client needs to compute in order to verify the update log ($\mathsf{work}_{\mathrm{UL}}$) and the base tree:

$$\mathsf{work}_{\mathrm{UL}}(t_v, a) := t_v/t_u \left(\max\left(1, \lceil \log_2(\mathsf{burst}_W/a)\rceil\right)\right)$$

$$\mathsf{work}_{\mathrm{L}}(t_v, a) := \mathsf{work}_{\mathrm{UL}} + \max\left(1, \lceil \log_2(N/a)\rceil\right)$$

The cost of a lookup when $\mathsf{agghistory} = \mathsf{true}$ adds a $\log_2(N)$ factor because the client looks up id in multiple base trees:

$$\mathsf{work}_{\mathrm{L},\mathsf{aghs}}(t_v, a) := \mathsf{work}_{\mathrm{UL}} + \log_2(N)\left(\max\left(1, \lceil \log_2(N/a)\rceil\right)\right)$$

The cost of monitoring without the history forest is the cost of doing lookups in the base tree for every verification period missed while the client was offline:

$$\mathsf{work}_{\mathrm{M}}(t_v, a) := \max\left(1, \lceil \log_2(N/a)\rceil\right) \cdot \frac{t_v}{t_{\mathsf{offline}}} \cdot \frac{N}{t_{\mathsf{offline}}}$$

When $\mathsf{agghistory} = \mathsf{true}$, the cost of monitoring is much lower because of the history forest:

$$\mathsf{work}_{\mathrm{M},\mathsf{aghs}}(t_v, a) := \max\left(1, \lceil \log_2(N/a)\rceil\right) \cdot \frac{N}{t_{\mathsf{offline}}}$$

Using the above equations, we can compute the optimal t_v^* for a fixed number of partitions a:

$$\mathsf{work}_{\mathrm{C}}(t_v, a) = \mathsf{work}_{\mathrm{L}}(t_v, a) + \mathsf{work}_{\mathrm{M}}(t_v, a)$$

$$\mathsf{work}_{\mathrm{C,aghs}}(t_v, a) = \mathsf{work}_{\mathrm{L,aghs}}(t_v, a) + \mathsf{work}_{\mathrm{M,aghs}}(t_v, a)$$

$$t_v^*(a) := \underset{t_v \in [t_u, t_{\mathsf{refresh}}]}{\arg\min} = \mathsf{work}_{\mathrm{C}}(t_v, a)$$

$$t_v^*(a, \mathsf{aghs}) := \underset{t_v \in [t_u, t_{\mathsf{refresh}}]}{\arg\min} = \mathsf{work}_{\mathrm{C,aghs}}(t_v, a)$$

Number of partitions. In order to set the number of partitions a, we need to consider all possible numbers of partitions a where $a \leq N$. The auditor work when $\mathsf{agghistory} = \mathsf{false}$ consists of verifying the append-only property of the update log and keeping track of the base tree root:

$$\mathsf{work}_{\mathrm{AoUL}}(a) := \max\left(1, \lceil \log_2(t_v^*(a)/t_u) \rceil\right)$$

$$\mathsf{work}_{\mathrm{A}}(a) := a \cdot (\mathsf{work}_{\mathrm{AoUL}}(a) + 1)$$

When $\mathsf{agghistory} = \mathsf{true}$, auditors must verify the append-only property of the update log and the history tree:

$$\mathsf{work}_{\mathrm{A,aghs}}(a) := a \cdot (\mathsf{work}_{\mathrm{AoUL}}(a) + \log_2(N))$$

Note that total auditor work monotonically increases with a. The optimal a^* is one that satisfies the auditor's CPU utilization constraint $\mathsf{cpu}_{\mathrm{A}}$ while minimizing client work:

$$a^* := \underset{a \in [1,N]}{\arg\min} \left(\mathsf{work}_{\mathrm{C}}(t_v^*(a))\right) \text{ s.t. } \mathsf{work}_{\mathrm{A}}(a) \leq \mathsf{cpu}_{\mathrm{A}}$$

$$a^*(\mathsf{aghs}) := \underset{a \in [1,N]}{\arg\min} \left(\mathsf{work}_{\mathrm{C,aghs}}(t_v^*(a, \mathsf{aghs}))\right) \text{ s.t. } \mathsf{work}_{\mathrm{A}}(a) \leq \mathsf{cpu}_{\mathrm{A}}$$

Using history forest. After computing the optimal number of partitions and verification period length for $\mathsf{agghistory} = \mathsf{true}$ and false, we can determine whether or not to use the history forest by comparing the client work of both options:

$$\mathsf{agghistory} := \mathsf{work}_{\mathrm{C,aghs}}(t_v^*(a^*, \mathsf{aghs})) \overset{?}{<} \mathsf{work}_{\mathrm{C}}(t_v^*(a^*))$$

Verifier. The planner looks at the leftover auditor CPU cycles after computing the above costs. If the remaining auditor cycles is enough to monitor on behalf of all writers, then the planner sets $\mathsf{verifier} = \mathsf{auditor}$ (otherwise $\mathsf{verifier} = \mathsf{client}$).

Option for detecting misbehavior early. LegoLog's planner can also help decide whether to detect misbehavior *before* other clients use incorrect values. This can be achieved by setting $t_v = t_{\mathsf{offline}}$ so that users never go offline longer than a verification period. The planner computes the optimal configuration for this setting. The developer can then compare this to detecting misbehavior reactively (instead of preventing it) and decide which approach to use for her application.

Server requirements. The server requirements returned by the planner are the number of hashes/second that the server needs to retrieve and transmit to the clients and auditors. This is computed by computing $\mathsf{work}_{\mathrm{C}}$ and $\mathsf{work}_{\mathrm{A}}$ for the optimal a^* and t_v^*, while accounting for burst_R.

Network bandwidth. Meeting network bandwidth requirements follows the same logic of meeting CPU requirements since network usage $\mathsf{bw}(\cdot)$ depends on the number of hash computations for lookup, monitoring, and auditing. If network bandwidth is the bottleneck, the planner optimizes over $\mathsf{bw}(\mathsf{work}_{\mathrm{C}})$ and $\mathsf{bw}(\mathsf{work}_{\mathrm{A}})$, and uses nw_{A} as the constraint.

3.5 Evaluation

In our evaluation, we aim to answer three questions:

1. How do parameter settings affect performance? (§3.5.2)
2. Does LegoLog perform comparably to existing special-purpose logs for different applications? (§3.5.3)
3. What workloads can LegoLog's API express, and how does it perform for these? (§3.5.4)

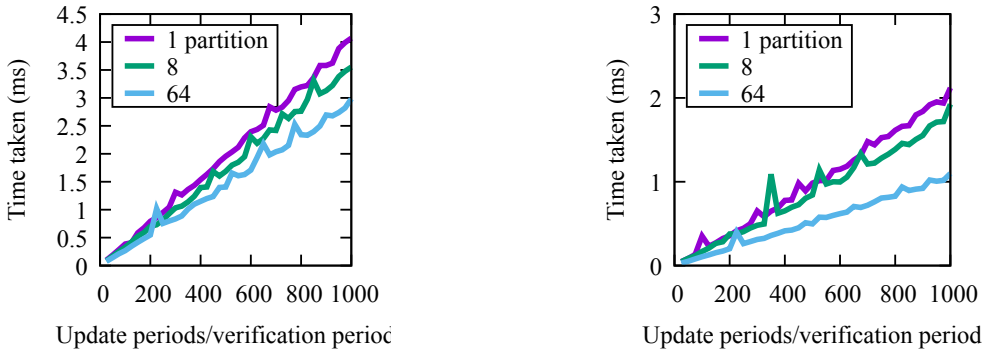
In answering these questions, we show that LegoLog's configurability does not come at a substantial loss in performance: LegoLog performs comparably to existing specialized transparency logs tailored to individual applications.

3.5.1 Implementation

Our LegoLog implementation is a fork of the existing Merkle^2 [143] implementation using Go and Python. We added $\sim$4000 LOC in order to implement additional core data structures, the LegoLog protocol, and the LegoLog planner. We use gRPC v1.49 for communication and SHAKE-128 as our hash function.

Memory optimization. To avoid copying the entire base tree every verification period, we use a combination of fat nodes and path copying from persistent data structures [88]. We maintain the invariant that child nodes are always at least as old as their parent nodes, which enables constant-time access to nodes at any verification period. We further optimize by only path copying between verification periods.

Experiment setup. We ran experiments on Google Cloud. For the server, we used an `n2-highmem-32` instance (32-core Intel Xeon CPU, 256GB RAM). For the client, we used an `n2-standard-8` instance (8-core Intel Xeon CPU, 32GB RAM). For the auditor, we used a lightweight `n2-standard-2` instance (2-core Intel Xeon CPU, 8GB RAM). For benchmarks, we read and write identifiers uniformly at random.



(a) Without history forests.

(b) With history forests.

Figure 3.7: Time taken for the server to generate a lookup proof for different settings of update periods in a single verification period (t_v/t_u) and number of partitions a.

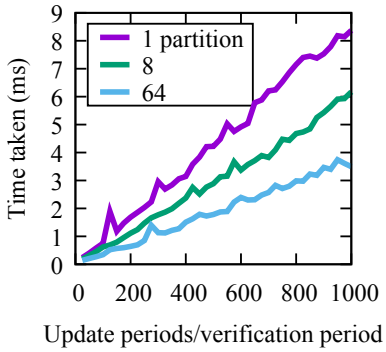
3.5.2 Microbenchmarks

We first investigate how different settings of internal system parameters impact the performance of LegoLog.

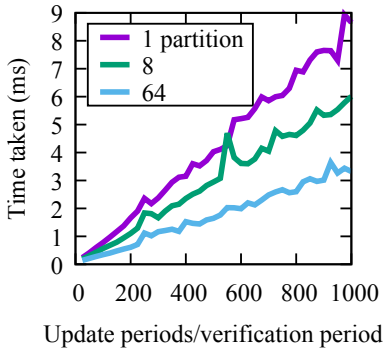
Read cost. We measure the client and server overheads of reading a value (Figures 3.7 and 3.8) for different numbers of partitions and different ratios of update periods to verification periods. We generate these plots for varying numbers of update periods in a single verification period (t_v/t_u) where each update period has 200 appends. The server is preloaded with 1M entries. Increasing the number of partitions decreases both client and server work because having more partitions reduces the amount of data in one partition and so the overhead of proving and verifying is smaller. Increasing the ratio of update periods to verification periods makes it possible for client writes to propagate without clients having to run monitoring as frequently. However, increasing this ratio also increases client and server lookup work. This increase is expected, as a lookup proof is required for each update period that has elapsed since the end of the last verification period.

Figures 3.9 and 3.10 measure the time taken by LegoLog configured to use history forests ($\mathsf{agghistory} = \mathsf{true}$) to generate and validate a lookup proof respectively, as the number of verification periods elapsed increases. The amount of work increases logarithmically with the number of verification periods elapsed because a lookup when $\mathsf{agghistory} = \mathsf{true}$ consists of a lookup for that value in each history tree's root and there are at most $\log(n)$ history trees in the history forest, where n is the number of nodes in the history forest (i.e., number of verification periods).

Auditor overhead. Figure 3.11 shows how the number of partitions affects auditing time. For each partition, the auditor must verify a chronological tree extension proof, and so the auditor work grows linearly with the number of partitions.



(a) Without history forests.



(b) With history forests.

Figure 3.8: Time taken for the client to validate a lookup proof for varying t_v/t_u and number of partitions a.

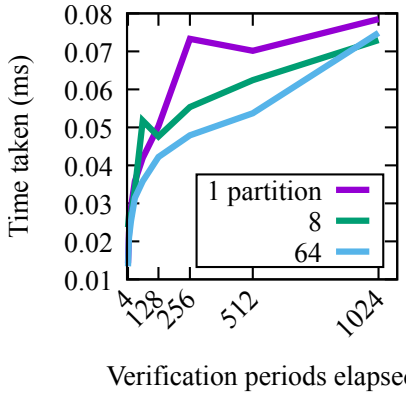


Figure 3.9: Lookup proof generation work over multiple verif. periods ($t_v = t_u$).

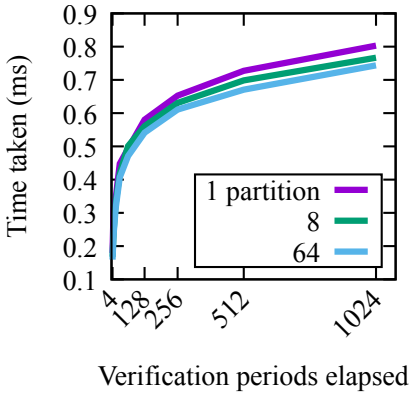


Figure 3.10: Lookup proof validation time over multiple verif. periods ($t_v = t_u$).

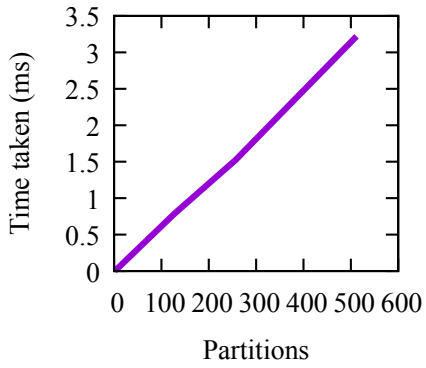


Figure 3.11: Median auditor work between update epochs as partitions a increase.

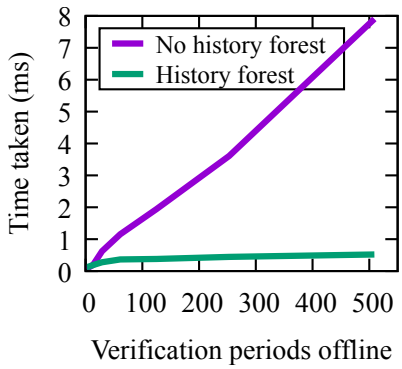


Figure 3.12: Client monitoring work as offline time increases.

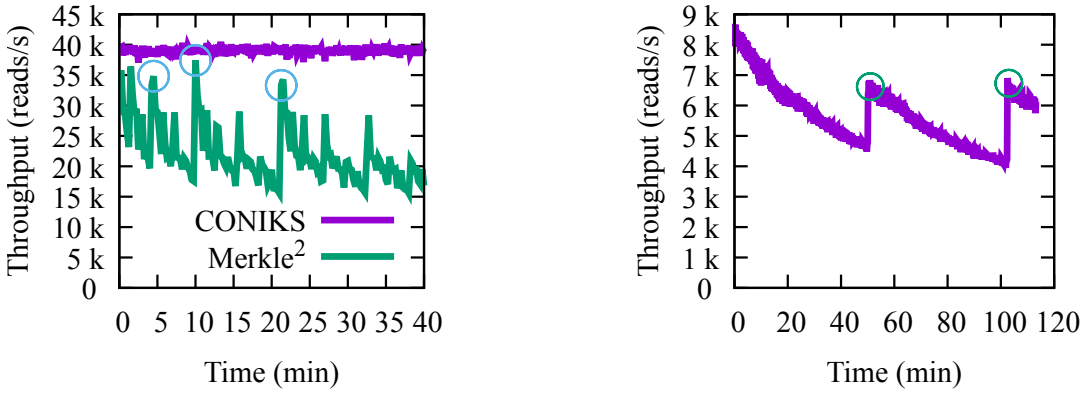
Specialized Log	N	t_W	burst$_W$	t_R	burst$_R$	t_{refresh}	t_{offline}	device$_R$	device$_W$	device$_A$
Merkle^2 [143]	1M	15s	75	30s	2.4k	30m	24h	`phone`	`phone`	`server`
CONIKS [208]	10M	60m	75	30s	24k	60m	60m	`phone`	`phone`	`server`
Binary transparency (Go)	30M	30s	50	1s	500	24h	24h	`laptop`	`laptop`	`laptop`

Table 3.2: Example developer parameter settings for specialized transparency logs for different applications.

Monitoring overhead. For client monitoring, we evaluate the case in which the monitor is offline for some number of verification periods (Figure 3.12). We measure the time taken to verify that a particular identifier-value pair exists in each verification period. Without the history forest, this means verifying the existence proof for each base verification tree. With the history forest, the monitor only has to verify existence for a logarithmic number of trees in the history forest. We perform 100 appends per verification period.

3.5.3 Existing transparency logs

In this section, we compare the *empirical performance* of LegoLog and three specialized transparency logs. To set the goal for the empirical performance evaluation, we remind the reader that LegoLog' contribution over existing transparency logs is that it is configurable as opposed to tailored for one specific setting, which is valuable for non-expert developers. As such, the purpose of the empirical evaluation is to show that the generality of LegoLog does not come at a substantial performance loss when compared to existing transparency logs, and not to beat the performance of other transparency logs (even though LegoLog is faster in a variety of settings). For each type of log, we report the throughput performance of the log server over multiple verification periods (t_v). We note that we report the results over singular runs, though we found the



(a) CONIKS and Merkle^2 workloads.

(b) Binary transparency workload.

Figure 3.13: LegoLog's throughput for three specialized logs.

throughput behavior of the LegoLog server to be consistent over multiple runs of each experiment.

Merkle^2 [143]. Key transparency (KT) allows clients to check that a server correctly maintains a mapping of usernames to public keys. We described how to set developer parameters based on the workload of a hypothetical KT system for Signal in §3.2.3. We first compare LegoLog's performance to Merkle^2 [143], which was designed for KT and runs on a log size of $N = 1$M. We scale the developer parameters (Table 3.2) back accordingly to match the log size of Merkle^2. For these parameters, LegoLog's planner outputs $t_u = 15$s, $t_v = 15$s, $a = 2010$, agghistory $=$ true, verifier $=$ client. We plot the average throughput over time in Figure 3.13a. The blue circles mark when the history forest merges into one tree and consequently the lookup proof size decreases, which increases throughput for that period. LegoLog achieves an average throughput of 21,688 reads/s, compared to Merkle^2's 28,910 reads/s (LookUpPKVerify, averaged over 5 min). Our observed throughput measurements fell between 16,075 reads/s (p5) and 31,718 reads/s (p95), and this variation comes from how many trees are in the history forest at a given point in time. LegoLog achieves comparable performance to Merkle^2 for the workload Merkle^2 is designed for. LegoLog's configurability comes with a small performance drop but offers a $\sim 5\times$ memory saving (25GB for Merkle^2 vs. 5GB for LegoLog).

CONIKS [208]. CONIKS runs on a log size of 10M, and we set the developer parameters (Table 3.2) to match their paper's evaluation [208]. We compare against the version of CONIKS enhanced with the persistent data structure implemented in [143]. For these parameters, the planner outputs $t_u = 60$m, $t_v = 60$m, $a = 78210$, agghistory $=$ false, verifier $=$ client. We plot the average throughput over time in Figure 3.13a. CONIKS has an average throughput of 35,141 reads/s (averaged over 5 min), LegoLog has an average throughput of 39,005 reads/s. Our observed throughput measurements fell between 37,778 reads/s (p5) and 40,014 reads/s (p95).

Application	Existing work				LegoLog		
	Work	Lookup	Audit	Monitor	Lookup	Audit	Monitor
Key transparency (KT) [208] KT for IOT [280] Certificate transparency (CT) [182] CT for IOT	[143]	$O(\log^2 N)$	$O(\log N)$	$O(\log^2 N)$	$O(\log^2 \frac{N}{a} + e \log \frac{U}{ea})$	$O(a(\log N + \log e + 1))$	$O(\log^2 \frac{N}{a})$
	[208]	$O(\log N)$	$O(1)$	$O(E \log N)$	$O(\log \frac{N}{a} + e \log \frac{U}{ea})$	$O(a(\log e + 1))$	$O(E \log \frac{N}{a})$
CT—deployed today Binary transparency [81, 141]	[120]	$O(N)$	$O(\log N)$	$O(N)$	$O(\frac{N}{a})$	$O(a \log N)$	$O(\frac{N}{a})$
Enhanced CT	[268]	$O(\log N)$	$O(N)$ or $O(\log N)$	$O(1)$ or $O(N)$	$O(\log \frac{N}{a} + e \log \frac{U}{ea})$	$O(a(\log e + 1) + N)$	$O(1)$
IOT authentication	[7]	$O(\log N)$	$O(N)$	$O(1)$	$O(\log \frac{N}{a} + e \log \frac{U}{ea})$	$O(a(\log e + 1) + N)$	$O(1)$
Encrypted backups	[74]	$O(\log N)$	$O(\log N)$	$O(1)$	$O(\log \frac{N}{a} + e \log \frac{U}{ea})$	$O(a(\log e + 1) + N)$	$O(1)$

Table 3.3: Asymptotics of lookup, auditing, and monitoring work for transparency logs derived by LegoLog for applications, compared to existing work. N is the number of entries in the log, U is the number of updates per verification period, a is the number of partitions, E is the number of verification periods since the value was last monitored, and e is the number of update periods in a verification period.

Binary Transparency. Go's checksum database enables users to authenticate their downloaded Go modules by maintaining the modules' checksums on a transparency log [141]. We set the developer parameters (Table 3.2, row 2) based on Go's deployed checksum database, which currently has 17M entries [113]. Every day ∼16k entries are added [114], averaging to 6 writes every 30s. A log size of 30M can support 5 years worth of module checksums. Go has 2.7M users [331], and a conservative estimate of 10 module downloads/user/day yields an average of 313 reads/s (we can treat this as the baseline throughput that we wish LegoLog to achieve). The devices used by the reader, writer, and auditor are set to be laptops. LegoLog's planner outputs $t_u = 30$s, $t_v = 55.5$m, $a = 7410$, agghistory = false, verifier = client. We plot the server's request throughput over time in Figure 3.13b and show that LegoLog can achieve on average 5,725 reads/s. Our observed throughput measurements fell between 4,259 reads/s (p5) and 7,733 reads/s (p95). The fluctuation in throughput comes from the update periods that have elapsed within a verification period. When a verification period is over, the throughput suddenly increases because half of the update log has been cleared and merged into the verified base tree (circled in blue in Figure 3.13b).

3.5.4 API flexibility

We now evaluate *functionality* by showing that LegoLog can generate the functionality of 6 transparency logs via its API, and we include a comparison of asymptotic complexity. Table 3.3 shows how the LegoLog API can express a variety of applications. For each application, we show asymptotic costs based on the input developer parameters. Our goal is to show that when tailored to various applications, LegoLog overheads are comparable to existing work (i.e., configurability does not come at a significant performance cost).

In key transparency and certificate transparency (Row 1, Table 3.3), setting t_W can affect whether agghistory is turned on or off, and so can produce transparency logs with different asymptotic behavior. When agghistory is turned on, a lookup consists of looking up a value

in the base tree of each history tree's root ($O(\log^2 \frac{N}{a})$). When agghistory is turned off, a lookup just consists of looking up a value in the base tree of the last verification period ($O(\log \frac{N}{a})$). In either case, the client also needs to look up the proofs of inclusion or exclusion in the update trees ($O(e \log \frac{U}{ea})$). When agghistory is turned on, the auditor must verify that the history forest of base trees in each partition is append-only ($O(\log N)$). In either case, the auditor needs to check that the update log in each partition is append-only ($O(\log e + 1)$). When agghistory is turned on, monitoring consists of verifying an inclusion proof from a path of base trees in the history forest ($O(\log^2 \frac{N}{a})$). When agghistory is turned off, monitoring consists of verifying an inclusion proof from *each* verification period since the client last monitored the value ($O(E \log \frac{N}{a})$).

Certificate transparency [182], as deployed today, and binary transparency [28] (Row 2, Table 3.3) use chronological trees, which LegoLog can output by setting $t_v = \infty$ (i.e., the chronological udpdate tree never gets rolled up into a base tree). Looking up and monitoring a value consists of searching through all of the values in a shard ($O(\frac{N}{a})$). Auditing consists of an extension proof in each shard ($O(a \log U) \subseteq O(a \log N)$).

In the case of enhanced CT (Row 3, Table 3.3) and IoT authentication (Row 4, Table 3.3), LegoLog can produce comparable transparency logs when a powerful auditor device is specified as input (so that verifier is set to auditor) and when $t_v = t_u$. A lookup for a value consists of an inclusion proof in a partition ($O(\log \frac{N}{a})$) and inclusion/exclusion proofs in the partition's update trees ($O(e \log \frac{U}{ea})$). Auditing consists of validating that the update log in each partition is append-only ($O(a(\log e + 1)$) and that update log entries have been correctly rolled into the next base tree ($O(N)$). The client only needs to keep track of the digest when monitoring ($O(1)$). In encrypted backups [74] (Row 5, Table 3.3), the threat model differs slightly from LegoLog's because clients trust a subset of the hardware security modules, enabling SafetyPin to achieve asymptotically better auditing costs.

3.6 Discussion

Scalability considerations. LegoLog's scalability is primarily determined by server and auditor resources. Our planner automatically calculates and outputs the minimum server requirements (CPU, bandwidth, and storage) needed for a specific deployment's performance goals. The system scales horizontally by increasing the number of partitions (a), which divides the log into independent shards. This partitioning allows lookups and verification operations to be parallelized across shards, significantly improving throughput for large deployments. As shown in our evaluation, increasing the number of partitions reduces both client and server work per operation, enabling LegoLog to scale to logs with millions of entries (§3.5.3). The trade-off is increased auditor work, which grows linearly with the number of partitions, and so this parameter should be optimized based on the specific workload constraints and available resources.

3.7 Related work

Transparency logs. A rich line of work has examined improving transparency logs. CONIKS was the first academic work to propose a transparency log for public keys [208] and backs Apple's deployment of iMessage key transparency [11]. It relies on the owner to monitor their key every epoch, which can be impractical for short epochs. LegoLog enables trading off less frequent monitoring for more expensive lookups. ECT [268] and WAVE [7] maintain both a chronological tree and a prefix tree, but require an auditor to check the correspondence between the two, which is expensive. Google's Trillian [120, 121] is mainly used for certificate transparency, and it provides a verifiable log abstraction.

Merkle^2 maintains a prefix tree inside each node of a chronological tree [143], enabling efficient monitoring proofs. Merkle^2 clients that go offline for a while can discover that a server misbehaved but cannot prevent this misbehavior. LegoLog enables the client to prevent misbehavior by trading off performance for longer verification periods.

OPTIKS [190], Parakeet [202], and SEEMless [53] build off of CONIKS [208] and use Merkle Patricia Tries (MPTs) as their core data structure. Ghosh and Chase describe an auditor-free version of OPTIKS [111]. Parakeet reduces storage space in comparison to SEEMless by storing the epoch in the leaf node, allowing for any epoch's MPT to be reconstructed. Similarly, LegoLog also allows for any verification period's tree to be efficiently reconstructed (§3.5.1).

Keybase was the first deployment of a publicly auditable public key directory [166]. WhatsApp recently deployed a key transparency system [183] based on SEEMless [53] and Parakeet [202]. Cloudflare's validation service audits the WhatsApp deployment [215], and an IETF working group is examining how to make transparency systems simple to run across organizations [147]. Elektra [189] builds off of SEEMless and targets the use case where users have multiple devices, using a form of signature chains similar to how users in LegoLog and Merkle^2 [143] sign updates (§3.3.4).

Binary transparency [4, 142, 230] builds off of certificate transparency [182] and is used for validating software updates. LegoLog can derive transparency logs for such workloads. Recent work also showed how to build a verifiable registry for voter registration that is backwards-compatible with existing systems [95].

Byzantine fault tolerance. In the decentralized system model, transparency logs can be implemented using consensus protocols [8, 21, 46, 90, 127, 128, 172, 181, 201, 228, 327]. LegoLog focuses on the single-server model and does not require heavyweight consensus protocols. Prior work also uses a public blockchain to publish an auditable directory, an approach constrained by the performance of the underlying blockchain. EthIKS [33] and Catena [300] show how to use the public blockchain as a public ledger to commit to a CONIKS directory and minimize client bandwidth for auditing.

Heavyweight cryptographic primitives. Another approach to transparency logs is to use powerful, but more expensive, cryptographic primitives such as succinct non-interactive arguments of knowledge (SNARKs) [58, 305, 306], accumulators [299] or vector commitments [191]

that are more expensive to compute, but have better asymptotic performance. LegoLog does not capture transparency logs based on these cryptographic primitives. While these logs improve asymptotics, they tend to have high constants and so high overheads in practice.

3.8 LegoLog protocol specification

We now fully specify the implementation of the LegoLog protocol, implementing the API specified in §3.3.6.

3.8.1 Building blocks

We first define the syntax for existing building blocks.

Chronological Merkle Tree

ChronTree.Append(val): Appends val to the tree.

ChronTree.ProveExtension($\mathsf{root}_{\mathsf{old}}$, root) $\to \pi$: Generate proof that the tree rooted at root is an extension of the tree rooted at $\mathsf{root}_{\mathsf{old}}$. A detailed algorithm for generating the extension proof is found in §2.1.4.1. of [182].

ChronTree.VerifyExtension(π, $\mathsf{root}_{\mathsf{old}}$, root)) $\to \{0,1\}$: Outputs 1 if the extension proof π is valid and 0 otherwise.

ChronTree.GetPath(i) $\to$ (val, Path): Outputs the value and Merkle path associated with i-th inserted value.

ChronTree.CheckPath(root, i, val, Path) $\to \{0,1\}$: Outputs 1 if the result of hashing Path, i, and val matches root and 0 otherwise.

ChronTree$[i]^* \to$ (val, $\{0,1\}$): Outputs the i-th inserted val and whether it has a valid proof by calling
ChronTree.CheckPath(root, i, ChronTree.GetPath(i)).

ChronTree$[i] \to$ val: Outputs the i-th inserted val.

Chronological Merkle Forest

ChronForest.Append(val): Appends val to the forest.

ChronForest.ProveExtension($\mathsf{roots}_{\mathsf{old}}$, roots) $\to \pi$: Generate proof that the forest with roots roots is an extension of the forest with roots $\mathsf{roots}_{\mathsf{old}}$. A detailed algorithm for generating the extension proof for chronological trees is found in §2.1.4.1. of [182] and a detailed algorithm for generating the extension proof for chronological forests is found in §V.A of [143].

ChronForest.VerifyExtension(π, $\mathsf{roots}_{\mathsf{old}}$, roots) $\to \{0,1\}$: Outputs 1 if the extension proof π is valid and 0 otherwise.

ChronForest.GetPath(i) $\to$ (val, Path): Outputs the value and Merkle path associated with i-th inserted value.

ChronForest.CheckPath(root, i, val, Path) $\rightarrow \{0,1\}$: Outputs 1 if the result of hashing Path, i, and val matches root and 0 otherwise.

ChronForest$[i]^* \rightarrow$ (val, $\{0,1\}$): Outputs the i-th inserted val and whether it has a valid proof by calling
ChronForest.CheckPath(root, i, ChronForest.GetPath(i)).

ChronForest$[i] \rightarrow$ val: Outputs the i-th inserted val.

Prefix Merkle Tree

PrefixTree.BuildPrefixTree(idvals) $\rightarrow$ PrefixTree: Constructs a Merkle prefix tree given id-value pairs idvals.

PrefixTree.GetPath(id) $\rightarrow$ (val, Path): Outputs the value and Merkle path associated with id in a Merkle prefix tree. If id is in the tree, Path consists of the siblings of the nodes in the path from id to the tree's root. If id is not in the tree, val $\leftarrow \perp$ and Path is a proof of exclusion which is the longest-prefix match of id currently in the tree (either a leaf node id$' \neq$ id that matches the first few bits of id or an empty node that is a prefix of id).

PrefixTree.CheckPath(root, id, val, Path) $\rightarrow \{0,1\}$: Outputs 1 if the result of hashing Path, id, and val matches root and 0 otherwise.

PrefixTree.Insert(id, val): Inserts id, val into the tree.

Signatures

Sign(sk, val) $\rightarrow \sigma$: Signs val with signing key sk.

VerifySig(vk, (val, σ)) $\rightarrow \{0,1\}$: Verifies signature σ.

3.8.2 Server

config := (t_u, t_v, a, agghistory, verifier), as described in Table 3.1.

Server$[\alpha]$:= ($QueryBT$, $VerifyBT$, $UpdateLog$,
$HistoryForest$, $CurrUpdates$, t,
$HashChain$)

- α is the partition index.
- $QueryBT$ is a PrefixTree.
- $VerifyBT$ is a PrefixTree.
- $UpdateLog$ is a ChronTree of PrefixTrees from each update period.
- $HistoryForest$ is a ChronForest consisting of base trees from each verification period.
- $CurrUpdates$ is a list of updates in the current update period.

- t is the current epoch.
- $HashChain$ is a hash chain of the server's state.

Notation: Server$[\alpha]$[id] denotes the value of id in partition α.

Server$[\alpha]$.Write(id, (val, σ))

Server$[\alpha].CurrUpdates \leftarrow$ ((val, σ), Server$[\alpha].CurrUpdates$)

Server.NextEpoch(config)

for $\alpha \leftarrow 1, \ldots,$ config.a **do**
    Server$[\alpha]$.NextEpoch(config)
end for

Server$[\alpha]$.NextEpoch(config)

updateTree $\leftarrow$ BuildPrefixTree(Server$[\alpha].CurrUpdates$)
Server$[\alpha].UpdateLog$.Append(updateTree)
if Server$[\alpha].t + 1 \mod$ config.$t_v = 0$ **then**
    if config.agghistory **then**
        Server$[\alpha].HistoryForest$.Append(
          Server$[\alpha].QueryBT$)
    else
        Server$[\alpha].HashChain \leftarrow H($
          Server$[\alpha].QueryBT$.root,
          Server$[\alpha].VerifyBT$.root,
          Server$[\alpha].t$, Server$[\alpha].HashChain$)
    end if
    Server$[\alpha].QueryBT \leftarrow$ Server$[\alpha].VerifyBT$
    for ut $\in$ Server$[\alpha].UpdateLog$ **do**
        for (id, val) $\in$ ut **do**
            $v \leftarrow$ Server$[\alpha]$[id]
            if $v = \perp$ **then**
                v.vk $\leftarrow$ val
            end if
            v.vals $\leftarrow$ (val, v.vals)
            v.last $\leftarrow$ val
            Server$[\alpha].VerifyBT$.Insert(id, v)
        end for
    end for
    Server$[\alpha].UpdateLog \leftarrow ()$
end if
Server$[\alpha].CurrUpdates \leftarrow ()$
Server$[\alpha].t \leftarrow$ Server$[\alpha].t + 1$

Server.GetDigest() $\rightarrow$ dgst

1: dgst $\leftarrow$ ()
2: **for** $\alpha \leftarrow 1, \ldots,$ config.a **do**
3: dgst$[\alpha] \leftarrow$ Server$[\alpha]$.GetDigest()
4: **end for**
5: Output dgst

Server$[\alpha]$.GetDigest() $\rightarrow$ dgst$[\alpha]$

1: Output $(t,$ Server$[\alpha].QueryBT$.root,
Server$[\alpha].VerifyBT$.root,
Server$[\alpha].HistoryForest$.roots,
Server$[\alpha].UpdateLog$.root, Server$[\alpha].HashChain)$

Server.Prove(config, t_{old}, t_{new}) $\rightarrow \pi$

1: $\pi \leftarrow$ ()
2: **for** $\alpha \leftarrow 1, \ldots,$ config.a **do**
3: $\pi[\alpha] \leftarrow$ Server$[\alpha]$.Prove(config, $t_{\text{old}}, t_{\text{new}}$)
4: **end for**
5: Output π

Server$[\alpha]$.Prove(config, t_{old}, t_{new}) $\rightarrow \pi[\alpha]$

1: Server$[\alpha]_t$:= the state of the server's partition α at epoch t.
2: **if** config.agghistory **then**
3: $\pi[\alpha].hf \leftarrow$ ChronForest.ProveExtension(
Server$[\alpha]_{t_{\text{old}}}.HistoryForest$.roots,
Server$[\alpha]_{t_{\text{new}}}.HistoryForest$.roots)
4: **else**
5: $\pi[\alpha].hc \leftarrow$ ((Server$[\alpha]_{t_{\text{old}}+1}.QueryBT$.root,
Server$[\alpha]_{t_{\text{old}}+1}.VerifyBT$.root, $t_{\text{old}}+1$),
$\ldots$,
(Server$[\alpha]_{t_{\text{new}}-1}.QueryBT$.root),
Server$[\alpha]_{t_{\text{new}}-1}.VerifyBT$.root, $t_{\text{new}}-1$))
6: **end if**
7: **if** $\lfloor t_{\text{old}}/\text{config}.t_v \rfloor = \lfloor t_{\text{new}}/\text{config}.t_v \rfloor$ **then**
8: $\pi[\alpha].ul \leftarrow$ ChronTree.ProveExtension(
Server$[\alpha]_{t_{\text{old}}}.UpdateLog$.root,
Server$[\alpha]_{t_{\text{new}}}.UpdateLog$.root)
9: **end if**
10: Output π

3.8.3 Auditor

dgst$[\alpha] := (t,\ qroot,\ vroot,\ hfroots,\ ulroot,\ HashChain)$

- t is the current epoch.
- $qroot$ is the root of $QueryBT$.
- $vroot$ is the root of $VerifyBT$.
- $hfroots$ are the roots of the $HistoryForest$.
- $ulroot$ is the root of $UpdateLog$.
- $HashChain$ is the hash chain of the server's state.

Auditor.GetDigest$(t) \rightarrow$ dgst:
Auditor.Audit(config, dgst_{old}, dgst_{new}, $\pi) \rightarrow \{0,1\}$:

1: ok $\leftarrow 1$
2: **for** $\alpha \leftarrow 1, \ldots,$ config.a **do**
3: **if** config.agghistory **then**
4: ok $\leftarrow$ ok $\bigwedge$ VerifyExtension$(\pi[\alpha].hf,$ $\text{dgst}_{\text{old}}[\alpha].hfroots,$ $\text{dgst}_{\text{new}}[\alpha].hfroots)$
5: **else**
6: $h' \leftarrow \text{dgst}_{\text{old}}[\alpha].HashChain$
7: **for** $n \in \pi[\alpha].hc$ **do**
8: $h' \leftarrow H(n,\ h')$
9: **end for**
10: ok $\leftarrow$ ok $\bigwedge\ h' \stackrel{?}{=} \text{dgst}_{\text{new}}[\alpha].HashChain$
11: **end if**
12: **if** $\text{dgst}_{\text{old}}[\alpha].t \mod$ config.$t_v =$ $\text{dgst}_{\text{new}}[\alpha].t \mod$ config.t_v **then**
13: ok $\leftarrow$ ok $\bigwedge$ VerifyExtension$(\pi[\alpha].ul,$ $\text{dgst}_{\text{old}}[\alpha].ulroot,$ $\text{dgst}_{\text{new}}[\alpha].ulroot)$
14: **end if**
15: **end for**
16: Output ok

3.8.4 Client

config $:=$ $(t_u,\ t_v,\ a,$ agghistory, verifier$)$, as described in Table 3.1.
Client $:=$ $(LastMonitored,\ VBTRoots)$

- $LastMonitored$ is a map of identifiers id to the latest verification period the client has monitored for that value.
- $VBTRoots$ is a map of identifiers id to a set of base tree roots that have already been verified by the client (for that id).

Client.Read(config, dgst, id, $t) \rightarrow$ val

1: ok $\leftarrow 1$
2: $\alpha \leftarrow H($id$) \mod$ config.a

```
3: root ← dgst[α].qroot
4: vals, Path_id ← Server[α].QueryBT.GetPath(id)
5: ok ← ok ⋀ CheckPath(root, id, vals, Path_id)
6: if config.agghistory then
7:     for root_h ∈ Server[α].HistoryForest.roots do
8:         t_h ← verification period root_h corresponds to
9:         val_h, Path_{id,h} ←
             Server[α].HistoryForest[t_h].GetPath(id)
10:        ok ← ok ⋀ CheckPath(root_h, id, val_h, Path_{id,h})
11:        ok ← ok ⋀ val_h.vals ⊆? val.vals
12:    end for
13:    h ← 0
14:    r ← ⌊t/t_v⌋
15: end if
16: if VerifySig(vals.vk, vals.last) then
17:    val_latest ← vals.last
18: end if
19: for u = 0, ..., ⌊(t − t_v · ⌊t/t_v⌋)/t_u⌋ do
20:    (tree_u, ok_u) ← dgst[α].UpdateLog[u]*
21:    ok ← ok ⋀ ok_u
22:    root_u ← tree_u.root
23:    val_u, Path_{id,u}, ← tree_u.GetPath(id)
24:    ok ← ok ⋀ CheckPath(root_u, id, val_u, Path_{id,u})
25:    if val_u ≠⊥ ⋀ VerifySig(vk, val_u) = 1 then
26:        val_latest ← val_u
27:    end if
28: end for
29: if ok then
30:    Output val_latest
31: else
32:    Output ⊥
33: end if
```

Client.Write(config, sk, id, val, t) → $\{0,1\}$

```
1: σ ← Sign(sk, (id, val))
2: α ← H(id) mod config.a
3: Server[α].Write(id, (val, σ))
4: Wait until epoch t + 1
5: dgst ← Auditor.GetDigest(t + 1)
6: val_latest ← Client.Read(config, dgst, id, t + 1)
7: Output val_latest ≟? val
```

Client.Monitor(config, dgst, id, val, t) $\rightarrow \{0, 1\}$

ok $\leftarrow 1$
$t_m \leftarrow \lfloor t/\text{config}.t_v \rfloor$
$\alpha \leftarrow H(\text{id}) \mod \text{config}.a$
val′, Path $\leftarrow$ Server$[\alpha].VerifyBT$.GetPath(id)
ok $\leftarrow$ ok $\bigwedge$ CheckPath(dgst$[\alpha].vroot$, id, val′, Path)
ok $\leftarrow$ ok $\bigwedge$ val′ $\stackrel{?}{=}$ val
if config.agghistory **then**
    $t' \leftarrow$ verification period id was updated to val
    for $t_l \in$ path from t' to root in Server$[\alpha].HistoryForest$ **do**
        dgst$_l \leftarrow$ Auditor.GetDigest(t_l)
        val′, Path $\leftarrow$ Server$[\alpha].HistoryForest[t_l]$.GetPath(id)
        ok $\leftarrow$ ok $\bigwedge$ CheckPath(dgst$_l[\alpha].vroot$, id, val′, Path)
        ok $\leftarrow$ ok $\bigwedge$ val′ $\stackrel{?}{=}$ val
    end for
else
    for $t_l \leftarrow$ Client.$LastMonitored$[id], $\ldots, t_m$ **do**
        dgst$_l \leftarrow$ Auditor.GetDigest(t_l)
        val′, Path $\leftarrow$ Server$[\alpha].HistoryForest[t_l]$.GetPath(id)
        ok $\leftarrow$ ok $\bigwedge$ CheckPath(dgst$_l[\alpha].vroot$, id, val′, Path)
        ok $\leftarrow$ ok $\bigwedge$ val′ $\stackrel{?}{=}$ val
    end for
end if
Client.$LastMonitored$[id] $\leftarrow t_m$
Output ok

3.9 Security analysis

3.9.1 Building blocks

Before we prove the security of LegoLog, we first introduce security definitions for Merkle trees that are used as building blocks in LegoLog's core data structure. These definitions and security proofs for Merkle trees have already been established [143, 182, 210], but we review them here before moving on to prove the security of LegoLog itself.

Security of prefix Merkle trees

Prefix Merkle trees, as introduced in §3.3.1, have their leaves organized in lexicographic order, and internal nodes store the range of the child nodes.

Lemma 3.2. Let T be a prefix Merkle tree containing a leaf node L with identifier id and value val. Let h be the hash function used to construct T. If an adversary is able to generate a proof π for the leaf $(\mathsf{id}, \mathsf{val}')$ where $\mathsf{val}' \neq \mathsf{val}$ that verifies against the root hash R of T, then the adversary has found a collision in the hash function h.

Proof. Since T contains the leaf $(\mathsf{id}, \mathsf{val})$, there must exist a path P from L to R consisting of nodes $n_1, n_2, \ldots, n_k$ such that:

$$\begin{aligned} n_1 &= h(L) \\ n_2 &= h(n_1, n_1') \text{ for some } n_1' \\ &\vdots \\ n_k &= R \end{aligned}$$

Similarly, π defines a path P' consisting of nodes $n_1', n_2', \ldots, n_k'$ such that:

$$\begin{aligned} n_1' &= h(L') \text{ where } L' = (\mathsf{id}, \mathsf{val}') \\ n_2' &= h(n_1', n_1'') \text{ for some } n_1'' \\ &\vdots \\ n_k' &= R \end{aligned}$$

Since $\mathsf{val} \neq \mathsf{val}'$, we have $L \neq L'$. Therefore, $n_1 = h(L) \neq h(L') = n_1'$. However, both paths P and P' lead to the same root hash R. Therefore, there exist distinct values n_1 and n_1' that hash to the same value $n_2 = n_2'$. This means h has a collision. □

Security of chronological Merkle trees

Chronological Merkle trees, as introduced in §3.3.1, have leaves organized by time the leaf was appended to the tree.

Lemma 3.3. Let T be a chronological Merkle tree. Let h be the hash function used to construct T. Suppose T contains leaves $L_1, L_2, \ldots, L_n$ that were appended in that order. Let $L_n = (\mathsf{val})$. Let R be the root hash of T. If an adversary is able to generate a proof π that a leaf $L_n' = (\mathsf{val}')$ was appended to T after L_{n-1}, where $\mathsf{val}' \neq \mathsf{val}$, then the adversary has found a collision in the hash function h.

Proof. The adversary generates a proof π that T contains L_n' under the same root R. As proved in Lemma 3.2, this is possible only if the adversary has found a collision in the hash function h. □

Lemma 3.4. Let T_n be a chronological Merkle tree containing leaves $L_1, \ldots, L_n$ appended in order. Let $\pi_{j \to n}$ be an extension proof showing T_n extends T_j, for $j < n$. If an adversary generates an extension proof $\pi'_{j \to n}$ showing tree T'_n extends T_j, where T'_n contains a leaf $L'_i = (\mathsf{val}'_i)$ such that $i < j$ and $\mathsf{val}'_i \neq \mathsf{val}_i$, then the adversary has found a collision in h.

Proof. Let R_j be the root of T_j and R'_n be the root of T'_n. Since π' shows T'_n extends T_j, $\pi'_{j \to n}$ must contain the hashes needed to compute R_j, and the hashes needed to compute R'_n. Since T_j contains the leaf $L_i = (\mathsf{val}_i)$, there must exist a path P from L to R_j consisting of nodes $n_1, n_2, \ldots, n_k$ such that:

$$\begin{aligned} n_1 &= h(L_i) \\ n_2 &= h(n_1, n'_1) \text{ for some } n'_1 \\ &\vdots \\ n_k &= R_j \end{aligned}$$

Similarly, if T'_n contains a leaf $L'_i = (\mathsf{val}'_i)$ such that $i < j$ and $\mathsf{val}'_i \neq \mathsf{val}_i$, then and $\pi'_{j \to n}$ must define a path P' from L'_i to R_j consisting of nodes $n'_1, n'_2, \ldots, n'_k$ such that:

$$\begin{aligned} n'_1 &= h(L'_i) \\ n'_2 &= h(n'_1, n''_1) \text{ for some } n''_1 \\ &\vdots \\ n'_k &= R_j \end{aligned}$$

Since $\mathsf{val}_i \neq \mathsf{val}'_i$, $n_1 = h(L) \neq h(L') = n'_1$. However, both paths P and P' lead to the same root hash R_j. Therefore, there exist distinct values n_1 and n'_1 that hash to the same value $n_2 = n'_2$. Thus, the adversary has found a collision in h. □

3.9.2 Security game

We define security for LegoLog via a security game (Figure 3.14, Experiment 3.5). In our security game, we model the case where honest clients interact with a server controlled by the adversary. In our model, all honest clients also interact with at least one honest auditor and all have the same view of the server state. To capture this in the security game, the challenger plays the role of the honest clients and an auditor. The adversary sends one digest per round to the challenger, capturing the fact that all honest clients have the same view. The adversary can choose the identifier-value pairs that the honest clients write and, for each round, choose which identifier-value pairs are monitored and which are not. Before the game ends, the challenger audits and reads all identifier-value pairs. The adversary's goal is to pass all auditing and monitoring checks while ensuring that a read outputs a value that does not match the latest write from an honest client.

This security game captures both detection and prevention of log tampering. For detection, the adversary can choose for each round, which identifier-value pairs are monitored and

which are not, and the challenger must detect any tampering in the last monitoring check on all identifier-value pairs inserted by the challenger. For prevention, in the last round, the challenger reads without doing any monitoring, and so any misbehavior must be caught by the auditing proof or the monitoring proof in the previous round(s).

3.9.3 Security proof

We now prove Theorem 3.1. At a high level, we must reduce the advantage of the adversary in Experiment 3.5 to the advantage of an adversary against the signature scheme or the collision-resistant hash function. We can prove this via a hybrid argument.

Hybrid game. We construct a hybrid security game (Figure 3.15, Experiment 3.6) where we modify the win conditions of the original security game so that the adversary only wins if the challenger outputs an old (overwritten) value for a given identifier as a result of a read. The difference between the original game and this hybrid game is precisely the probability that the adversary can convince the challenger to accept a new value that was never written with the original secret key associated with that identifier.

In Figure 3.16, Experiment 3.7, we define $\mathcal{A}_\triangle$ to be the adversary that can win the difference in the games of Experiment 3.5 and Experiment 3.6. That is, $\mathcal{A}_\triangle$'s goal is to pass all auditing and monitoring checks while ensuring that read outputs a value that does not match *any* previous write from an honest client.

In Figure 3.17, we use the adversary $\mathcal{A}_\triangle$ to construct an adversary $\mathcal{B}$ in the EUF-CMA security game, where $\mathcal{B}$ plays the challenger in our security game and also interfaces between the challenger in the EUF-CMA game and the adversary $\mathcal{A}_\triangle$, sending identifier-value pairs from the adversary $\mathcal{A}_\triangle$ to the EUF-CMA challenger as messages and using signature responses from the EUF-CMA challenger to construct messages to send to the adversary $\mathcal{A}$ as a part of write requests. $\mathcal{B}$ fixes the signature scheme for one of the identifiers $\mathsf{id}_{\mathsf{pk}}$ that $\mathcal{A}_\triangle$ sends to be interacting with the challenger in the EUF-CMA game. For other identifiers, $\mathcal{B}$ samples a new signing key (if it does not yet exist) and uses that when writing values for that identifier to the server. If $\mathcal{A}_\triangle$ wins, then it has produced a new value that was never written (with a valid signature) with the original key associated with that identifier. If this identifier is $\mathsf{id}_{\mathsf{pk}}$, then $\mathcal{B}$ can send this back to the EUF-CMA challenger to win the game. Since $\mathcal{A}_\triangle$ cannot distinguish between which identifier is using the EUF-CMA challenger and is using $\mathcal{B}$'s sampled signing keys, $\mathcal{B}$ will succeed with probability $1/\mathsf{poly}(\cdot)$. Therefore, the difference between the adversary's advantage in the original security game and in the first hybrid game is simply the advantage an EUF-CMA adversary (with some polynomial factors) which, given a secure signature scheme, is negligible.

We now prove that if the adversary wins in this modified game (Experiment 3.6), then we can construct an adversary $\mathcal{C}$ that can find a collision in the hash function. We prove this for the case when $\mathsf{agghistory} = \mathsf{false}$ and $\mathsf{agghistory} = \mathsf{true}$ separately.

Case 1. We first argue security when $\mathsf{agghistory} = \mathsf{false}$. Assume that the challenger reads an old value val' for id when the last write to id was val ($\mathsf{agghistory} = \mathsf{false}$), and that all monitoring

Experiment 3.5. The below security game for a configurable transparency log against an adversary $\mathcal{A}$ is parameterized by a configuration config, and the security parameter λ.
The game proceeds as follows:

- The challenger initializes dictionary dict and round number $i = 0$.
- Each round i proceeds as follows:
  - The challenger receives $(\mathsf{id}_1, \mathsf{val}_1, \ldots, \mathsf{id}_n, \mathsf{val}_n)$ from $\mathcal{A}$.
  - The challenger sets $w_i = 1$. w_i represents whether all of the writes in this round were valid.
  - For $j \in [n]$, the challenger:
    - Checks if there exists $\mathsf{sk}_{\mathsf{id}_j}$. If not, the challenger samples a new signing key $\mathsf{sk}_{\mathsf{id}_j}$.
    - The adversary plays the role of the server to run:

$$w_i \leftarrow w_i \wedge \mathsf{Client.Write}(\mathsf{config}, \mathsf{sk}_{\mathsf{id}_j}, \mathsf{id}_j, \mathsf{val}_j, i)$$

    - Sets $\mathsf{dict}[\mathsf{id}_j] \leftarrow \mathsf{val}_j$.
  - The challenger gets $(\mathsf{dgst}_i, \pi_i, M)$ from $\mathcal{A}$.
  - The challenger runs

$$b_{i,\mathsf{id}} \leftarrow \mathsf{Client.Monitor}(\mathsf{config}, \mathsf{dgst}_i, \mathsf{id}, \mathsf{dict}[\mathsf{id}], i)$$

    for each $\mathsf{id} \in M$ with $\mathcal{A}$ playing the role of the server. For $\mathsf{id} \in \mathsf{dict}, \mathsf{id} \notin M$, the challenger sets $b_{i,\mathsf{id}} \leftarrow 1$.
  - The challenger runs

$$\mathsf{ret}_{\mathsf{id}}^{(i)} \leftarrow \mathsf{Client.Read}(\mathsf{config}, \mathsf{dgst}_i, i)$$

    for each $\mathsf{id} \in \mathsf{dict}$ with $\mathcal{A}$ playing the role of the server.
  - Initialize dictionary dict_i with the contents of dict.
  - The adversary sends "continue" or "finish". If the challenger receives "finish", break out of the loop. Otherwise, $i \leftarrow i + 1$ and continue to the next round.
- The challenger runs

$$b_{i,\mathsf{id}} \leftarrow \mathsf{Client.Monitor}(\mathsf{config}, \mathsf{dgst}_i, \mathsf{id}, \mathsf{dict}[\mathsf{id}], i)$$

  for each $\mathsf{id} \in \mathsf{dict}$ with $\mathcal{A}$ playing the role of the server.
- The challenger sets $i \leftarrow i + 1$ and gets (dgst_i, π_i) from $\mathcal{A}$.
- The challenger runs

$$\mathsf{ret}_{\mathsf{id}}^{(i)} \leftarrow \mathsf{Client.Read}(\mathsf{config}, \mathsf{dgst}_i, i)$$

  for each $\mathsf{id} \in \mathsf{dict}$ with $\mathcal{A}$ playing the role of the server.

Let $R \leftarrow i - 1$ (the number of rounds). The adversary wins if:

1. $\mathsf{Auditor.Audit}(\mathsf{config}, \mathsf{dgst}_i, \mathsf{dgst}_{i+1}, \pi_{i+1}) = 1$ and $w_i = 1$ for $i \in [R]$,
2. for all $\mathsf{id} \in \mathsf{dict}$ and $i \in [R]$, $b_{i,\mathsf{id}} = 1$, and
3. there exists $i \in [R]$, $\mathsf{id} \in \mathsf{dict}$ such that $\mathsf{dict}_i[\mathsf{id}] \neq \mathsf{ret}_{\mathsf{id}}^{(i+1)}$.

Figure 3.14: Security game for a configurable transparency log.

Experiment 3.6. The below security game is modified from Experiment 3.5 so that the adversary only wins if the challenger outputs an old value for a given identifier as a result of a read.
The game proceeds as in Experiment 3.5 and only the win condition is different.
Let $R \leftarrow i - 1$ (the number of rounds). The adversary wins if:

1. $\mathsf{Auditor.Audit}(\mathsf{config}, \mathsf{dgst}_i, \mathsf{dgst}_{i+1}, \pi_{i+1}) = 1$ and $w_i = 1$ for $i \in [R]$,
2. for all $\mathsf{id} \in \mathsf{dict}$ and $i \in [R]$, $b_{i,\mathsf{id}} = 1$, and
3. there exists $i, j \in [R]$, $\mathsf{id} \in \mathsf{dict}$ such that $\mathsf{dict}_j[\mathsf{id}] = \mathsf{ret}^i_{\mathsf{id}}$ and $i > j$.

Figure 3.15: Hybrid security game, modified from Experiment 3.5.

Experiment 3.7. Let $\mathcal{A}_\triangle$ be the adversary that can win the difference in the games of Experiment 3.5 and Experiment 3.6. The game proceeds as in Experiment 3.5 and only the win condition is different.
Let $R \leftarrow i - 1$ (the number of rounds). The adversary $\mathcal{A}_\triangle$ wins if:

1. $\mathsf{Auditor.Audit}(\mathsf{config}, \mathsf{dgst}_i, \mathsf{dgst}_{i+1}, \pi_{i+1}) = 1$ and $w_i = 1$ for $i \in [R]$,
2. for all $\mathsf{id} \in \mathsf{dict}$ and $i \in [R]$, $b_{i,\mathsf{id}} = 1$, and
3. there exists $i \in [R]$, $\mathsf{id} \in \mathsf{dict}$ such that $\mathsf{dict}_i[\mathsf{id}] \neq \mathsf{ret}^{(i+1)}_{\mathsf{id}}$ and $\forall j < i$, $\mathsf{dict}_j[\mathsf{id}] \neq \mathsf{ret}^{(i+1)}_{\mathsf{id}}$.

Figure 3.16: Security game representing the difference between Experiment 3.5 and Experiment 3.6.

and auditing checks passed. In Client.Read, Client.Write, Client.Monitor, and Auditor.Audit, there are a series of checks performed by the client and the auditor that the adversary must clear before convincing the client to read an old value (stored in the variable ok). This means that the adversary did one of the following:

(1) Generated a prefix tree proof for the query base tree attesting to $(\mathsf{id}, \mathsf{val}')$ but the query base tree contains $(\mathsf{id}, \mathsf{val})$:
$\underline{\mathsf{Client.Read}(\mathsf{config},\ \mathsf{dgst},\ \mathsf{id},\ t)}$
3: ...
4: $\mathsf{val}',\ \mathsf{Path}_{\mathsf{id}} \leftarrow \mathsf{Server}[\alpha].QueryBT.\mathsf{GetPath}(\mathsf{id})$
5: $\mathsf{ok} \leftarrow \mathsf{ok} \bigwedge \mathsf{CheckPath}(\mathsf{root},\ \mathsf{id},\ \mathsf{vals},\ \mathsf{Path}_{\mathsf{id}})$
6: ...
By Lemma 3.2, the adversary cannot do this without finding a hash collision.

(2) Generated a prefix tree proof for an update tree attesting to $(\mathsf{id}, \mathsf{val}')$, but the update tree did not contain such an update:
$\underline{\mathsf{Client.Read}(\mathsf{config},\ \mathsf{dgst},\ \mathsf{id},\ t)}$

$\mathcal{B} = (\mathcal{A}_\triangle, \text{Challenger}_{\text{EUF-CMA}})$

- $\text{Challenger}_{\text{EUF-CMA}}$ is the challenger in the EUF-CMA game.
- $\mathcal{B}$ acts as the challenger in the game played with $\mathcal{A}_\triangle$ (Experiment 3.7).
- $\mathcal{B}$ receives public key pk from $\text{Challenger}_{\text{EUF-CMA}}$.
- $\mathcal{B}$ sets $\mathsf{id}_{\mathsf{pk}}$ to be empty.
- $\mathcal{B}$ modifies $\mathsf{Sign}(\mathsf{sk}, \mathsf{val})$ to be $\mathcal{B}$ sending val as the chosen message to $\text{Challenger}_{\text{EUF-CMA}}$ and receiving back σ when $\mathsf{sk} = \mathsf{pk}$. For all other values of sk, $\mathcal{B}$ runs Sign unmodified to produce a signature (since it knows the signing key).
- Each round i proceeds as follows:
  - $\mathcal{B}$ receives $(\mathsf{id}_1, \mathsf{val}_1, \ldots, \mathsf{id}_n, \mathsf{val}_n)$ from $\mathcal{A}_\triangle$.
  - For each $j \in [n]$, $\mathcal{B}$:
    - If $\mathsf{id}_{\mathsf{pk}}$ is empty, then set $\mathsf{id}_{\mathsf{pk}}$ to be id_j.
    - If $\mathsf{id}_j = \mathsf{id}_{\mathsf{pk}}$, $\mathcal{B}$ then runs (with $\mathcal{A}_\triangle$ playing the role of the server)

      $$w_i \leftarrow w_i \wedge \mathsf{Client.Write}(\mathsf{config}, \mathsf{pk}, \mathsf{id}_j, \mathsf{val}_j, i)$$

    - Otherwise, $\mathcal{B}$ checks if there exists $\mathsf{sk}_{\mathsf{id}_j}$. If not, the $\mathcal{B}$ samples a new signing key $\mathsf{sk}_{\mathsf{id}_j}$. Then the adversary plays the role of the server to run:

      $$w_i \leftarrow w_i \wedge \mathsf{Client.Write}(\mathsf{config}, \mathsf{sk}_{\mathsf{id}_j}, \mathsf{id}_j, \mathsf{val}_j, i)$$

    - Sets $\mathsf{dict}[\mathsf{id}_j] \leftarrow \mathsf{val}_j$.
  - The rest of the round proceeds as defined in Experiment 3.5.
- The rest of the game proceeds as defined in Experiment 3.5.
- If the adversary $\mathcal{A}_\triangle$ wins, there exists $i \in [R]$, $\mathsf{id} \in \mathsf{dict}$ such that $\mathsf{dict}_i[\mathsf{id}] \neq \mathsf{ret}_{\mathsf{id}}^{(i+1)}$ and $\forall j < i$, $\mathsf{dict}_j[\mathsf{id}] \neq \mathsf{ret}_{\mathsf{id}}^{(i+1)}$. If $\mathsf{id} = \mathsf{id}_{\mathsf{pk}}$, $\mathcal{B}$ simply sends $\mathsf{ret}_{\mathsf{id}}^{(i+1)}$ and its associated signature to $\text{Challenger}_{\text{EUF-CMA}}$.

With an $\mathcal{A}_\triangle$ that can win Experiment 3.7, wins the EUF-CMA game with advantage $1/\mathsf{poly}(\cdot)$ because the invalid read that $\mathcal{A}_\triangle$ forges a signature and one of the ids uses the signatures from $\text{Challenger}_{\text{EUF-CMA}}$.

Figure 3.17: Adversary $\mathcal{B}$ that can be constructed from $\mathcal{A}_\triangle$ to win the EUF-CMA game.

22: ...
23: $\mathsf{val}_u,\ \mathsf{Path}_{\mathsf{id},u} \leftarrow \mathsf{tree}_u.\mathsf{GetPath}(\mathsf{id})$
24: $\mathsf{ok} \leftarrow \mathsf{ok} \bigwedge \mathsf{CheckPath}(\mathsf{root}_u,\ \mathsf{id},\ \mathsf{val}_u,\ \mathsf{Path}_{\mathsf{id},u})$
25: ...

By Lemma 3.2, the adversary cannot do this without finding a hash collision.

(3) Generated a chronological tree proof attesting that the most recent addition to the update log is a prefix tree containing $(\mathsf{id}, \mathsf{val}')$:
$\mathsf{Client.Read}(\mathsf{config},\ \mathsf{dgst},\ \mathsf{id},\ t)$

19: ...
20: $(\mathsf{tree}_u, \mathsf{ok}_u) \leftarrow \mathsf{dgst}[\alpha].\mathit{UpdateLog}[u]^*$
21: $\mathsf{ok} \leftarrow \mathsf{ok} \bigwedge \mathsf{ok}_u$
22: ...

By Lemma 3.3, the adversary cannot do this without finding a hash collision.

(4) Generated an extension proof attesting that the update log containing a prefix tree containing $(\mathsf{id}, \mathsf{val}')$ is an extension of an update log containing a prefix tree containing $(\mathsf{id}, \mathsf{val})$:
$\mathsf{Client.Write}(\mathsf{config},\ \mathsf{sk},\ \mathsf{id},\ \mathsf{val},\ t) \rightarrow \{0,1\}$

2: ...
3: $\mathsf{Server}[\alpha].\mathsf{Write}(\mathsf{id},\ (\mathsf{val},\ \sigma))$
4: Wait until epoch $t+1$
5: $\mathsf{dgst} \leftarrow \mathsf{Auditor.GetDigest}(t+1)$
6: $\mathsf{val}_{\mathsf{latest}} \leftarrow \mathsf{Client.Read}(\mathsf{config},\ \mathsf{dgst},\ \mathsf{id},\ t+1)$
7: ...

$\mathsf{Auditor.Audit}(\mathsf{config},\ \mathsf{dgst}_{\mathsf{old}},\ \mathsf{dgst}_{\mathsf{new}},\ \pi) \rightarrow \{0,1\}$:

11: ...
12: **if** $\mathsf{dgst}_{\mathsf{old}}[\alpha].t \mod \mathsf{config}.t_v = \mathsf{dgst}_{\mathsf{new}}[\alpha].t \mod \mathsf{config}.t_v$ **then**
13: $\quad \mathsf{ok} \leftarrow \mathsf{ok} \bigwedge \mathsf{VerifyExtension}(\pi[\alpha].\mathit{ul}, \mathsf{dgst}_{\mathsf{old}}[\alpha].\mathit{ulroot},\ \mathsf{dgst}_{\mathsf{new}}[\alpha].\mathit{ulroot})$
14: **end if**
15: ...

By Lemma 3.4, the adversary cannot do this without finding a hash collision.

(5) Created a collision in the hash chain of base prefix trees (so the client did not detect the equivocation in a later epoch):
$\mathsf{Auditor.Audit}(\mathsf{config},\ \mathsf{dgst}_{\mathsf{old}},\ \mathsf{dgst}_{\mathsf{new}},\ \pi) \rightarrow \{0,1\}$:

5: ...
6: $h' \leftarrow \mathsf{dgst}_{\mathsf{old}}[\alpha].\mathit{HashChain}$
7: **for** $n \in \pi[\alpha].\mathit{hc}$ **do**
8: $\quad h' \leftarrow H(n,\ h')$
9: **end for**

10: $\mathsf{ok} \leftarrow \mathsf{ok} \bigwedge h' \stackrel{?}{=} \mathsf{dgst}_{\mathsf{new}}[\alpha].HashChain$
11: ...

Thus, if the adversary does win, then we can construct an adversary $\mathcal{C}$ that can find a collision in the hash function.

Case 2. We now argue security for when $\mathsf{agghistory} = \mathsf{true}$. Assume that the challenger reads an old value val' for id when the last write to id was val ($\mathsf{agghistory} = \mathsf{true}$), and that all monitoring and auditing checks passed. This means that the adversary did one of the following:

* (1) through (4) are the same as in Case 1.

(5) Each read and monitor invocation query a number of prefix trees that have been generated since the most recent write, and for a given identifier id, if the monitor call happens after the read call, then the set of prefix trees queried must intersect.

Client.Read(config, dgst, id, t)

6: ...
7: **for** $\mathsf{root}_h \in \mathsf{Server}[\alpha].HistoryForest.\mathsf{roots}$ **do**
8: $\quad t_h \leftarrow$ verification period root_h corresponds to
9: $\quad \mathsf{val}_h, \mathsf{Path}_{\mathsf{id},h} \leftarrow$ $\mathsf{Server}[\alpha].HistoryForest[t_h].\mathsf{GetPath}(\mathsf{id})$
10: $\quad \mathsf{ok} \leftarrow \mathsf{ok} \bigwedge \mathsf{CheckPath}(\mathsf{root}_h, \mathsf{id}, \mathsf{val}_h, \mathsf{Path}_{\mathsf{id},h})$
11: $\quad \mathsf{ok} \leftarrow \mathsf{ok} \bigwedge \mathsf{val}_h.\mathsf{vals} \stackrel{?}{\subseteq} \mathsf{val}.\mathsf{vals}$
12: **end for**
13: ...

Client.Monitor(config, dgst, id, val, t) $\rightarrow \{0, 1\}$

7: ...
8: $t' \leftarrow$ verification period id was updated to val
9: **for** $t_l \in$ path from t' to root in $\mathsf{Server}[\alpha].HistoryForest$ **do**
10: $\quad \mathsf{dgst}_l \leftarrow \mathsf{Auditor.GetDigest}(t_l)$
11: $\quad \mathsf{val}', \mathsf{Path} \leftarrow \mathsf{Server}[\alpha].HistoryForest[t_l].\mathsf{GetPath}(\mathsf{id})$
12: $\quad \mathsf{ok} \leftarrow \mathsf{ok} \bigwedge \mathsf{CheckPath}(\mathsf{dgst}_l[\alpha].vroot, \mathsf{id}, \mathsf{val}', \mathsf{Path})$
13: $\quad \mathsf{ok} \leftarrow \mathsf{ok} \bigwedge \mathsf{val}' \stackrel{?}{=} \mathsf{val}$
14: **end for**
15: ...

Because the prefix trees are stored in a chronological forest, if the adversary can convince the challenger that this set of prefix trees does not overlap, then it has found a collision in the hash function.

Given an overlapping set of prefix trees, the adversary then must produce a prefix tree proof attesting to $(\mathsf{id}, \mathsf{val}')$ when the last write to id was for $\mathsf{val} \neq \mathsf{val}'$. Based on the security

of prefix trees (Lemma 3.2), an adversary that can do this has found a collision in the hash function.

If the adversary does win in this case (when $\mathsf{agghistory} = \mathsf{true}$), then we can construct an adversary $\mathcal{C}$ that can find a collision in the hash function. This completes the proof. □

3.10 Conclusion

In this chapter, we introduced LegoLog, a configurable transparency log that can *automatically* output a suitable log design given a workload description. It enables non-experts to use transparency logs, thus enabling more applications to leverage them. We showed that LegoLog provides configurability while performing comparably to special-purpose transparency logs.

Part II

Scaling

Chapter 4

Snoopy: A scalable oblivious storage

4.1 Introduction

Organizations increasingly outsource sensitive data to the cloud for better convenience, cost-efficiency and availability [101, 176, 279]. Encryption cannot fully protect this data: how the user accesses data (the "access pattern") can leak sensitive information to the cloud [45, 82, 124, 152, 161, 171]. For example, the frequency with which a doctor accesses a medication database might reveal a patient's diagnosis.

Oblivious object stores allow clients to outsource data to a storage server without revealing access patterns to the storage server. A rich line of work has shown how to build efficient oblivious RAMs (ORAMs), which can be used to construct oblivious object stores [29,47,72,116,240,263,269, 292–294,322]. In order to be practical for applications, oblivious storage must provide many of the same properties as plaintext storage. Prior work has shown how to reduce latency [217,263,294], scale to large data sizes via data parallelism [199], and improve request throughput [72,269,322]. Despite this progress, leveraging task parallelism to *scale for high-throughput workloads* remains an open problem: existing oblivious storage systems do not scale.

Identifying the scalability bottleneck. Scalability bottlenecks are system components that must perform computation for every request and cannot be parallelized. These bottlenecks limit the overall system throughput; once their maximum throughput has been reached, adding resources to the system no longer improves performance. To scale, plaintext object stores traditionally shard objects across servers, and clients can route their queries to the appropriate server. Unfortunately, this approach is insecure for oblivious object stores because it reveals the mapping of objects to partitions [45, 124, 152, 161, 171]. For example, if clients query different shards, the attacker learns that the requests were for different objects.

To understand why scaling oblivious storage is hard, we examine two properties oblivious storage systems traditionally satisfy. First, systems typically maintain a dynamic mapping (hidden from the untrusted server) between the logical layout and physical layout of the outsourced data. Clients must look up their logical key using the freshest mapping and remap it to a new location after every access, creating a central point of coordination. Second, for efficient access,

oblivious systems typically store data in a hierarchical or tree-like structure, creating a bottleneck at the root [263, 293, 294].

Thus high-throughput oblivious storage systems are all built on hierarchical [293] or tree-like [263, 294] structures and either require a centralized coordination point (e.g., a query log [47, 322] or trusted proxy [29, 72, 269, 292]) or inter-client communication [36]. We ask: *How can we build an oblivious object store that handles high throughput by scaling in the same way as a plaintext object store?*

Removing the scalability bottleneck. In this chapter, we propose Snoopy (scalable nodes for oblivious object repository), a high-throughput oblivious storage system that scales similarly to a plaintext storage system. While our system is secure for any workload, we design it for high-throughput workloads. Specifically, we develop techniques for grouping requests into equal-sized batches for each partition regardless of the underlying request distribution and with minimal cover traffic. These techniques enable us to efficiently partition and securely distribute every system component without prohibitive coordination costs.

Like prior work, Snoopy leverages hardware enclaves for both performance and security [3, 217, 270]. Hardware enclaves makes it possible to (1) deploy the entire system in a public cloud; (2) reduce network overheads, as private and public state can be located on the same machine; and (3) support multiple clients without creating a central point of attack. This is in contrast with the traditional trusted proxy model (Figure 4.1), which can be both a deployment headache and a scalability concern. Hardware enclaves do not entirely solve the problem of hiding access patterns for oblivious storage: enclave side channels allow attackers to exploit data-dependent memory accesses to extract enclave secrets [38, 130, 186, 188, 220, 275, 312, 324]. To defend against these attacks, we must ensure that all algorithms running inside the enclave are oblivious, meaning that memory accesses are data-independent. Existing work targets latency-sensitive deployments [3, 217, 270] and is prohibitively expensive for the concurrent, high-throughput deployment we target. We instead leverage our oblivious partitioning scheme to design new algorithms tailored to our setting.

We experimentally show that Snoopy scales to achieve high throughput. The state-of-the-art oblivious storage system Obladi [72] reaches a throughput of 6,716 reqs/sec with average latency under 80ms for two million 160-byte objects and cannot scale beyond a proxy machine (32 cores) and server machine (16 cores). In contrast, Snoopy uses 18 4-core machines to scale to a throughput of 92K reqs/sec with average latency under 500ms for the same data size, achieving a 13.7× improvement over Obladi. We report numbers with 18 machines due to cloud quota limits, not because Snoopy stops scaling. We formally prove the security of the entire Snoopy system, independent of the request load.

4.1.1 Summary of techniques

Snoopy is comprised of two types of entities: *load balancers* and *subORAMs* (Figure 4.1). Load balancers assemble batches of requests, and subORAMs, which store data partitions, process the requests. In order to securely achieve horizontal scaling, we must consider how to design both

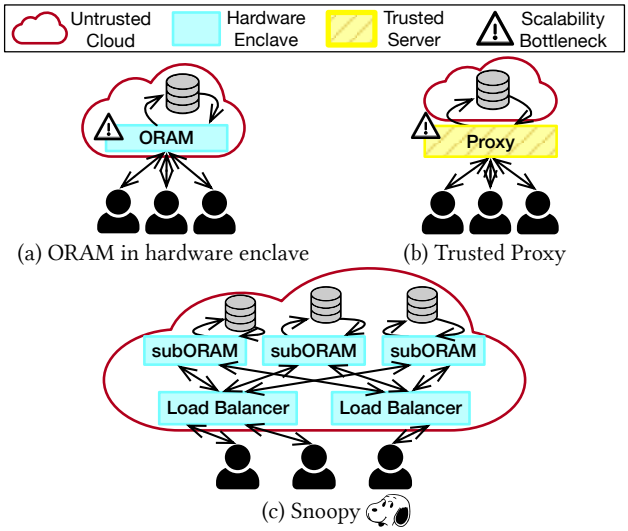


Figure 4.1: Different oblivious storage system architectures: (a) ORAM in a hardware enclave is bottlenecked by the single machine, (b) ORAM with a trusted proxy is bottlenecked by the proxy machine, and (c) Snoopy can continue scaling as more subORAMs and load balancers are added to the system.

the load balancer and subORAM to (1) leverage efficient oblivious algorithms to defend against memory-based side-channel attacks, and (2) be easy to partition without incurring coordination costs.

Challenge #1: Building an oblivious load balancer. To protect the contents of the requests, our load balancer design must guarantee that (1) the batch structure leaks no information about the requests, and (2) the process of constructing these batches is oblivious and efficient. Furthermore, we need to design our oblivious algorithm such that we can add load balancers without incurring additional coordination costs.

Approach. We build an efficient, oblivious algorithm that groups requests into batches without revealing the mapping between requests and subORAMs. We size batches using only public information, ensuring that the load balancer never drops requests and the batch size does not leak information. Our load balancer design enables us to run load balancers independently and in parallel, allowing Snoopy to scale past the capacity of a single load balancer (§4.4).

Challenge #2: Designing a high-throughput subORAM. To ensure that Snoopy can achieve high throughput, we need a subORAM design that efficiently processes large batches of requests and defends against enclave side-channel attacks. Existing ORAMs that make use of hardware enclaves [3,217,270] only process requests sequentially and are a poor fit for the high-throughput scenario we target.

Approach. Rather than building batching support into an existing ORAM scheme, we design a new ORAM that only supports batched accesses. We observe that in the case where data is partitioned over many subORAMs, a single scan amortized over a large batch of requests is concretely cheaper than servicing the batch using ORAMs with polylogarithmic access costs [3, 217, 270], particularly in the hardware enclave setting. We leverage a specialized data structure to process batches efficiently and obliviously in a single linear scan (§4.5).

Challenge #3: Choosing the optimal configuration. The design of Snoopy makes it possible to scale the system by adding both load balancers and subORAMs. An application developer needs to know how to configure the system to meet certain performance targets while minimizing cost.

Approach. To solve this problem, we design a planner that, given a minimum throughput, maximum average latency, and data size, outputs a configuration minimizing cost (§4.6).

Limitations. Snoopy is designed specifically to overcome ORAM's scalability bottleneck to support high-throughput workloads, as solutions already exist for low-throughput, low-latency workloads [263, 294]. In the low-throughput regime, although Snoopy is still secure, its latency will likely be higher than that of non-batching systems like ConcurORAM [47], TaoStore [269], or PrivateFS [322]. For large data sizes and low request volume, a system like Shroud [199] will leverage resources more efficiently. Snoopy can use a different, latency-optimized subORAM with a shorter epoch time if latency is a priority. We leave for future work the problem of adaptively switching between solutions that are optimal under different workloads.

4.2 Security and correctness guarantees

We consider a cloud attacker that can:

- control the entire cloud software stack outside the enclave (including the operating system),
- view (encrypted) network traffic arriving at and within the cloud (including traffic sent by clients and message timing),
- view or modify (encrypted) memory outside the enclaves in the cloud, and
- observe access patterns between the enclaves and external memory in the cloud.

We design Snoopy on top of an abstract enclave model where the attacker controls the software stack outside the enclave and can observe memory access patterns but cannot learn the contents of the data inside the processor. Snoopy can be used with any enclave implementation [34, 71, 187]; we chose to implement Snoopy on Intel SGX as it is publicly available on Microsoft Azure. Enclaves do not hide memory access patterns, enabling a large class of side-channel attacks, including but not limited to cache attacks [38, 130, 220, 275], branch prediction [188], paging-based attacks [312, 324], and memory bus snooping [186]. By using oblivious algorithms, Snoopy defends against this class of attacks. Snoopy does not defend against enclave integrity attacks such as rollback [241] and transient execution attacks [56, 257, 274, 309, 310, 313, 314], which we discuss in greater detail below.

We defend against memory access patterns to both data and code by building oblivious algorithms on top of an oblivious "compare-and-set" operator. While our source code defends against access patterns to code, we do not ensure that the final binary does, as other factors like compiler optimizations and cache replacement policies may leak information (existing solutions may be employed here [125, 195]).

Timing attacks. A cloud attacker has access to three types of timing information: (1) when client requests arrive, (2) when inter-cloud processing messages are sent/received, and (3) when client responses are sent. Snoopy allows the attacker to learn (1). In theory, these arrival times can leak data, and so we could hide when clients send requests and how many they send by requiring clients to send a constant number of requests at predefined time intervals [10]; we do not take this approach because of the substantial overhead and because, for some applications, clients may not always be online. Snoopy ensures that (2) and (3) do not leak request contents; the time to execute a batch depends entirely on public information, as defined in §4.2.1.

Data integrity and protection against rollback attacks. Snoopy guarantees the integrity of the stored objects in a straightforward way: for memory within the enclave, we use Intel SGX's built-in integrity tree, and for memory outside the enclave, we store a digest of each block inside the enclave. We assume that the attacker cannot roll back the state of the system [241]. We discuss how Snoopy can integrate with existing rollback-attack solutions in §4.9.

Attacks out of scope. We build on an abstract enclave model where the attacker's power is limited to viewing or modifying external memory and observing memory access patterns (we formalize this as an ideal functionality in §4.12). Any attack that breaks the abstract enclave model is out of scope and should be addressed with techniques complementary to Snoopy. For example, we do not defend against leakage due to power consumption [59, 227, 297] or denial-of-service attacks due to memory corruptions [126, 154]. We additionally consider transient execution attacks [56, 257, 274, 309, 310, 313, 314] to be out of scope; in many cases, these have been patched by the enclave vendor or the cloud provider. These attacks break Snoopy's assumptions (and hence guarantees) as they allow the attacker to, in many cases, extract enclave secrets. We note that, Snoopy's design is not tied to Intel SGX, and also applies to academic enclaves like MI6 [34], Keystone [187], or Sanctum [71], which avoid many of the drawbacks of Intel SGX.

We also do not defend against denial-of-service attacks; the attacker may refuse queries or even delete the clients' data.

Clients. For simplicity, in the rest of this chapter, we describe the case where all clients are honest. We make this simplification to focus on protecting client requests from the server, a technical challenge that motivates our techniques. However, in practice, we might not want to trust every client with read and write access to every object in the system. Adding access-control lookups to our system is fairly straightforward and requires an oblivious lookup in an access-control matrix to check a client's privileges for a given object. We can perform this check obliviously via a recursive lookup in Snoopy (we describe how this works in §4.14). Supporting access control in Snoopy ensures that compromised clients cannot read or write data that they do

not have access to. Furthermore, if compromised clients collude with the cloud, the cloud does not learn anything beyond the public information that it already learns (specified in §4.2.1) and the results of read requests revealed by compromised clients.

Linearizability. Because we handle multiple simultaneous requests, we must provide some ordering guarantee. Snoopy provides linearizability [137]: if one operation happens after another in real time, then the second will always see the effects of the first (see §4.4.3 for how we achieve this). We include a linearizability proof in §4.13.

4.2.1 Formalizing security

We formalize our system and prove its security in §4.12. We build our security definition on an enclave ideal functionality (representing the abstract enclave model), which provides an interface to load a program onto a network of enclaves and then execute that program on an input. Execution produces the program output, as well as a *trace* containing the network communication and memory access patterns generated as a result of execution (what the adversary has access to in the abstract enclave model).

The Snoopy protocol allows the attacker to learn public information such as the number of requests sent by each client, request timing, data size (number of objects and object size), and system configuration (number of load balancers and subORAMs); this public information is standard in oblivious storage. Snoopy protects private information, including the data content and, for each request, the identity of the requested object, the request type, and any read or write content. To prove security, we show how to simulate all accesses based solely on public information (as is standard for ORAM security [116]). Our construction is secure if an adversary cannot distinguish whether it is interacting with enclaves running the real Snoopy protocol (the "real" experiment) or an ideal functionality that interacts with enclaves running a simulator program that only has access to public information (the "ideal" experiment) from the trace generated by execution. We now informally define these experiments, delegating the formal details to Figure 4.16.

Real and ideal experiments (informal). In the real experiment, we load the protocol Π (either our Snoopy protocol or our subORAM protocol, depending on what we are proving security of) onto a network of enclaves and execute the initialization procedure (the adversary can view the resulting trace). Then, the adversary can run the batch access protocol specified by Π on any set of queries and view the trace. The adversary repeats this process a polynomial number of times before outputting a bit.

The ideal experiment proceeds in the same way as the real experiment, except that, instead of interacting with enclaves running Π, the adversary interacts with an ideal functionality that in turn interacts with the enclaves running the simulator program. The adversary can view the traces generated by the simulator enclaves. The goal of the adversary is to distinguish between these experiments. We describe both experiments more formally in Figure 4.16.

Using these experiments, we present our security definition:

Definition 4.1. The oblivious storage scheme Π is secure if for any non-uniform probabilistic polynomial-time (PPT) adversary Adv, there exists a PPT Sim such that

$$\left|\Pr\left[\mathbf{Real}^{\text{OSTORE}}_{\Pi,\text{Adv}}(\lambda)=1\right]-\Pr\left[\mathbf{Ideal}^{\text{OSTORE}}_{\text{Sim},\text{Adv}}(\lambda)=1\right]\right|\leq \text{negl}(\lambda)$$

where λ is the security parameter, the real and ideal experiments are defined informally above and formally in Figure 4.16, and the randomness is taken over the random bits used by the algorithms of Π, Sim, and Adv.

We prove security in a modular way, which enables future systems to make standalone use of our subORAM design. We note that our subORAM scheme is secure only if the batch received contains unique requests (this property is guaranteed by our load balancer). We describe these requirements formally and prove security in Definition 4.2. We prove the security of Snoopy using any subORAM scheme that is secure under this modified definition.

Theorem 4.1. *Given a two-tiered oblivious hash table [49], an oblivious compare-and-set operator, and an oblivious compaction algorithm, the subORAM scheme described in §4.5 and formally defined in Figure 4.18 is secure according to Definition 4.2.*

Theorem 4.2. *Given a keyed cryptographic hash function, an oblivious compare-and-set operator, an oblivious sorting algorithm, an oblivious compaction algorithm, and an oblivious storage scheme (secure according to Definition 4.2), Snoopy, as described in §4.4 and formally defined in Figure 4.20, is secure according to Definition 4.1.*

All of the tools we use in the above theorems can be built from standard cryptographic assumptions. We prove both theorems in §4.12.

4.3 System overview

To motivate the design of our system, we begin by describing several solutions that do *not* work for our purposes.

Attempt #1: Scalable but not secure. Sharding is a straightforward way to achieve horizontal scaling. Each server maintains a separate ORAM for its data shard, and the client queries the appropriate server. This simple solution is insecure: repeated accesses to the same shard leaks query information. For example, if two clients query different servers, the attacker learns that they requested different objects.

Attempt #2: Secure but not scalable. To fix the above problem, we could remap an object to a different partition after it is accessed, similar to how single-server ORAMs remap objects after accesses [263, 294]. A central proxy running on a lightweight, trusted machine keeps a mapping of objects to servers. The client sends its request to the proxy, which then accesses the server currently storing that object and remaps that object to a new server [29, 292]. While this solution is secure, this single proxy is a scalability bottleneck. Every request must use the most up-to-date mapping for security; otherwise, requests might fail and re-trying them will leak when the

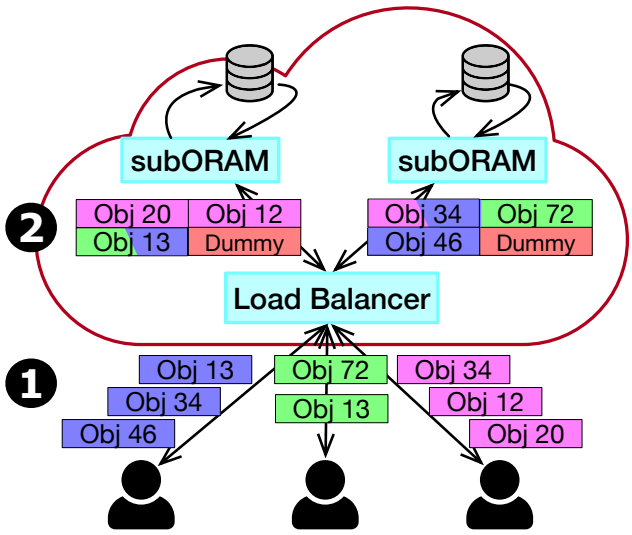


Figure 4.2: Secure distribution of requests in Snoopy. ❶ The load balancer receives requests from clients. ❷ At the end of the epoch, the load balancer generates a batch of requests for each subORAM, padding with dummy requests as necessary.

requested object was last accessed. Therefore, all requests must be serialized at the proxy, and so the proxy's throughput limits the system's throughput.

Our approach. We achieve the scalability of the first approach and the security of the second approach. To efficiently scale, we exploit characteristics of the high-throughput regime to develop new techniques that allow us to provide security *without* remapping objects across partitions. These techniques enable us to send equal-sized batches to each partition while both (1) hiding the mapping between requests and partitions (for security), and (2) ensuring that requests are distributed somewhat equally across partitions (for scalability).

4.3.1 System architecture

Snoopy's system architecture (Figure 4.2) consists of clients (running on private machines) and, in the public cloud, load balancers and subORAMs (running on hardware enclaves). All communication is encrypted using an authenticated encryption scheme with a nonce to prevent replay attacks. We establish all communication channels using remote attestation so that clients are confident that they are interacting with legitimate enclaves running Snoopy [17].

The role of the *load balancer* is to partition requests received during the last epoch into equally sized batches while providing security and efficiency (§4.4). In order to horizontally scale the load balancer, each load balancer must be able to operate independently and without coordination. The role of the *subORAM* is to manage a data partition, storing the current version of the data and executing batches of requests from the load balancers (§4.5). Snoopy can be deployed using

any oblivious storage scheme for hardware enclaves [3, 217, 270] as a subORAM. However, our subORAM design is uniquely tailored to our target workload and end-to-end system design.

4.3.2 Real-world applications

Snoopy is valuable for applications that need a high-throughput object store for confidential data, including outsourced file storage [3], cloud electronic health records, and Signal's private contact discovery [204]. Privacy-preserving cryptocurrency light clients can also benefit from Snoopy. These allow lightweight clients to query full nodes for relevant transactions [206]. Maintaining many ORAM replicas is not enough to support high-throughput blockchains because each replica needs to keep up with the system state. As blockchains continue to increase in the throughput [267, 290], oblivious storage systems like Obladi [72] with a scalability bottleneck simply cannot keep up.

Snoopy can also enable private queries to a transparency log; for example, Alice could look up Bob's public key in a key transparency log [120, 208] without the server learning that she wants to talk to Bob. A key transparency log should support up to a billion users, making high throughput critical [119].

4.4 Oblivious load balancer

In this section, we detail the design of the load balancer, focusing on how batching can be used to hide the mapping between requests and subORAMs at low cost (§4.4.1), designing oblivious algorithms to efficiently generate batches while protecting the contents of the requests (§4.4.2), and scaling the load balancer across machines (§4.4.3).

4.4.1 Setting the batch size

To provide security, we need to ensure that constructing batches leaks no information about the requests. Specifically, we must guarantee that (1) the size of batches leaks no information, and (2) the process of constructing batches is similarly oblivious. We focus on (1) now and discuss (2) in §4.4.2. For security, we need to ensure that the batch size B depends only on public information visible to the attacker: namely, the number of requests R and number of subORAMs S, but not the contents of these requests. Therefore, we define B as a function $B = f(R, S)$ that outputs an efficient yet secure batch size for R requests and S subORAMs. Each subORAM will receive B requests. Because R is not fixed across epochs (requests can be bursty), B can also vary across epochs.

In choosing how to define this function f, we need to (1) ensure that requests are not dropped, and (2) minimize the overhead of dummy requests. Ensuring that requests are not dropped is critical for security: if a request is dropped, the client will retry the request, and an attacker who sees a client participate in two consecutive epochs may infer that a request was dropped, leaking information about request contents. Minimizing the overhead of dummy requests is important

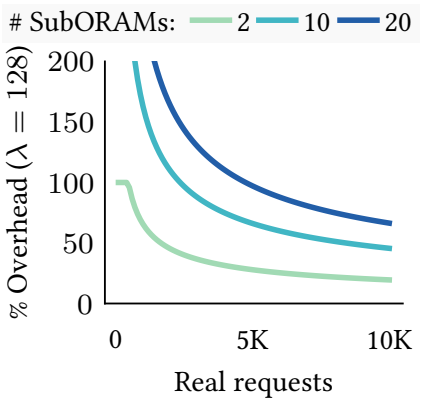


Figure 4.3: Dummy request overhead. A 50% overhead means for every two real requests there is one dummy request.

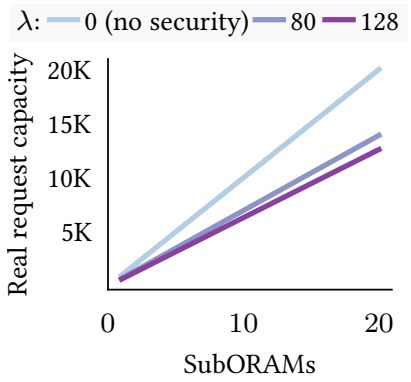


Figure 4.4: The total real request capacity of our system for an epoch, assuming $\leq$1K requests per subORAM per epoch.

for scalability. A simple way to satisfy security would be to set $f(R, S) = R$; this ensures that even if all the requests are for the same object, no request was potentially dropped. However, this approach is not scalable because every subORAM would need to process a request for every client request. We refine this approach in two steps.

Deduplication to address skew. When assembling a batch of requests, the load balancer can ensure that all requests in a batch are for distinct objects by aggregating reads and writes for the same object (for writes, we use a "last write wins" policy) [72]. Deduplication allows us to combat workload skew. If the load balancer receives many requests for object A and a single request for object B, the load balancer only needs to send one request for object A and one request for object B. Deduplication simplifies the problem statement; we now need to distribute a batch of at most R *unique* requests across subORAMs. This reframing allows us to achieve security with high probability for $f(R, S) < R$ if we distribute objects randomly across subORAMs, as we now do not have to worry about the case where all requests are for the same object.

Choosing a batch size. Given R requests and S subORAMs, we need to find the batch size B such that the probability that any subORAM receives more than B requests is negligible in our security parameter λ. Like many systems that shard data, we use a hash function to distribute objects across subORAMs, allowing us to recast the problem of choosing B as a balls-into-bins problem [255]: we have R balls (requests) that we randomly toss into S bins (subORAMs), and we must find a bin size B (batch size) such that the probability that a bin overflows is negligible. We add balls (dummy requests) to each of the S bins such that each bin contains exactly B balls.

Using the balls-into-bins model, we can start to understand how we expect R and S to affect B. As we add more balls to the system ($R \uparrow$), it becomes more likely for the balls to be distributed evenly over every bin, and the ratio of dummy balls to original balls decreases. Conversely, as we add more bins to the system ($S \uparrow$), we need to proportionally add more dummy balls. We validate

this intuition in Figure 4.3 and Figure 4.4. Figure 4.3 shows that as the total number of requests R increases, the percent overhead due to dummy requests decreases. Thus larger batch sizes are preferable, as they minimize the overhead introduced by dummy requests. Figure 4.4 illustrates how adding more subORAMs increases the total request capacity of Snoopy, but at a slower rate than a plaintext system. Adding subORAMs helps Snoopy scale by breaking data into partitions, but adding subORAMs is not free, as it increases the dummy overhead.

We prove that the following f for setting batch size B guarantees negligible overflow probability in §4.11:

Theorem 4.3. *For any set of R requests that are distinct and randomly distributed, number of subORAMs S, and security parameter λ, let $\mu = R/S$, $\gamma = -\log(1/(S \cdot 2^{\lambda}))$, and $W_0(\cdot)$ be branch 0 of the Lambert W function [67]. Then for the following function $f(R, S)$ that outputs a batch size, the probability that a request is dropped is negligible in λ:*

$$f(R, S) = \min(R,\ \mu \cdot \exp\left[W_0\left(e^{-1}\left(\gamma/\mu - 1\right)\right) + 1\right]) .$$

Proof intuition. For a single subORAM s, let $X_1, \ldots, X_R \in \{0, 1\}$ be independent random variables where X_i represents request i mapping to s. Then, $\Pr[X_i = 1] = 1/S$. Next, let the random variable $X = \sum_{i=1}^{R} X_i$ represent the total number of requests that hashed to s. We use a Chernoff bound to upper-bound the probability that there are more than k requests to a single subORAM, $\Pr[X \geq k]$. In order to upper-bound the probability of overflow for *all* subORAMs, we use the union bound and solve for the smallest k that results in an upper bound on the probability of overflow negligible in λ. In order to solve for k, we coerce the inequality into a form that can be solved with the Lambert W function, which is the inverse relation of $f(w) = we^w$, i.e., $W(we^w) = w$ [67]. When $f(R, S) = R$, the overflow probability is zero, and so we can safely upper-bound $f(R, S)$ by R. We target the high-throughput case where R is large, in which case our bound is less than R.

We now explain how Theorem 4.3 applies to Snoopy. For security, it is important that an attacker cannot (except with negligible probability) choose a set of requests that causes a batch to overflow. Thus Snoopy needs to ensure that requests chosen by the attacker are transformed to a set of requests that are distinct and randomly distributed across subORAMs. Snoopy ensures that requests are distinct through deduplication and that requests are randomly distributed by using a keyed hash function where the attacker does not know the key. Because the keyed hash function remains the same across epochs, Snoopy must prevent the attacker from learning which request is assigned to which subORAM during execution (otherwise, the attacker could use this information to construct requests that will overflow a batch). Snoopy does this by ensuring that each subORAM receives the same number of requests and by obliviously assigning requests to the correct subORAM batch (§4.4.2). Theorem 4.3 allows us to choose a batch size that is less than R in the high-throughput setting (for scalability) while ensuring that the probability that an attacker can construct a batch that causes overflow is cryptographically negligible. Thus Snoopy achieves security for *all workloads*, including skewed ones.

The bound we derive is valuable in applications beyond Snoopy where there are a large number of balls and it is important that the overflow probability is very small for different numbers of

balls and bins. Our bound is particularly useful in the case where the overflow probability must be negligible in the security parameter as opposed to an application parameter (e.g. the number of bins) [26, 218, 255, 256].

4.4.2 Oblivious batch coordination

As with other components of the system, the load balancer runs inside a hardware enclave, and so we must ensure that its memory accesses remain independent of request content. The load balancer runs two algorithms that must be oblivious: generating batches of requests (§4.4.2), and matching subORAM responses to client requests (§4.4.2).

Practically, designing oblivious algorithms requires ensuring that the memory addresses accessed do not depend on the data; often this means that the access pattern is fixed and depends only on public information (alternatively, access patterns might be randomized). The data contents remain encrypted and inaccessible to the attacker, and only the pattern in which memory is accessed is visible. We build our algorithms on top of an oblivious "compare-and-set" operator that allows us to copy a value if a condition is true without leaking if the copy happened or not.

Background: oblivious building blocks.

We first provide the necessary background for two oblivious building blocks from existing work that we will use in our algorithms.

Oblivious sorting. An oblivious sort orders an array of n objects without leaking information about the relative ordering of objects. We use bitonic sort, which runs in time $O(n \log^2 n)$ and is highly parallelizable [20]. Bitonic sort accesses the objects and performs compare-and-swaps in a *fixed, predefined order*. Since its access pattern is independent of the final order of the objects, bitonic sort is oblivious.

Oblivious compaction. Given an array of n objects, each of which is tagged with a bit $b \in \{0, 1\}$, oblivious compaction removes all objects with bit $b = 0$ without leaking information about which objects were kept or removed (except for the the total number of objects kept). We use Goodrich's algorithm, which runs in time $O(n \log n)$ and is *order-preserving*, meaning that the relative order of objects is preserved after compaction [118]. Goodrich's algorithm accesses array locations in a fixed order using a $\log n$-deep routing network that shifts each element a fixed number of steps in every layer.

Generating batches of requests.

Generating fixed-size batches *obliviously* requires care. It is not enough to simply pad batches with a variable number of dummy requests, as this can leak the number of real requests in each batch. Instead, we must pad each batch with the right number of dummy requests *without revealing the exact number of dummy requests added to each batch.* To solve this problem, we obliviously

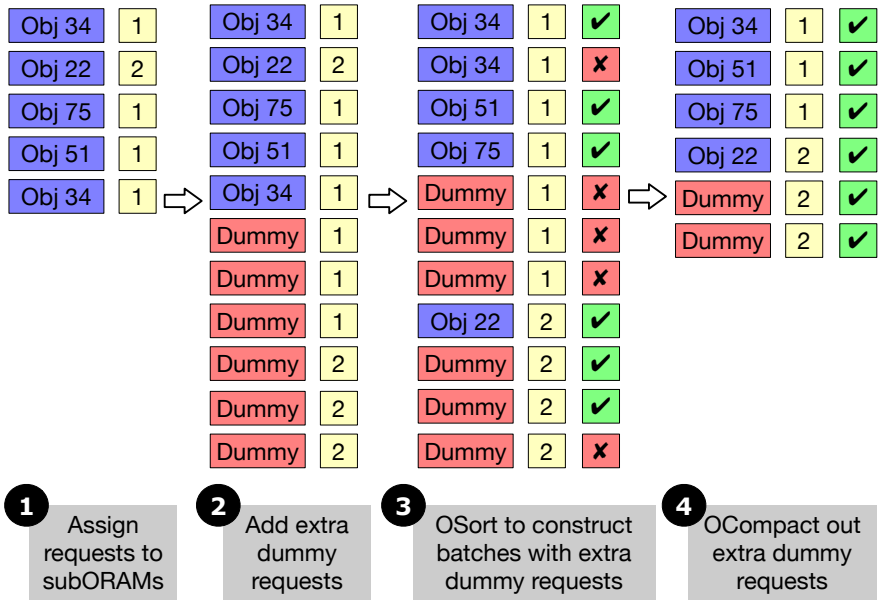


Figure 4.5: Generating batches of requests at the load balancer.

generate batches in three steps, which we show in Figure 4.5: ❶ we first assign client requests to subORAMs according to their requested object; ❷ we add the maximum number of dummy requests to each subORAM; ❸ we construct batches with those extra dummies; and ❹ we filter out unnecessary dummies.

First (❶), we scan through the list of client requests. For each client request, we compute the subORAM ID by hashing the object ID, and we store it with the client request. Second (❷), we append the maximum number of dummy requests for each subORAM, $B = f(R, S)$ to the end of the list. These dummy requests all have a tag bit $b = 1$. Third (❸), we group real and dummy requests into batches by subORAM. We do this by obliviously sorting the lists of requests, setting the comparison function to order first by subORAM (to group requests into subORAM batches), then by tag bit b (to push the dummies to the end of the batches), and then by object ID (to place duplicates next to each other). Finally (❹), to choose which requests to keep and which to remove, we iterate through the sorted request list again. We keep a counter x of the number of distinct requests seen so far for the current subORAM. We securely update the counter by performing an oblivious compare-and-set for each request, ensuring that access patterns don't reveal when the counter is updated. If $x < B$ and the request is not a duplicate (i.e. it is not preceded by a request for the same object), we set bit $b = 1$ (otherwise $b = 0$). To filter out unnecessary dummy requests and duplicates, we obliviously compact by bit b, leaving us with a B-sized batch for each subORAM.

The algorithm is oblivious because it only relies on linear scans and appends (both are data-independent) and our oblivious building blocks. The runtime is dominated by the cost of oblivious sorting and compaction.

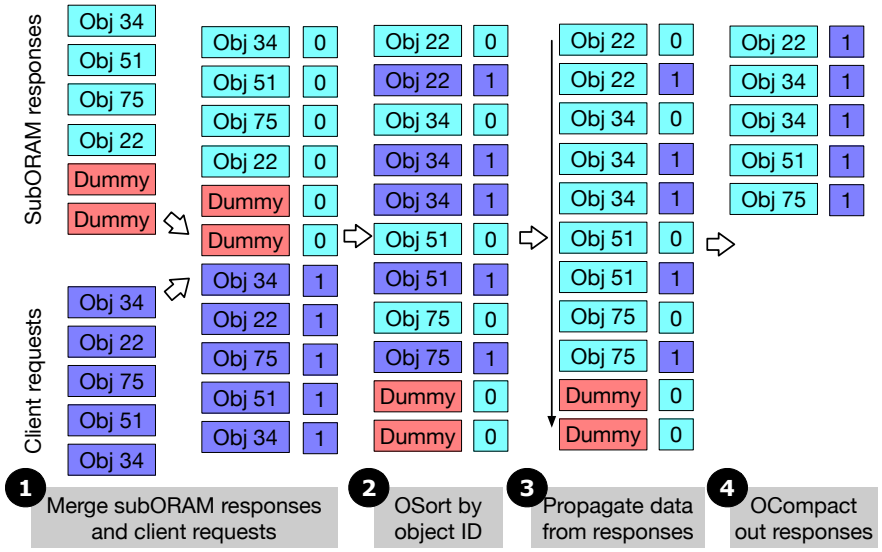


Figure 4.6: Mapping subORAM responses to client requests at the load balancer.

Mapping responses to client requests.

Once we receive the batches of responses from the subORAMs, we need to send replies to clients. This requires mapping the data from subORAM responses to the original requests, making sure that we propagate data correctly to duplicate responses and that we ignore responses to dummy requests. We accomplish this obliviously in four steps, which we show in Figure 4.6: ❶ we merge together the client requests and the subORAM responses and then sort the list; ❷ we sort the merged list to group requests with responses; ❸ we propagate data from the responses to the original requests; and ❹ we filter out the now unnecessary subORAM responses.

The load balancer takes as input two lists: a list of subORAM responses and a list of client requests. First (❶), we merge the two lists, tagging the subORAM responses with a bit $b = 0$ and the client requests with $b = 1$. Second (❷), we sort this combined list by object ID and then, to break ties, by the tag bit b. Breaking ties by the tag bit b arranges the data so that we can easily propagate data from subORAM responses to requests. Third (❸), we iterate through the list, propagating data in objects with the tag bit $b = 0$ (the subORAM responses) to the following object(s) with the tag bit $b = 1$ (the client requests). As we iterate through the list, we keep track of the last object we have seen with $b = 1$, prev (i.e. the last subORAM response we've scanned over). Then, for the current object curr, we copy the contents of prev into the curr if $b = 0$ for curr (it's a request). Any requests following a response must be for the same object because every request has a corresponding response and we sort by object ID. Note that dummy responses will not have a corresponding client request. Finally (❹), we need to filter down the list to include only the client requests. We do this using oblivious compaction, removing objects with the tag bit $b = 0$ (the subORAM responses). Note that, in order to respond to a request, we need to map a client request to the original network connection; we can do this by keeping a pointer to the

connection with the request data.

This procedure is oblivious because it relies only on oblivious building objects as well as concatenating two lists and a linear scan, both of which are data-independent. As in the algorithm for generating batches, the runtime is dominated by the cost of oblivious sorting and oblivious compaction.

4.4.3 Scaling the load balancer

Our load balancer design scales horizontally; it is both correct and secure to add load balancers without introducing additional coordination costs. Clients randomly choose one load balancer to contact, and then each load balancer batches requests independently. This is a significant departure from prior work where a centralized proxy receives all client requests and must maintain dynamic state relevant to all requests [29, 72, 269, 292]. SubORAMs execute load balancer batches in a fixed order, and within a single load balancer, we aggregate reads and writes using a "last-write-wins" policy.

Adding load balancers eliminates a potential bottleneck, but is not entirely free. Because (1) load balancers do not coordinate to deduplicate requests and (2) subORAMs assume that a batch contains distinct requests, subORAMs cannot combine batches from different load balancers. Our subORAM must scan over all stored objects to process a single batch (§4.5). As a result, if there are L load balancers, each subORAM must perform L scans over the data every epoch.

4.5 Throughput-optimized subORAM

Many ORAMs target asymptotic complexity, often at the expense of concrete cost. In contrast, recent work has explored how to leverage *linear scans* to build systems that can achieve better performance for expected workloads than their asymptotically more efficient counterparts [77, 87]. We take a similar approach to design a high-throughput subORAM optimized for hardware enclaves. We exploit the fact that, due to Snoopy's design, each subORAM stores a relatively small data partition and receives a batch of distinct requests. In this setting, using a *single linear scan* over the data partition to process a batch is concretely efficient in terms of amortized per-request cost.

We draw inspiration from Signal's private contact discovery protocol [204]. There, the client sends its contacts to an enclave, and the enclave must determine which contacts are Signal users without leaking the client's contacts. Their solution employs an *oblivious hash table*. The core idea is that the enclave performs some expensive computation to construct a hash table such that the construction access patterns don't leak the mapping of contacts to buckets. Once this hash table is constructed, the enclave can directly access the hash bucket for a contact without the memory access pattern revealing which contact was looked up. Note that obliviousness only holds if (1) the enclave performs a lookup for each contact at most once, and (2) the enclave scans the entire bucket (to avoid revealing the location of the contact accessed inside the bucket). With this tool, private contact discovery is straightforward: the enclave constructs an oblivious hash

table for the client's contacts and then scans over every Signal user, looking up each Signal user in the contact hash table.

Signal's setting is similar to ours: instead of a set of contacts, we have a batch of distinct requests, and instead of needing to find matches with the Signal users, we need to find the stored objects corresponding to requests. However, Signal's approach has some serious shortcomings when applied to our setting. First, their hash table construction takes $O(n^2)$ time for n contacts. While this complexity is acceptable when n is the size of a user's contacts list (relatively small), it is prohibitively expensive for batches with thousands of requests. Second, they do not size their buckets to prevent overflow. Overflows can leak information about bucket contents, and attempting to recover causes further leakage [49, 177].

Choosing an oblivious hash table. We need to identify an oblivious hash table that is efficient and secure in our setting. A natural first attempt to solve the overflow problem is to use the number of requests that hash to each bucket to set the bucket size dynamically. This simple solution is insecure: the attacker can infer the probability that an object was requested based on the size of the bucket that object hashes to.

Instead, we need to set the bucket size so that the overflow probability is cryptographically negligible. This provides the security property we want, and is exactly the problem that we solved in the load balancer, where we separated requests into "bins" such that the probability that any "bin" overflows is negligible. Using our load balancer approach also reduces construction cost from $O(n^2)$ to $O(n \operatorname{polylog} n)$. However, while this solution works well at the load balancer, it becomes expensive when applied to the subORAM. Recall that to perform an oblivious lookup, we must scan the entire bucket that might contain a request, and so we want buckets to be as small as possible. Unfortunately, decreasing the bucket size results in substantial dummy overhead. This overhead was the reason for making our batches as large as possible at the load balancer (Figure 4.3). In our subORAM, we want to keep the dummy overhead low *and* have a small bucket size.

To achieve both these properties, we identify *oblivious two-tier hash tables* as a particularly well-suited to our setting [49]. Chan et al. show how to size buckets such that overflow requests are placed into a second hash table, allowing us to have both low dummy overhead and a small bucket size: for batches of 4,096 requests, buckets in a two-tier hash table are $\sim$10$\times$ smaller than their single-tier counterparts. Construction now requires two oblivious sorts, one for each tier, but is still much faster than Signal's approach, both asymptotically and concretely for our expected batch sizes. We refer the reader to Chan et al. for the details of oblivious construction, oblivious lookups, and the security analysis [49].

Processing a batch of requests. We now describe how to leverage an oblivious two-tier hash table to obliviously process a batch of requests (Figure 4.7). First (❶), when the batch of requests arrives, we construct the oblivious two-tier hash table as described above. To avoid leaking the relationship between requests across batches, for every batch we sample a new key (unknown to the attacker) for the keyed hash function assigning objects to buckets. Second (❷), we iterate through the stored objects. For each object obj, we perform an oblivious hash table lookup. A

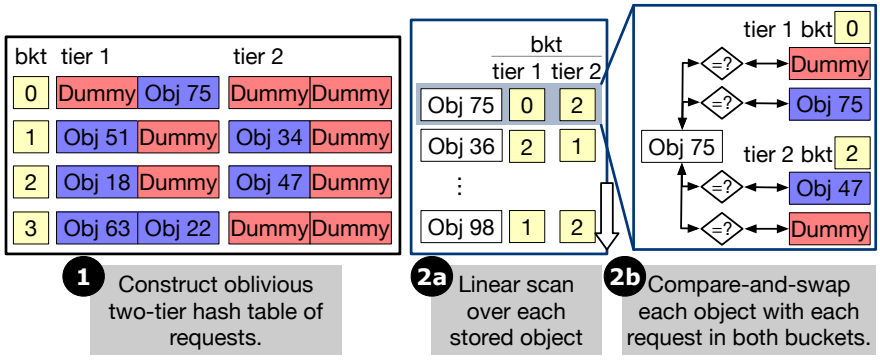


Figure 4.7: Processing a batch of requests at a subORAM.

lookup requires hashing obj.id in order to find the corresponding bucket in both hash tables and then scanning the entire bucket; this scan is necessary to hide the specific object being looked up. For every request req scanned, we perform an oblivious compare-and-set to update either the req in the hash table or the obj in subORAM storage depending on (1) whether req.id matches obj.id, and (2) whether req is a read or write. By conditioning the oblivious compare-and-set on the request type and performing it twice (once on the contents of req and once on the contents of obj), we hide whether the request is a read or a write.

Finally, we scan through every hash table bucket, marking real requests with tag bit $b = 1$ and dummies with $b = 0$. We then use oblivious compaction to filter out the dummies, leaving us with real entries to send back to the load balancer.

4.6 Planner

Our Snoopy planner takes as input a data size D, minimum throughput X_{Sys}, maximum latency L_{Sys}, and outputs a configuration (number of load balancers and subORAMs) that minimizes system cost. As the search space is large, we rely on heuristics and make simplifying assumptions to approximate the optimal configuration. We derive three equations capturing the relationship between our core system parameters: the epoch length T, number of objects N, number of subORAMs S, and number of load balancers B.

To estimate throughput for some epoch time T, we observe that, on average, we must be able to process all requests received during the epoch in time $\leq T$ (otherwise, the set of outstanding requests continues growing). We can pipeline the subORAM and load balancer processing such that the upper bound on the requests we can process per epoch is determined by either the load balancer or subORAM processing time, depending on which is slower. Adding load balancers decreases the work done at each load balancer, but each subORAM must process a batch of requests from every load balancer. Let $L_{\mathsf{LB}}(R, S)$ be the time it takes a load balancer to process R requests in a system with S subORAMs, and let $L_S(R, S, N)$ be the time it takes a subORAM to process a

batch of R requests with N stored objects. We then derive:

$$T \geq \max[L_{\mathsf{LB}}(X_{\mathsf{Sys}} \cdot T/B, S),\ B \cdot L_S(f(X_{\mathsf{Sys}} \cdot T/B, S), N)] \tag{4.1}$$

Requests will arrive at different times and have to wait until the end of the current epoch to be serviced, and so on average, if the timing of requests is uniformly distributed, requests will wait on average $T/2$ time to be serviced. The time to process a batch is upper-bounded by T at both the subORAM and the load balancer, and so:

$$L_{\mathsf{Sys}} \leq 5T/2 \tag{4.2}$$

Let C_{LB} be the cost of a load balancer and C_S be the cost of a subORAM. We then compute the system cost C_{Sys}:

$$C_{\mathsf{Sys}}(B, S) = B \cdot C_{LB} + S \cdot C_S \tag{4.3}$$

Our planner uses these equations and experimental data to approximate the cheapest configuration meeting performance requirements. While our planner is useful for selecting a configuration, it does not provide strong performance guarantees, as our model makes simplifying assumptions and ignores subtleties that could affect performance (e.g. our simple model assumes that requests are uniformly distributed). Our planner is meant to be a starting point for finding a configuration. Our design could be extended to provide different functionality; for example, given a throughput, data size, and cost, output a configuration minimizing latency.

4.7 Implementation

We implemented Snoopy in ∼7,000 lines of C++ using the OpenEnclave framework v0.13 [237] and Intel SGX v2.13. We use gRPC v1.35 for communication and OpenSSL for cryptographic operations. Our bitonic sort [20] and oblivious compaction [118] implementations set the size of oblivious memory to the register size. We use Intel's AVX-512 SIMD instructions for oblivious compare-and-swaps and compare-and-sets.

Reducing enclave paging overhead. The size of the protected enclave memory (EPC) is limited and enclave memory pages that do not fit must be paged in when accessed, which imposes high overheads [238]. The data at a subORAM often does not fit inside the EPC, so to reduce the latency to page in from untrusted memory, we rely on a shared buffer between the enclave and the host. A host loader thread fills the buffer with the next objects that the linear scan will read. This eliminates the need to exit and re-enter the enclave to fetch data, dramatically reducing linear scan time. The enclave encrypts objects (for confidentiality) and stores digests of the contents inside the enclave (for integrity). This approach has been explored in prior enclave systems [248, 250].

4.8 Evaluation

To quantify how Snoopy overcomes the scalability bottleneck in oblivious storage, we ask:

	Redis [261]	Obladi [72]	Oblix [217]	**Snoopy**
Oblivious	✗	✓	✓	✓
No trusted proxy	✓	✗	✓	✓
High throughput	✓	✓	✗	✓
Throughput scales with machines	✓	✗	✗	✓

Table 4.1: Comparison of baselines based on security guarantees (oblivious), setup (no trusted proxy), and performance properties (high throughput and throughput scales).

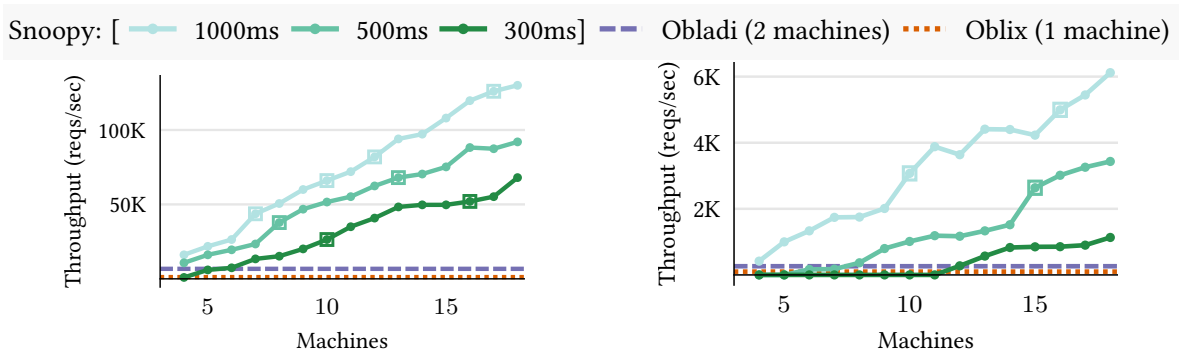
1. How does Snoopy's throughput scale with more compute, and how does it compare to existing systems? (§4.8.2)
2. How does adding compute resources help Snoopy reduce latency and scale to larger data sizes? (§4.8.3)
3. How do Snoopy's individual components perform? (§4.8.4)
4. Given performance and monetary constraints, what is the optimal way to allocate resources in Snoopy? (§4.8.5)

Experiment Setup. We run Snoopy on Microsoft Azure, which provides support for Intel SGX hardware enclaves in the DCsv2 series. For the load balancers and subORAMs, we use `DC4s_v2` instances with 4-core Intel Xeon E-2288G processors with Intel SGX support and 16GB of memory. For clients, we use `D16d_v4` instances with 16-core Intel Xeon Platinum 8272CL processors and 64GB of memory. We choose these instances for their comparatively high network bandwidth. We evaluate our baselines Redis [261] on `D4d_v4` instances, Obladi [72] on `D32d_v4` for the proxy and `D16d_v4` for the storage server, and Oblix on the same `DC4s_v2` instances as our subORAMs. For benchmarking, we use a uniform request distribution. This choice is only relevant for our Redis baseline; the oblivious security guarantees of Snoopy and other oblivious storage systems ensure that the request distribution does not impact their performance. Unless otherwise specified, we set the object size to 160 bytes (same as Oblix [217]).

4.8.1 Baselines

We compare Snoopy to three state-of-the-art baselines: Obladi [72] is a batched, high-throughput oblivious storage system, Oblix [217] efficiently leverages enclaves for oblivious storage, and Redis [261] is a widely used plaintext key-value store. Each baseline provides a different set of security guarantees and performance properties (Table 4.1).

Obladi. Obladi [72] uses batching and parallelizes RingORAM [263] to achieve high throughput. While Obladi also uses batching to improve throughput, its security model is different, as it uses a single trusted proxy rather than a hardware enclave. The trusted proxy model has two primary



(a) 2M objects, 160B block size.

(b) Key Transparency with 5M users (10M objects, 32B block size).

Figure 4.8: Snoopy achieves higher throughput with more machines. Boxed points denote when a load balancer is added instead of a subORAM. Oblix and Obladi cannot securely scale past 1 and 2 machines, respectively.

drawbacks: (1) the trusted proxy cannot be deployed in the untrusted cloud (desirable for convenience and scalability), and (2) the proxy is a central point of attack in the system (an attacker that compromises the proxy learns the queries of every user in the system). Practically, using a trusted proxy rather than a hardware enclave means the proxy does not have to use oblivious algorithms. Designing an oblivious algorithm for Obladi's proxy is not straightforward and would likely introduce significant overhead. Further, Obladi's trusted proxy is a compute bottleneck that cannot be horizontally scaled securely without new techniques, and so we only measure Obladi with two machines (proxy and storage server). We configure Obladi with a batch size of 500.

Oblix. Oblix [217] uses hardware enclaves and provides security guarantees comparable to ours. However, Oblix optimizes for latency rather than throughput; requests are sequential, and, unlike Obladi, Oblix does not employ batching or parallelism. Like Obladi, Oblix cannot securely scale across machines. We measure performance using Oblix's DORAM implementation and simulate the overhead of recursively storing the position map (as in §VI.A of [217]).

Redis. To measure the overhead of security (obliviousness), we compare Snoopy to an insecure baseline Redis [261], a popular unencrypted key-value store. In Redis, the server can directly see access patterns and data contents. We benchmark a Redis cluster using its own `memtier` benchmark tool [209], enabling client pipelining to trade latency for throughput. We expect it to achieve a much higher throughput than Snoopy.

4.8.2 Throughput scaling

Figure 4.8a shows that adding more machines to Snoopy improves throughput. We measure throughput where the average latency is less than 300ms, 500ms, and 1s. We start with 4 machines

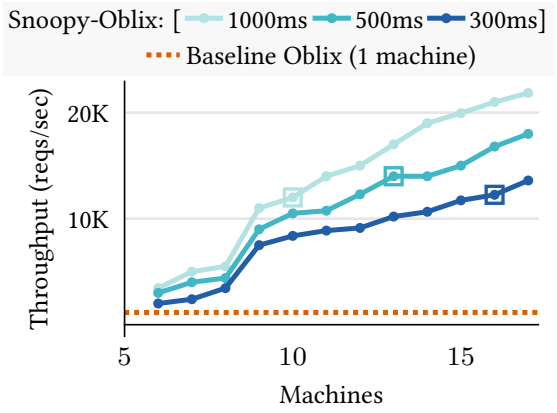


Figure 4.9: Throughput of Snoopy using Oblix [217] as a subORAM (2M objects, 160B block size). We measure throughput with different maximum average latencies.

(3 subORAMs and 1 load balancer) and scale to 18 machines (13 subORAMs and 5 load balancers for 1s latency; 15 subORAMs and 3 load balancers for 500ms/300ms latency). For 2M objects, Snoopy uses 18 machines to process 68K reqs/sec with 300ms latency, 92K reqs/sec with 500ms latency, and 130K reqs/sec with 1s latency. Each additional machine improves throughput by 8.6K reqs/sec on average for 1s latency. Relaxing the latency requirement improves throughput because we can group requests into larger batches, reducing the overhead of dummy requests.

We generate Figure 4.8a by measuring throughput with different system configurations and plotting the highest throughput configuration for each number of machines. We start with 4 machines rather than 2 because we need to partition the 2M objects to meet our 300ms latency requirement due to the subORAM linear scan (recall eq. (4.2) would require a subORAM to process a batch in $\leq$ 120ms). Both the load balancer and subORAM are memory-bound, as the EPC size is limited and enclave paging costs are high (§4.7).

Snoopy achieves higher throughput than Oblix (1,153 reqs/sec) and Obladi (6,716 reqs/sec) as we increase the number of machines. For 300ms, Snoopy outperforms Oblix with $\geq$5 machines and Obladi with $\geq$6 machines, and for 500ms and 1s, Snoopy outperforms Oblix and Obladi for all configurations. Oblix and Obladi beat Snoopy with a small number of machines for low latency requirements because our subORAM performs a linear scan over subORAM data whereas Oblix and Obladi only incur polylogarithmic access costs, allowing them to handle larger data sizes on a single machine. Snoopy can scale to larger data sizes by adding more machines (§4.8.3).

Comparison to Redis. To show the overhead of obliviousness, we also measure the throughput of Redis for 2M 160-byte objects with an increasing cluster size. For 15 machines, Redis achieves a throughput of 4.2M reqs/sec, 39.1$\times$ higher than Snoopy when configured with 1s latency. Because we pipeline Redis aggressively in order to maximize throughput, the mean Redis latency is $<$800ms.

Application: key transparency. Figure 4.8b shows throughput for parameter settings that

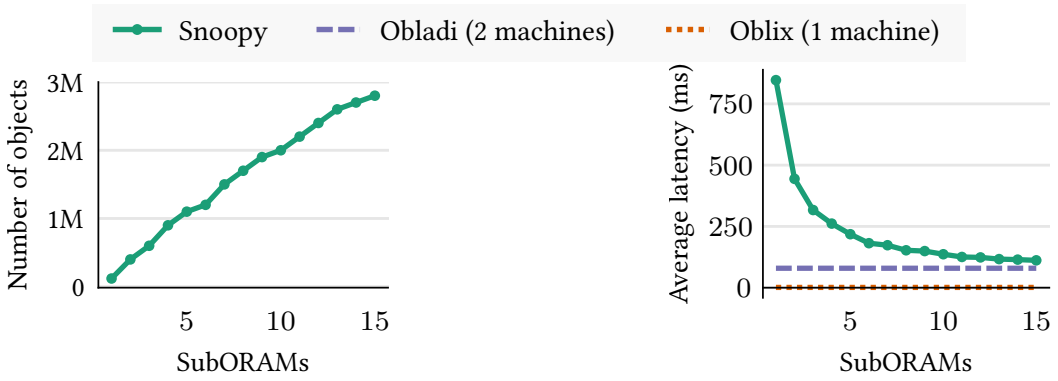
support key transparency (KT) [120, 208] for 5 million users. Due to the security guarantees of oblivious storage, an application's performance does not depend on its workload (i.e. request distribution), but only on the parameter settings. In KT, to look up Bob's key, Alice must retrieve (1) Bob's key, (2) the signed root of the transparency log, and (3) a proof that Bob's key is included in the transparency log (relative to the signed root) [208]. This inclusion proof is simply a Merkle proof. Thus, for n users, Alice must make $\log_2 n + 1$ ORAM accesses (Alice can request the signed root directly). Figure 4.8b shows that by adding machines, Snoopy scales to support high throughput for KT. At 18 machines (15 subORAMs and 3 load balancers), Snoopy can process 1.1K reqs/sec with 300ms latency, 3.2K reqs/sec with 500ms latency, and 6.1K reqs/sec with 1s latency. Note that the throughput in Figure 4.8b is much lower than Figure 4.8a because each KT operation requires 24 ORAM accesses.

Oblix as a subORAM. In Figure 4.9, we run Oblix [217] as a subORAM instead of Snoopy's throughput-optimized subORAM (§4.5). Snoopy's load balancer design enables us to securely scale Oblix beyond a single machine, achieving 15.6× higher throughput with Snoopy-Oblix for 17 machines with a max latency of 500ms (18K reqs/sec) than vanilla, single-machine Oblix (1.1K reqs/sec). The spike in throughput between 8 and 9 machines is due to sharding the data such that two instead of three layers of recursive lookups are required for every ORAM access. Snoopy-Oblix's performance also illustrates the value of our subORAM design; using our throughput-optimized subORAM (Figure 4.8a) improves throughput by $4.85\times$ with 17 machines and 500ms latency.

4.8.3 Scaling for latency and data size

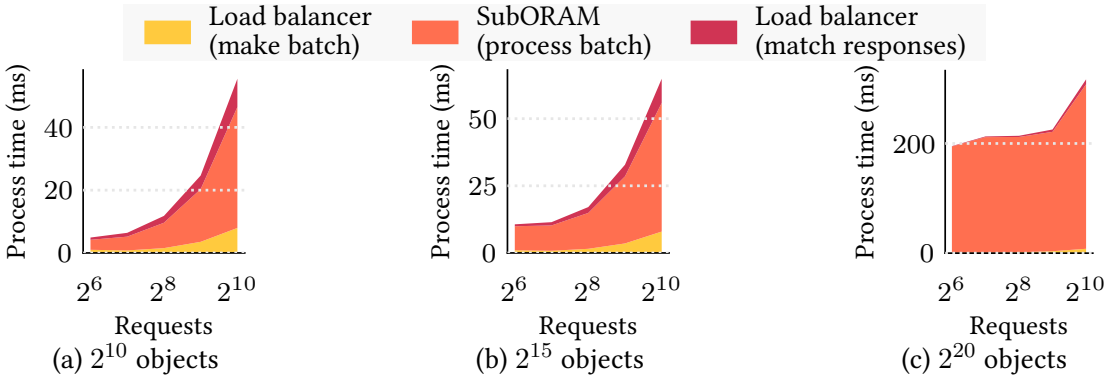
While Snoopy is designed specifically for throughput scaling (§4.8.2), adding machines to Snoopy can have other benefits if the load remains constant. We show how scaling can be used to both reduce latency and tolerate larger data sizes under constant load in Figure 4.10. Figure 4.10a illustrates how adding more subORAMs enables us to increase the number of objects Snoopy can store while keeping average response time under 160ms (the round-trip time from the US to Europe). The number of subORAMs required scales linearly with the data size because of the linear scan every epoch. Adding a subORAM allows us to store on average 191K more objects, and with 15 subORAMs, we can store 2.8M objects.

Figure 4.10b shows how adding subORAMs reduces latency when data size and load are fixed: for 2M objects, the mean latency is 847ms with 1 subORAM and 112ms with 15 subORAMs. Adding subORAMs parallelizes the linear scan across more machines, but has diminishing returns on latency because the dummy request overhead also increases when we add subORAMs (Figure 4.3). As expected, Oblix achieves a substantially lower latency (1.1ms) because it uses a tree-based ORAM and processes requests sequentially. Obladi achieves a latency of 79ms with batch size 500.



(a) SubORAMs vs. data size (b) SubORAMs vs. latency

Figure 4.10: (a) Adding more subORAMs allows for increasing the data size while keeping the average response time under 160ms (RTT from US to Europe). (b) Adding more subORAMs reduces latency. Snoopy is running 1 load balancer and storing 2M objects.



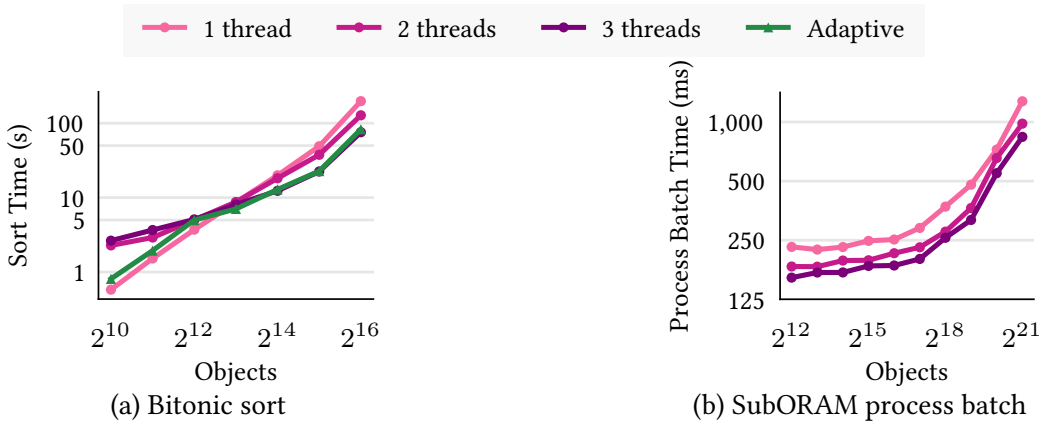
(a) 2^{10} objects (b) 2^{15} objects (c) 2^{20} objects

Figure 4.11: Breakdown of time to process one batch for different data sizes (one load balancer and one subORAM).

4.8.4 Microbenchmarks

Breakdown of batch processing time. Figure 4.11 illustrates how time is spent processing a batch of requests as batch size increases. As batch size increases, the load balancer time also increases, as the load balancer must obliviously generate batches. The subORAM time is largely dependent on the data size, as the processing time is dominated by the linear scan over the data. The subORAM batch processing time jumps between 2^{15} and 2^{20} objects due to the cost of enclave paging.

Sorting parallelism. In Figure 4.12a, we show how parallelizing bitonic sort across threads reduces latency, especially for larger data sizes. For smaller data sizes, the coordination overhead



(a) Bitonic sort

(b) SubORAM process batch

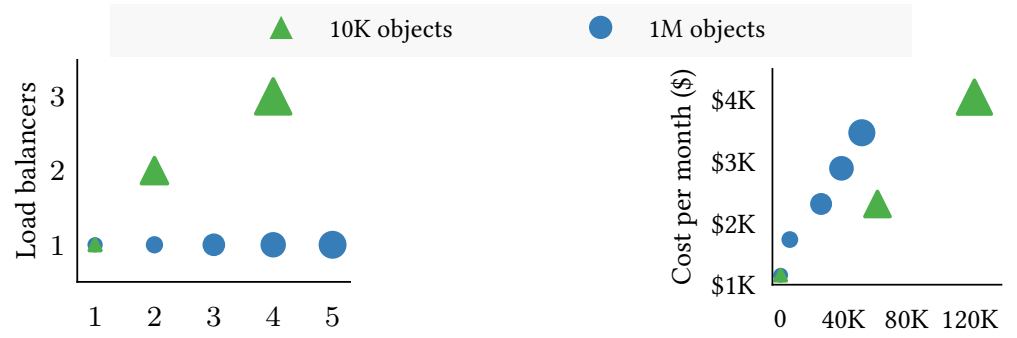
Figure 4.12: (a) Parallelizing bitonic sort across multiple threads. (b) Parallelizing batch processing at the subORAM across multiple enclave threads (batch size 4K requests).

actually makes it cheaper to use a single thread, and so we adaptively switch between a single-threaded and multi-threaded sort depending on data size. Parallelizing bitonic sort improves load balancer and subORAM performance.

SubORAM Parallelism. Similarly, in Figure 4.12b, we show how additional cores can be used to reduce subORAM batch processing time. We rely on a host thread to buffer in the encrypted data in the linear scan over the all objects in the subORAM (§4.7), and we can use the remaining cores to parallelize both the hash table construction and linear scan.

4.8.5 Planner

In Figure 4.13, we use our planner to find the optimal resource allocation for different performance requirements. Figure 4.13a shows the optimal number of subORAMs and load balancers to handle an increasing request load for different data sizes with 1s average latency. To support higher throughput levels, deployments with larger data sizes benefit from a higher ratio of subORAMs to load balancers, as partitioning across subORAMs parallelizes the linear scan over stored objects. In Figure 4.13b, we show how increasing throughput requirements affects system cost for different data sizes. Increasing data increases system cost: for ∼$4K/month, we can support 51.6K reqs/sec for 1M objects and 122.9K reqs/sec for 10K objects. To compute these configurations, the planner takes as input microbenchmarks for different batch sizes and data sizes. Because we cannot benchmark every possible batch and data size, we use the microbenchmarks for the closest parameter settings. Our planner's estimates could be sharpened further by running microbenchmarks at a finer granularity.



(a) Planner machine allocation (b) Planner cost

Figure 4.13: Optimal system configuration as throughput requirements increase for different data sizes (max latency 1s). Larger dot sizes represent higher throughput requirements. We show a subset of configurations from our planner in order to illustrate the overall trend of how adding machines best improves throughput.

4.9 Discussion

Fault tolerance and rollback protection.

Data loss in Snoopy can arise through node crashes and malicious rollback attacks. Many modern enclaves are susceptible to rollback attacks where, after shutdown, the attacker replaces the latest sealed data with an older version without the enclave detecting this change [241]. Prior work has explored how to defend against such attacks [37, 205]. Fault tolerance and rollback prevention are not the focus of this work, and so we only briefly describe how Snoopy could be extended to defend against data loss. All techniques are standard. Load balancers are stateless; we thus exclusively consider subORAMs. We propose to use a quorum replication scheme to replicate data to $f + r + 1$ nodes where f is the maximum number of nodes that can fail by crashing and r the maximum number of nodes that can be maliciously rolled back. Systems like ROTE [205] or SGX's monotonic counter provide a trusted counter abstraction that can be used to detect which of the received replies corresponds to the most recent epoch. The performance overhead of rollback protection would depend on the trusted counter mechanism employed, but Snoopy only invokes the trusted counter once per epoch.

Next-generation SGX enclaves. While current SGX enclaves can only support a maximum EPC size of 256MB, upcoming third-generation SGX enclaves can support EPC sizes up to 1TB [149]. This new enclave would not affect Snoopy's core design, but could improve performance by reducing the time for the per-epoch linear scan in the subORAM. With improved subORAM performance, Snoopy might need fewer subORAMs for the same amount of data, affecting the configurations produced by the planner (§4.8.5).

Private Information Retrieval (PIR). Snoopy's techniques can also be applied to the problem of private information retrieval (PIR) [61, 62]. A PIR protocol allows a client to retrieve an object from a storage server without the server learning the object retrieved. One fundamental limitation of PIR is that, if the object store is stored in its original form, the server must scan the entire object store for each request.

Snoopy's techniques can help overcome this limitation. We can replace the subORAMs with PIR servers, each of which stores a shard of the data. Our load balancer design then makes it possible to obliviously route requests to the PIR server holding the correct shard of the data. "Batch" PIR schemes that allow a client to fetch many objects at roughly the server-side cost of fetching a single object are well-suited tor our setting, as the load balancer is already aggregating batches of requests [136, 150]. Existing systems develop relevant batching [10, 129] and preprocessing [169] techniques.

4.10 Related work

We summarize relevant existing work, focusing on (1) oblivious algorithms designed for hardware enclaves, (2) ORAM parallelism, (3) distributing an ORAM across machines, and (4) balls-into-bins bounds for maximum load.

ORAMs with secure hardware. Existing research on oblivious computation using hardware enclave primarily targets latency. Oblix [217], ZeroTrace [270], Obliviate [3], Pyramid ORAM [70], and POSUP [140] do not support concurrency. Snoopy, in contrast, optimizes for throughput and leverages batching for security and scalability. ObliDB [93] supports SQL queries by integrating PathORAM with hardware enclaves, but uses an oblivious memory pool unavailable in Intel SGX. GhostRider [195] and Tiny ORAM [102] use FPGA prototypes designed specifically for ORAM. While no general-purpose, enclave-based ORAM supports request parallelism, MOSE [139] and Shroud [199] leverage data parallelism to improve the latency of a single request on large datasets. MOSE runs CircuitORAM [50] inside a hardware enclave and distributes the work for a single request across multiple cores. Shroud instead parallelizes Binary Tree ORAM across many secure co-processors by accessing different layers of the ORAM tree in parallel. Shroud uses data parallelism to optimize for latency and data size; throughput scaling is still limited because requests are processed sequentially.

Supporting ORAM parallelism. A rich line of work explores executing multiple client requests in parallel at a single ORAM server. Each requires some centralized component(s) that eventually bottlenecks scalability. PrivateFS [322] and ConcurORAM [47] coordinate concurrent requests to shared data using an encrypted query log on top of a hierarchical ORAM or a tree-based ORAM, respectively. This query log quickly becomes a serialization bottleneck. TaoStore [269] and Obladi [72] similarly rely on a trusted proxy to coordinate accesses to PathORAM and RingORAM, respectively. Taostore processes requests immediately, maintaining a local subtree to securely handle requests with overlapping paths. Obladi instead processes requests in

batches, amortizing the cost of reading/writing blocks over multiple requests. Batching also removes any potential timing side-channels; while TaoStore has to time client responses carefully, Obladi can respond to all client requests at once, just as in Snoopy.

PRO-ORAM [301], a read-only ORAM running inside an enclave, parallelizes the shuffling of batches of $\sqrt{N}$ requests across cores, offering competitive performance for read workloads. Snoopy, in contrast, supports both reads and writes.

A separate, more theoretical line of work considers the problem of Oblivious Parallel RAMs (OPRAMs), designed to capture parallelism in modern CPUs. Initiated by Boyle et al. [36], OPRAMs have been explored in subsequent work [48–50, 55] and expanded to other models of parallelism [258].

Scaling out ORAMs. Several ORAMs support distributing compute and/or storage across multiple servers. Oblivistore [292] distributes partitions of SSS-ORAM [293] across machines and leverages a load balancer to coordinate accesses to these partitions. This load balancer, however, does not scale and becomes a central point of serialization. CURIOUS [29] is similar, but uses a simpler design that supports different subORAMs (e.g. PathORAM). CURIOUS distributes storage but not compute; a single proxy maintains the mapping of blocks between subORAMs and runs the subORAM clients, which bottlenecks scalability. In contrast, Snoopy distributes both compute and storage and can scale in the number of subORAMs *and* load-balancers. Moreover, Snoopy remains secure when an attacker can see client response timing, unlike Oblivistore or CURIOUS [269].

Pancake [123] leverages a trusted proxy to transform a set of plaintext accesses to a uniformly distributed set of encrypted accesses that can be forwarded directly to an encrypted, non-oblivious storage server. While this approach achieves high throughput, the proxy remains a bottleneck as it must maintain dynamic state about the request distribution.

Balls-into-bins analysis. Prior work derives bounds for the maximum number of balls in a bin that hold with varying definitions of high probability, but are poorly suited to our setting because they are either inefficient to evaluate or do not have a cryptographically negligible overflow probability under realistic system parameters [26, 218, 255, 256]. Berenbrink et al. [26] assume a sufficiently large number of bins to derive an overflow probability n^{-c} for n bins and some constant c (Onodera and Shibuya [236] apply this bound in the ORAM setting). Raab and Steger [256] use the first and second moment method to derive a bound where overflow probability depends on bucket load. Ramakrishna's [259] bound can be numerically evaluated but is limited by the accuracy of floating-point arithmetic, and we were unable to compute bounds with a negligible overflow probability for $\lambda \geq 44$. Reviriego et al. [264] provide an alternate formulation that can be evaluated by a symbolic computation tool, but we were unable to efficiently evaluate it with SymPy.

4.11 Parameter analysis

Theorem 4.3. *For any set of R requests that are distinct and randomly distributed, number of subORAMs S, and security parameter λ, let $\mu = R/S$, $\gamma = -\log(1/(S \cdot 2^{\lambda}))$, and $W_0(\cdot)$ be branch 0 of the Lambert W function [67]. Then for the following function $f(R, S)$ that outputs a batch size, the probability that a request is dropped is negligible in λ:*

$$f(R, S) = \min(R,\ \mu \cdot \exp\left[W_0\left(e^{-1}\left(\gamma/\mu - 1\right)\right) + 1\right]) .$$

Proof. Let $X_1, X_2, \ldots, X_R$ be independent 0/1 random variables that represent request i hashing to a specific subORAM where $\Pr[X_i = 1] = 1/S$. Then, $X = \sum_{i=1}^{R} X_i$ is a random variable representing the total amount of requests hashing to a specific subORAM.

We can apply the Chernoff bound here. Let $\mu = \exists[X]$, which is $\sum_{i=1}^{R} 1/S = R/S$. Then,

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(\delta+1)^{\delta+1}}\right)^{\mu}$$

The variable X represents the total number of requests mapping to subORAM S, but we want to upper bound the number of requests received at *any* subORAM. We can define a bad event overflow that occurs when the number of requests received at any subORAM exceeds our upper bound. We can compute the probability of this bad event by taking a union bound over all S subORAMs:

$$\Pr[\mathsf{overflow}] \leq \sum_{j=1}^{S} \Pr[X \geq (1+\delta)\mu] = S \cdot \Pr[X \geq (1+\delta)\mu]$$

In order to ensure that we do not drop a request except with negligible probability, we want $\Pr[\mathsf{overflow}] \leq 1/2^{\lambda}$, which means we need to find some δ such that:

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(\delta+1)^{\delta+1}}\right)^{\mu} \leq \frac{1}{S \cdot 2^{\lambda}}$$

From this point, we can solve for δ to find the upper bound:

$$
\begin{aligned}
-\log\left(\left(\frac{e^{\delta}}{(\delta+1)^{\delta+1}}\right)^{\mu}\right) &\geq -\log\left(\frac{1}{S\cdot 2^{\lambda}}\right) = \gamma \\
-\mu(\log(e^{\delta}) - (\delta+1)\log(\delta+1)) &\geq \gamma \\
-\delta + (\delta+1)\log(\delta+1) &\geq \frac{\gamma}{\mu} \\
(-\delta-1) + (\delta+1)\log(\delta+1) &\geq \frac{\gamma}{\mu} - 1 \\
(\delta+1)(\log(\delta+1)-1) &\geq \frac{\gamma}{\mu} - 1 \\
e^{\log(\delta+1)}(\log(\delta+1)-1) &\geq \frac{\gamma}{\mu} - 1 \\
e^{\log(\delta+1)-1}(\log(\delta+1)-1) &\geq e^{-1}\left(\frac{\gamma}{\mu} - 1\right) \\
\log(\delta+1) - 1 &\geq W_0\left(e^{-1}\left(\frac{\gamma}{\mu} - 1\right)\right) \\
\delta &\geq e^{W_0\left(e^{-1}\left(\frac{\gamma}{\mu}-1\right)\right)+1} - 1
\end{aligned}
$$

where $W_0(\cdot)$ is branch 0 of the Lambert W function [67].

For small R, the above bound is greater than R. For $f(R,S) = R$, the overflow probability is zero, and so we can safely upper-bound f by R.

□

4.12 Security analysis

We adopt the standard security definition for ORAM [293, 294]. Intuitively, this security definition requires that the server learns nothing about the access pattern. In the enclave setting, this means that the enclave's memory access pattern shouldn't reveal any information about the requests or data. Because Snoopy uses multiple enclaves, the communication pattern between enclaves also shouldn't reveal any information. We refer to the information that the adversary learns (the memory access patterns and communication patterns) as the "trace". At a high level, we must prove security by showing that the adversary cannot distinguish between a real experiment, where enclaves are running the Snoopy protocol on real requests and data, and an ideal experiment, where enclaves are running a simulator program that only takes as input public information. We define these experiments in detail below.

4.12.1 Enclave definition

We model a directed acyclic graph (DAG) of enclaves as the ideal functionality $\mathcal{F}_{\mathsf{Enc}}$ with the following interface:

- $\mathsf{E}_\mathsf{P} \leftarrow \mathsf{Load}(\mathsf{P})$: The load function takes a program P and produces an enclave DAG E_P loaded with P (the program specifies the individual programs running on each enclave and the paths of communication). This is implemented using a remote attestation procedure in Intel SGX.
- $(\mathsf{out}, \gamma) \leftarrow \mathsf{Execute}(\mathsf{E}_\mathsf{P}, \mathsf{in})$: The execute function takes an enclave DAG loaded with P, feeds in to the enclave DAG and produces the resulting output out as well as a trace of memory accesses and communication patterns between enclaves γ. Execute supports programs that communicate across enclaves and access individual enclave memories and simply outputs the trace of executing such programs.

We treat the enclave DAG as a black box that realizes the above ideal functionalities. We assume that the server cannot roll back the enclaves during execution and that Execute provides privacy and integrity for the enclave's internal memory and communication between enclaves.

Our ideal functionality interface is loosely based on the interface in ZeroTrace [270]. However, ZeroTrace only considers a single enclave whereas we consider a DAG of enclaves (similar to Opaque [332]). Also, ZeroTrace outputs proofs of correctness, whereas we use an ideal functionality where the enclave always loads and executes correctly.

4.12.2 Our model

We only model the case where there is a single client controlled by the adversary. We informally discuss how to extend our security guarantees to the multi-user setting in §4.12.7.

Our ideal enclave DAG functionality hides the details of how enclaves securely communicate; using authenticated encryption and nonces to avoid replaying messages are standard techniques and discussed in other works [332]. We assume that the system configuration (the number of load balancers and subORAMs) is fixed. Also, our ideal functionality protects the contents of memory, and so we do not model the optimization (§4.7) where we place encrypted data in external memory in order to reduce enclave paging overhead. Finally, we do not allow the attacker to perform rollbacks attacks and we do not model fault tolerance (we do not model the system using the fault tolerance and rollback protection techniques discussed in §4.9).

4.12.3 Oblivious storage definitions

An oblivious storage scheme consists of two protocols $(\textsc{OStoreInitialize}, \textsc{OStoreBatchAccess})$, where
$\textsc{OStoreInitialize}$ initializes the memory, and $\textsc{OStoreBatchAccess}$ performs a batch of accesses. We describe the syntax for both protocols below, which we will load and execute on an enclave DAG:

- $\textsc{OStoreInitialize}(1^\lambda, \mathbf{O})$, takes as input a security parameter λ and an object store $\mathbf{O}$ and runs initialization.

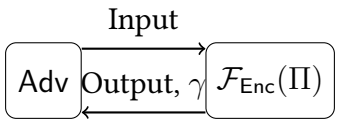


Figure 4.14: Real experiment for protocol Π running inside the enclave ideal functionality $\mathcal{F}_{\mathsf{Enc}}$ where γ is the trace.

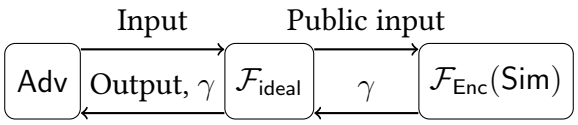


Figure 4.15: Ideal experiment where adversary interacts with the ideal functionality (computes the output for the given input) and the ideal functionality sends the public information to a simulator program running inside the enclave ideal functionality ($\mathcal{F}_{\mathsf{Enc}}$) to generate the trace γ.

- $\mathbf{V} \leftarrow$ OStoreBatchAccess($\mathbf{R}$), a protocol where the client's input is a batch $\mathbf{R}$ of requests of the form $(\mathtt{op},\ i,\ v_i)$ where $\mathtt{op}$ is the type of operation (read or write), i is an index, v_i is the value to be written (for $\mathtt{op} = \mathtt{read}$, $v_i = \perp$). The output consists of the updated secret state σ and the requested values $\mathbf{V}$ (i.e., $v_1, \ldots, v_\mu$) assigned to the $i_1, \ldots, i_\mu$ values of $\mathbf{O}$ if $\mathtt{op} = \mathtt{read}$ (for $\mathtt{op} = \mathtt{write}$, the returned value is the value before the write).

Security. The security of an oblivious storage scheme is defined using two experiments (real, ideal). In the real experiment (Figure 4.14), the adversary interacts with an enclave DAG loaded with the real protocol, and in the ideal experiment (Figure 4.15), the adversary interacts with an ideal functionality. The ideal functionality has the same interface as the real scheme but, rather than running the real protocol on the enclave DAG, it instead invokes a simulator (executing on the enclave DAG). Crucially, the simulator does not get access to the set of requests and only knows the public information, which includes the number of requests, structure of enclave DAG, and any other protocol-specific public parameters (e.g. number of load balancers and subORAMs). The adversary can execute OStoreInitialize and a polynomial number of OStoreBatchAccess for any set of requests, during which it observes the memory access patterns and communication patterns in the enclave DAG (represented by the trace produced by the Execute routine). The goal of the adversary is to distinguish between the real and ideal experiments.

An oblivious storage scheme is secure if no efficient polynomial-time adversary can distinguish between these two experiments with more than negligible probability. Our security definition has a different setup than that of traditional ORAM [293, 294] (we use a network of enclaves rather than the traditional client-server model), but our definition embodies the same security guarantees (namely, that the trace generated from an access is simulatable from public information).

We prove the security of Snoopy modularly: we first prove that our subORAM construction

$bit \leftarrow \mathbf{Real}^{\text{OSTORE}}_{\Pi,\mathsf{Adv}}(\lambda)$:

1: $(s,\ \mathbf{O}) \leftarrow \mathsf{Adv}(1^\lambda)$
2: $\mathsf{E_P} \leftarrow \mathcal{F}_{\mathsf{Enc}}.\mathsf{Load}(\Pi)$
3: $\gamma_0 \leftarrow \mathcal{F}_{\mathsf{Enc}}.\mathsf{Execute}(\mathsf{E_P},\ (\text{OSTOREINITIALIZE},\ 1^\lambda, \mathbf{O}))$.
4: **for** $k = 1$ to q **do** ▷ q: polynomial #queries
5: $\quad(\mathbf{R}_k,\ s) \leftarrow \mathsf{Adv}^1(\gamma_0, \ldots, \gamma_{k-1},\ \mathbf{V}_1, \ldots, \mathbf{V}_{k-1},\ s)$.
6: $\quad(\mathbf{V}_k,\ \gamma_k) \leftarrow \mathcal{F}_{\mathsf{Enc}}.\mathsf{Execute}(\mathsf{E_P},\ (\text{OSTOREBATCHACCESS},\ \mathbf{R}_k))$.
7: **end for**
8: **return** $bit \leftarrow \mathsf{Adv}(\gamma_0, \ldots, \gamma_k,\ \mathbf{V}_1, \ldots, \mathbf{V}_k,\ s)$.

$bit \leftarrow \mathbf{Ideal}^{\text{OSTORE}}_{\mathsf{Sim},\mathsf{Adv}}(\lambda)$:

1: $(s,\ \mathbf{O}) \leftarrow \mathsf{Adv}(1^\lambda)$.
2: $\mathsf{E_P} \leftarrow \mathcal{F}_{\mathsf{Enc}}.\mathsf{Load}(\mathsf{Sim})$
3: $\gamma_0 \leftarrow \text{IDEALOSTOREINITIALIZE}(\mathsf{E_P},\ 1^\lambda,\ \mathbf{O})$.
4: **for** $k = 1$ to q **do** ▷ q: polynomial #queries
5: $\quad(\mathbf{R}_k,\ s) \leftarrow \mathsf{Adv}^1(\gamma_0, \ldots, \gamma_{k-1},\ \mathbf{V}_1, \ldots, \mathbf{V}_{k-1},\ s)$.
6: $\quad(\mathbf{V}_k, \gamma_k) \leftarrow \text{IDEALOSTOREBATCHACCESS}(\mathsf{E_P},\ \mathbf{R}_k)$.
7: **end for**
8: **return** $bit \leftarrow \mathsf{Adv}(\gamma_0, \ldots, \gamma_k,\ \mathbf{V}_1, \ldots, \mathbf{V}_k,\ s)$.

[1] Security Definition 4.2 (weaker oblivious storage definition): Adv is not allowed to submit duplicate requests in batch R_k.

Figure 4.16: Real and ideal experiments for an oblivious storage scheme.

is secure, and then we prove that our Snoopy construction is secure when built on top of a secure subORAM. To do this, we need a slightly different notion of subORAMs. In particular, our SubORAM construction cannot be proven secure with Definition 4.1, since its security relies on the assumption that a batch of oblivious accesses contains *distinct* requests. In order to prove the security of our SubORAM we introduce a second, weaker security definition below.

Definition 4.2. (Weaker oblivious storage def.) The oblivious storage scheme Π is secure if for any non-uniform probabilistic polynomial-time (PPT) adversary Adv who *does not submit duplicated requests inside a batch* there exists a PPT Sim such that

$$\left|\Pr\left[\mathbf{Real}^{\text{OSTORE}}_{\Pi,\mathsf{Adv}}(\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}^{\text{OSTORE}}_{\mathsf{Sim},\mathsf{Adv}}(\lambda) = 1\right]\right| \leq \text{negl}(\lambda)$$

where λ is the security parameter, the above experiments are defined in Figure 4.16 (see note 1), and the randomness is taken over the random bits used by the algorithms of Π, Sim, and Adv.

$\gamma \leftarrow$ IdealOStoreInitialize($\mathsf{E_P}$, 1^λ, $\mathbf{O}$):
1: Initialize a key-value store S with contents from $\mathbf{O}$.
2: Run $\gamma \leftarrow \mathcal{F}_{\mathsf{Enc}}$.Execute($\mathsf{E_P}$, (SimOStoreInitialize, 1^λ, $|\mathbf{O}|$))
3: **return** γ.

$(\mathbf{V}, \gamma) \leftarrow$ IdealOStoreBatchAccess($\mathsf{E_P}$, $\mathbf{R}$):
1: Run the batch of requests $\mathbf{R}$ on the key-value store S to produced requested values $\mathbf{V}$.
2: Run $\gamma \leftarrow \mathcal{F}_{\mathsf{Enc}}$.Execute($\mathsf{E_P}$, (SimOStoreBatchAccess, $|\mathbf{R}|$))
3: **return** ($\mathbf{V}$, γ).

Figure 4.17: Ideal functionalities.

4.12.4 Oblivious building blocks

We use the following oblivious building blocks:

- OCmpSwap(b, x, y): If $b = 1$, swap x and y.
- OCmpSet(b, x, y): If $b = 1$, set $x \leftarrow y$.
- $L' \leftarrow$ OSort(L, f): Obliviously sorts the list L by some ordering function f, outputs sorted list L'.
- $L' \leftarrow$ OCompact(L, B): Obliviously compacts the list L, outputting element L_i only if $B_i = 1$. The order of the original list L is preserved.

Our algorithms require only a simple "oblivious swap" primitive to build oblivious compare-and-set, oblivious sort, and oblivious compact. In our implementation, we instantiate oblivious sort using bitonic sort [20] and oblivious compaction using Goodrich's algorithm [118]. We set the client's memory to be constant size in both. OCmpSwap and OCmpSet are standard oblivious building blocks, as described in Oblix [217]. Thus, we can assume the existence of simulators SimOCmpSwap, SimOCmpSet, SimOSort, and SimOCompact. While simulator algorithms usually run in their own "address space", because we need to produce memory traces that are indistinguishable from those produced by the original algorithm, we need to pass in the address of some objects, even if the algorithms do not need to know the *values* of these objects. We define the following simulator algorithms:

- SimOCmpSwap(addr$\langle x \rangle$, addr$\langle y \rangle$)): Simulates swapping x and y given a hidden input bit.
- SimOCmpSet(addr$\langle x \rangle$), addr$\langle y \rangle$)): Simulates setting x to y given a hidden input bit.
- SimOSort(addr$\langle L \rangle$, n, f): Simulates sorting list L of length n by ordering function f.
- SimOCompact(addr$\langle L \rangle$, n, addr$\langle B \rangle$, m): Simulates compacting list L of length n using bits in list B where the number of bits in B set to 1 is m.

We additionally use OHashTable [49], which is a two-tiered oblivious hash table that consists of the polynomial-time algorithms (Construct, GetBuckets):

- $T \leftarrow$ OHashTable.Construct(D): Given some data D, output a two-tiered oblivious hash table T.
- $(B_1,\ B_2) \leftarrow$ OHashTable.GetBuckets$(T,$ idx$)$: Given an oblivious hash table T and some index idx, output pointers to the two buckets corresponding to idx. Note that these buckets may be both read from and written to.

As these algorithms are oblivious [49], we can assume the existence of a simulator SimOHashTable with algorithms (Construct, GetBuckets):

- $T \leftarrow$ SimOHashTable.Construct$($addr$\langle D\rangle,\ n)$: Given the address of data D of size n, simulate constructing an oblivious hash table.
- $(B_1,\ B_2) \leftarrow$ SimOHashTable.GetBuckets$(T,$ addr$\langle$idx$\rangle)$: Given a hash table T, simulate outputting pointers to two buckets corresponding to the private input idx.

Finally, we assume we have access to a keyed cryptographic hash function H.

4.12.5 SubORAM

We define an oblivious storage scheme SubORAM in Figure 4.18 that provides the interface defined in §4.12.3 (we leave some empty lines in the protocol figure and corresponding simulator figure so that corresponding operations have the same line number).

Theorem 4.1. *Given a two-tiered oblivious hash table [49], an oblivious compare-and-set operator, and an oblivious compaction algorithm, the subORAM scheme described in §4.5 and formally defined in Figure 4.18 is secure according to Definition 4.2.*

Proof. We construct our simulator in Figure 4.19 (we leave some empty lines so that corresponding operations in Figure 4.18 have the same line number). We need to argue that the traces the adversary receives as a result of executing the Initialize and BatchAccess routines do not allow the adversary to distinguish between the real and ideal experiments. Communication patterns aren't a concern, as SubORAM only uses a DAG with a single enclave. Thus we only need to show that memory access patterns are indistinguishable. To simplify the proof and our description of the simulator, we assume that functions with different signatures are indistinguishable; the memory accesses of simulator functions that take fewer parameters (because they only take public input) can easily be made indistinguishable from those of the actual functions by passing in dummy arguments. We show how memory accesses are indistinguishable, first for Initialize and then for BatchAccess (line numbers correspond to Figure 4.18 and Figure 4.19).

Initialization.

- (Line 1) The subORAM algorithm takes as input an array of size n, whereas the simulator algorithm generates a random array of the same size with the same size objects. The resulting arrays are indistinguishable.
- (Line 2) These steps are the same and only involve storing the arrays that we already established are indistinguishable.

SubORAM.Initialize($1^\lambda, \mathbf{O}$)

1: Parse $\mathbf{O}$ as $(o_1, \ldots, o_n)$ where $o_i = (\mathsf{idx},\ \mathsf{content})$.
2: Store $\mathbf{O}$.

$\mathbf{V} \leftarrow$ SubORAM.BatchAccess($\mathbf{R}$)

1: Parse $\mathbf{R}$ as $(r_1, \ldots, r_N)$, where $r_i = (\mathsf{type},\ \mathsf{idx},\ \mathsf{content})$.
2: **if** $\mathbf{R}$ contains duplicates **then**
3: **return** $\perp$.
4: **end if**
5: Set $T \leftarrow$ OHashTable.Construct($\mathbf{R}$).
6: **for** $i = 1, \ldots, n$ **do**
7: Set $\mathbf{Bkt}_1, \mathbf{Bkt}_2 \leftarrow$ OHashTable.GetBuckets($T,\ \mathbf{O}[i].\mathsf{idx}$).
8: **for** $j = 1, 2$ **do**
9: **for** $l = 1, \ldots, |\mathbf{Bkt}_j|$ **do**
10: OCmpSet($(\mathbf{Bkt}_j[l].\mathsf{idx} \stackrel{?}{=} \mathbf{O}[i].\mathsf{idx}),\ \mathbf{O}[i].\mathsf{content},\ \mathbf{Bkt}_j[l].\mathsf{content}$).
11: OCmpSet($(\mathbf{Bkt}_j[l].\mathsf{idx} \stackrel{?}{=} \mathbf{O}[i].\mathsf{idx}) \wedge (\mathbf{Bkt}_j[l].\mathsf{type} \stackrel{?}{=} \mathsf{write}),\ \mathbf{Bkt}_j[l].\mathsf{content},\ \mathbf{O}[i].\mathsf{content}$).
12: **end for**
13: **end for**
14: **end for**
15: Scan through T, marking each entry i with bit $b_i = 0$ if it is a dummy, setting $b_i = 1$ otherwise.
16: Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{|T|})$.
17: Run $\mathbf{V} \leftarrow$ OCompact($T,\ \mathbf{B}$).
18: **return** $\mathbf{V}$.

Figure 4.18: Our subORAM construction.

Batch access.

- (Lines 1-4) The original algorithm doesn't perform any processing while the simulator algorithm generates an array of the same size and same object size as the array passed as input to the original algorithm. Even though the objects are randomly chosen in the simulator algorithm, because the sizes of the same, both have the same memory usage.
- (Line 5) From the security of the two-tier oblivious hash table, the hash table construction algorithm and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Lines 6, 8, 9) Both use the same looping structure that depends only on public data (i.e. the number of objects and the bucket size).
- (Line 7) By the security of the oblivious hash table, the get buckets algorithm and the corre-

SimSubORAM.Initialize($1^\lambda, |\mathbf{O}|$)

1: Let $(n, \kappa) = |\mathbf{O}|$ (κ is the object size). Create an array $\mathbf{O} = o_1, \ldots, o_n$ of random entries of size κ, where $o_i = (\mathsf{idx},\ \mathsf{content})$.
2: Store $\mathbf{O}$.

SimSubORAM.BatchAccess(N)

1: Let N be a public parameter, which denotes the number of requests that the input batch contains.
2: Choose N random distinct identifiers $\mathsf{idx}_1, \ldots, \mathsf{idx}_N$ where for all $i \in [N]$, idx_i is an idx value in $\mathbf{O}$.
3: Create $\mathbf{R}$ of the form $(r_1, \ldots, r_N)$, where $r_i = (\mathsf{read},\ \mathsf{idx}_i,\ \perp)$.
4:
5: Run $T \leftarrow \mathsf{SimOHashTable.Construct}(\mathsf{addr}\langle\mathbf{R}\rangle, N)$.
6: **for** $i = 1, \ldots, n$ **do**
7: Set $\mathbf{Bkt}_1, \mathbf{Bkt}_2 \leftarrow \mathsf{SimOHashTable.GetBuckets}(T, \mathsf{addr}\langle\mathbf{O}[i].\mathsf{idx}\rangle)$.
8: **for** $j = 1, 2$ **do**
9: **for** $l = 1, \ldots, |\mathbf{Bkt}_j|$ **do**
10: $\mathsf{SimOCmpSet}(\mathsf{addr}\langle\mathbf{O}[i].\mathsf{content}\rangle,\ \mathsf{addr}\langle\mathbf{Bkt}_j[l].\mathsf{content}\rangle)$.
11: $\mathsf{SimOCmpSet}(\mathsf{addr}\langle\mathbf{Bkt}_j[l].\mathsf{content}\rangle,\ \mathsf{addr}\langle\mathbf{O}[i].\mathsf{content}\rangle)$.
12: **end for**
13: **end for**
14: **end for**
15: Scan through T, marking each entry with bit $b_i = 0$.
16: Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{|T|})$.
17: Run $\mathbf{V} \leftarrow \mathsf{SimOCompact}(\mathsf{addr}\langle T\rangle,\ |T|,\ \mathsf{addr}\langle\mathbf{B}\rangle,\ N)$.
18:

Figure 4.19: Simulator algorithms SimSubORAM = (Initialize, BatchAccess).

sponding simulator algorithm produce indistinguishable memory access patterns.

- (Lines 10, 11) By the security of the oblivious compare-and-swap, the original algorithm and the simulator algorithm produce indistinguishable memory access patterns.
- (Line 15) Both algorithms perform linear scans over an array with a public size and add an extra bit to each array entry.
- (Line 16) These lines are identical and make a new array where the size is public (same size and object size as an existing array).
- (Line 17) By the security of oblivious compaction, the original compaction algorithm and the simulator algorithm produce indistinguishable memory access patterns.

The only task that remains is to show that the responses returned in the real and ideal ex-

periments are indistinguishable. The correctness of the results follows from Theorem 4.5, where we prove that our subORAM responds to read requests to an object by returning the last write to that object. □

4.12.6 Snoopy

We now define Snoopy as a protocol for L load balancers and S subORAMs $\mathcal{S}_1, \ldots, \mathcal{S}_S$ in Figure 4.20, as well as a load balancer scheme in Figure 4.22 and Figure 4.24 (we leave some empty lines in the protocol figures and corresponding simulator figures so that corresponding operations have the same line number).

Theorem 4.2. *Given a keyed cryptographic hash function, an oblivious compare-and-set operator, an oblivious sorting algorithm, an oblivious compaction algorithm, and an oblivious storage scheme (secure according to Definition 4.2), Snoopy, as described in §4.4 and formally defined in Figure 4.20, is secure according to Definition 4.1.*

Proof. Our Snoopy construction is presented in Figure 4.20, with the corresponding simulator in Figure 4.21. We again need to show that the traces that the adversary receives as a result of executing Initialize and BatchAccess do not allow the adversary to distinguish between the real and ideal experiments.

The communication patterns in the real and ideal experiments are indistinguishable. Both experiments perform setup at the first load balancer and then copy state to the remaining load balancer (communication pattern is deterministic). For BatchAccess, in both experiments, we choose a random load balancer, which then communicates with every subORAM (the amount of data sent to each subORAM depends only on public information). Thus there is no difference in the distribution of communication patterns between the real and ideal experiments.

We now discuss memory access patterns. As in the proof for Theorem 4.2, to simplify the proof and our description of the simulator, we assume that functions with different signatures are indistinguishable; the memory accesses of simulator functions that take fewer parameters (because they only take public input) can easily be made indistinguishable from those of the actual functions by passing in dummy arguments. As is clear from Figure 4.20 and Figure 4.21, the Initialize and BatchAccess algorithms are identical except that (1) the simulator algorithm generates random objects and random requests rather than taking them as input, and (2) the simulator algorithm calls the SimLoadBalancer algorithms. Thus the only task that remains is to show that the memory access patterns generated by the LoadBalancer and SimLoadBalancer algorithms are indistinguishable.

We start with Initialize and then examine BatchAccess (line numbers correspond to Figure 4.22, Figure 4.23, Figure 4.24, Figure 4.25).

Initialization.

- (Lines 1-2) The load balancer algorithm takes an array $\mathbf{O}$ whereas the simulator algorithm generates a random array of the same size (same number of objects and same object size). Thus the memory used by these arrays is indistinguishable.

$\mathsf{Snoopy.Initialize}_{L,S}(1^\lambda,\ \mathbf{O})$

1: Let L be a public parameter, which denotes the number of load balancers.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: $k \leftarrow \mathsf{LoadBalancer.Initialize}_S(1^\lambda,\ \mathbf{O})$.
4: Send k to the remaining $L-1$ load balancers.

$\mathbf{V} \leftarrow \mathsf{Snoopy.BatchAccess}_{L,S}(\mathbf{R})$

1: Let L be a public parameter, which denotes the number of load balancers.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: Wait to receive $|\mathbf{R}|$ requests.
4: Pick at random a load balancer i.
5: Run $\mathbf{V}_i \leftarrow \mathsf{LoadBalancer}_i.\mathsf{BatchAccess}_S(\mathbf{R})$.
6: **return** $\mathbf{V}_i$.

Figure 4.20: Our Snoopy construction.

$\mathsf{SimSnoopy.Initialize}_{L,S}(1^\lambda,\ |\mathbf{O}|)$

1: Let L be a public parameter, which denotes the number of load balancers.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: $k \leftarrow \mathsf{SimLoadBalancer.Initialize}_S(1^\lambda,\ |\mathbf{O}|)$.
4: Send k to the remaining $L-1$ load balancers.

$\mathsf{SimSnoopy.BatchAccess}_{L,S}(N)$

1: Let L be a public parameter, which denotes the number of load balancers.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: Let N be the number of requests.
4: Pick at random a load balancer i.
5: Run $\mathsf{SimLoadBalancer}_i.\mathsf{BatchAccess}_S(N)$.
6:

Figure 4.21: Simulator algorithms SimSnoopy = (Initialize, BatchAccess).

$k \leftarrow \mathsf{LoadBalancer.Initialize}_S(1^\lambda,\ \mathbf{O})$

1: Parse $\mathbf{O}$ as $o_1, \ldots, o_n$.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: Let H be a keyed cryptographic hash function that outputs an element in $[S]$.
4: Sample a secret key $k \xleftarrow{R} \{0,1\}^\lambda$.
5: **for** $i = 1, \ldots, n$ **do**
6: Attach to o_i the tag $t = H_k(o_i.\mathsf{idx})$.
7: **end for**
8: Let f_{order} be the ordering function that orders by tag t.
9: $\mathbf{O} \leftarrow \mathsf{OSort}(\mathbf{O},\ f_{\mathsf{order}})$.
10: Let $x \leftarrow 0$.
11: Let prev $\leftarrow \perp$.
12: **for** $i = 1, \ldots, |\mathbf{O}|$ **do**
13: **if** $\mathbf{O}[i].t \neq$ prev **then**
14: Let $y_x \leftarrow i$.
15: Let $x \leftarrow x + 1$.
16: Let prev $\leftarrow \mathbf{O}[i].t$.
17: **end if**
18: **end for**
19: **for** $i = 1, \ldots, S$ **do**
20: Run $\mathsf{SubORAM.Initialize}(1^\lambda,\ \mathbf{O}[y_{i-1} : y_i])$.
21: **end for**
22: Store k.
23: **return** k.

Figure 4.22: Our load balancer initialization construction. Lines 13-16 would in practice be implemented using OCmpSet, but we write it using an if statement that depends on private data to improve readability.

$k \leftarrow$ SimLoadBalancer.Initialize(1^λ, $|\mathbf{O}|$)

1: Let $(n, \kappa) = |\mathbf{O}|$. ▷ κ is the size of the object
2: Create an array $\mathbf{O}$ $(1, o_1), (2, o_2), \ldots, (n, o_n)$ of the form (idx, content), where o_i is a random entry of size κ.
3: Let H be a keyed cryptographic hash function that outputs an element in $[S]$.
4: Sample a secret key $k \xleftarrow{R} \{0,1\}^\lambda$.
5: **for** $i = 1, \ldots, n$ **do**
6: Attach to o_i the tag $t = H_k(o_i.\text{idx})$.
7: **end for**
8: Let f_{order} be the ordering function that orders by tag t.
9: OSort($\mathbf{O}$, f_{order}).
10: Let $x \leftarrow 0$.
11: Let prev $\leftarrow \perp$.
12: **for** $i = 1, \ldots, |\mathbf{O}|$ **do**
13: **if** $\mathbf{M}[i].t \neq$ prev **then**
14: Let $y_x \leftarrow i$.
15: Let $x \leftarrow x + 1$.
16: Let prev $\leftarrow \mathbf{O}[i].t$.
17: **end if**
18: **end for**
19: **for** $i = 1, \ldots, S$ **do**
20: Run SimSubORAM$_i$.Initialize(1^λ, $|\mathbf{O}[y_{i-1} : y_i]|$).
21: **end for**
22: Store k.
23: **return** k.

Figure 4.23: Load balancer simulator for SimLoadBalancer.Initialize. Lines 13-16 would in practice be implemented using OCmpSet, but we write it using an if statement that depends on private data to improve readability.

$\mathbf{V} \leftarrow \mathsf{LoadBalancer.BatchAccess}_S(\mathbf{R})$

Let S be a public parameter, which denotes the number of used SubORAMs.
Let $\mathsf{H}_k(\cdot)$ be a cryptographic hash function keyed by stored key k that outputs an element in $[S]$.
Parse $\mathbf{R}$ as $(r_1, \ldots, r_N)$, where $r_i = (\mathsf{type},\ \mathsf{idx},\ \mathsf{content})$.

Compute $\alpha \leftarrow f(N, S)$ and initialize the empty list $\mathbf{L}$ of size $N + \alpha S$.
for $i = 1, \ldots, N$ **do**
    $\mathbf{L}[i] = (r_i.\mathsf{type},\ r_i.\mathsf{idx},\ r_i.\mathsf{content},\ \mathsf{H}_k(r_i.\mathsf{idx}))$.
end for
$\mathbf{L}' \leftarrow$ Create a copy of $\mathbf{L}$.
Append to $\mathbf{L}'$ α dummy requests for each SubORAM of the form $(\mathsf{read},\ \mathsf{idx},\ \perp,\ s)$, where idx is $\mathsf{H}_k(\mathsf{idx}) = s$.
Let f_{order} be the ordering function that orders by SubORAM and then by type (where $\perp$ is last and treated as read).
Run $\mathbf{L}' \leftarrow \mathsf{OSort}(\mathbf{L},\ f_{\mathsf{order}})$.
Tag the first α distinct requests per SubORAM with $b = 1$ and the remaining requests with $b = 0$.
Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{N+\alpha S})$ and run $\mathbf{L}' \leftarrow \mathsf{OCompact}(\mathbf{L}',\ \mathbf{B})$.
for $i = 1, \ldots, S$ **do**
    Run $\mathbf{V}_i \leftarrow \mathsf{SubORAM}_i.\mathsf{BatchAccess}(\mathbf{L}'[(i-1)\alpha + 1 : i\alpha])$.
end for
Set $\mathbf{X} \leftarrow (\mathbf{V}_1, \ldots, \mathbf{V}_S, \mathbf{L})$ tagging all responses with $b = 0$ and requests with $b = 1$.
Let f_{order} be the ordering function that orders by idx and then by b (i.e., giving priority to responses over requests).
Set $\mathbf{X}' \leftarrow \mathsf{OSort}(\mathbf{X},\ f_{\mathsf{order}})$.
Set prev $\leftarrow \perp$.
for $i = 1, \ldots, |X'|$ **do**
    $\mathsf{OCmpSet}(b_i \stackrel{?}{=} 0,\ \mathsf{prev},\ \mathbf{X}'[i].\mathsf{content})$ and $\mathsf{OCmpSet}(b_i \stackrel{?}{=} 1,\ \mathbf{X}'[i].\mathsf{content},\ \mathsf{prev})$.
end for
Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{N+\alpha S})$.
Run $\mathbf{V} \leftarrow \mathsf{OCompact}(\mathbf{X}',\ \mathbf{B})$.
return $\mathbf{V}$.

Figure 4.24: Our load balancer construction.

SimLoadBalancer.BatchAccess(N)

Let N be a public parameter, which denotes the number of requests that the queried batch contains. Let S be a public parameter, which denotes the number of used SubORAMs.
Let $\mathsf{H}_k(\cdot)$ be a cryptographic hash function keyed by stored key k that outputs an element in $[S]$.
Choose N random identifiers $\mathsf{idx}_1, \ldots, \mathsf{idx}_N$ where for all $i \in [N]$, idx_i is an idx value in $\mathbf{O}$.
Create $\mathbf{R}$ of the form $(r_1, \ldots, r_N)$, where $r_i = (\mathsf{read}, \mathsf{idx}_i, \perp)$.
Compute $\alpha \leftarrow f(N, S, \lambda)$ and initialize the empty list $\mathbf{L}$ of size $N + \alpha S$.
for $i = 1, \ldots, N$ **do**
    $\mathbf{L}[i] = (r_i.\mathsf{type}, r_i.\mathsf{idx}, r_i.\mathsf{content}, \mathsf{H}_k(r_i.\mathsf{idx}))$.
end for
$\mathbf{L}' \leftarrow$ Create a copy of $\mathbf{L}$.
Append to $\mathbf{L}'$ α dummy requests for each SubORAM of the form $(\mathsf{read}, \mathsf{idx}, \perp, s)$, where idx is $\mathsf{H}_k(\mathsf{idx}) = s$.
Let f_{order} be the ordering function that orders by SubORAM and then by type (where $\perp$ is last and treated as read).
Run SimOSort($\mathsf{addr}\langle\mathbf{L}'\rangle, |\mathbf{L}'|, f_{\mathsf{order}}$).
Tag the first α requests per SubORAM with $b = 1$ and the remaining requests with $b = 0$.
Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{N+\alpha S})$ and run SimOCompact($\mathsf{addr}\langle\mathbf{L}', \rangle, N + \alpha S, \mathsf{addr}\langle\mathbf{B}\rangle, \alpha S$).
for $i = 1, \ldots, S$ **do**
    Run $\mathbf{V}_i \leftarrow \mathsf{SimSubORAM}_i.\mathsf{BatchAccess}(\alpha)$.
end for
Let $\mathbf{X}$ be an array of $N + \alpha S$ objects the same size as the objects in $\mathbf{L}$ with a tag bit.
Let f_{order} be the ordering function that orders by idx and then by b (i.e., giving priority to responses over requests).
Run SimOSort($\mathsf{addr}\langle\mathbf{X}\rangle, |\mathbf{X}|, f_{\mathsf{order}}$).
Set prev $\leftarrow \perp$.
for $i = 1, \ldots, |\mathbf{X}'|$ **do**
    SimOCmpSet($\mathsf{addr}\langle\mathsf{prev}\rangle, \mathsf{addr}\langle\mathbf{X}'[i].\mathsf{content}\rangle$) and SimOCmpSet($\mathsf{addr}\langle\mathbf{X}'[i].\mathsf{content}\rangle, \mathsf{addr}\langle\mathsf{prev}\rangle$).
end for
Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{N+\alpha S})$.
Run SimOCompact($\mathsf{addr}\langle\mathbf{X}'\rangle, |\mathbf{X}'|, \mathsf{addr}\langle\mathbf{B}\rangle, N$).

Figure 4.25: Load balancer simulator for SimLoadBalancer.BatchAccess.

- (Lines 3-8) These lines are identical. We sample a key and then perform a linear scan over an array where the size of the array and object size is public, attaching a tag to each element.
- (Line 9) By the security of our oblivious sort, the sorting algorithms over different arrays with the same length, same object size, and same ordering function produce indistinguishable memory access patterns because of the existence of the simulator function that only takes in array length, object size, and the ordering function.
- (Lines 10-17) These lines are identical. We iterate over the array where the array size is public. We write the algorithm as branching based on a comparison to private data in order to improve readability, but this would in practice be implemented using OCmpSet in the original algorithm and SimOCmpSet in the simulator algorithm, which produce indistinguishable access patterns.
- (Lines 19-21) By the security of the underlying subORAM scheme, the initialize procedure for the subORAM and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Lines 22-23) These lines are identical and only store a cryptographic key.

Batch access.

- (Lines 1-2) Establishing parameters and hash functions.
- (Lines 3-4) The load balancer receives a list of requests whereas the simulator algorithm generates a random array of the same size (same number of requests and same format). Thus the memory used by these arrays is indistinguishable.
- (Lines 5-11) These lines are identical and only compute a function based on public information and perform a linear scan over an array (same size and format in both). Thus the memory access patterns are indistinguishable.
- (Line 12) By the security of the oblivious sorting algorithm, the oblivious sort and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Line 13) These lines are identical and require accessing α objects in a fixed location where α is computed using public information.
- (Line 14) By the security of the oblivious compaction algorithm, the oblivious compaction and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Lines 15-17) By the security of the underlying subORAM scheme, the batch access algorithm and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Line 18) These lines are identical and create an array where the number of objects is based on public information and the object size is a public parameter.
- (Line 19) These lines set the same function.
- (Line 20) By the security of the underlying sorting algorithm, the oblivious sort and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Line 21-24) The structure of the loop is the same in both algorithms and depends only on public information ($N + \alpha S$), and the compare-swap primitive guarantees that the algorithm

and the simulator algorithm produce indistinguishable memory access patterns.

- (Line 25) Creates a list where the list size is based on public information ($N + \alpha S$) and the object size is public.
- (Line 26) By the security of the underlying compaction algorithm, the oblivious compaction and the corresponding simulator algorithm produce indistinguishable memory access patterns.

While the memory access patterns generated are indistinguishable in all cases, the adversary could potentially be able to distinguish between the real and ideal experiments if the adversary could cause the responses between the real and ideal experiments to differ. The only way that the adversary could do this is if the number of requests assigned to a subORAM exceeds $f(N, S)$ for N total requests and S subORAMs. The load balancer algorithm guarantees that requests in a batch are distinct (we use oblivious compaction to remove duplicates) and randomly distributed (we use a keyed hash function). Furthermore, the attacker cannot learn information about how requests are routed to subORAMs because the access patterns do not leak the assignment of requests to subORAMs (as proven above). Thus we can apply Theorem 4.3, and so the probability that a batch overflows is negligible in λ. Finally, Theorem 4.4 guarantees that reads always see the result of the last write, and so, the probability that the adversary can distinguish between the real experiment and the ideal experiment is negligible in λ. □

4.12.7 Discussion of multiple clients

Our proof only considers a single client, and so we briefly (and informally) discuss how to extend our guarantees to multiple clients. In the case where multiple clients are controlled by a single adversary, we simply need to modify the adversary to choose requests for each client, and then the clients forward the requests to the oblivious storage system. The oblivious storage protocol and the ideal functionality then needs to route the correct response to the correct client (rather than sorting by object ID on line 19 in Figure 4.24, the load balancer can sort by the client ID, object ID, bit b tuple).

We now consider the case where there is an honest client submitting read requests and all other clients are controlled by the adversary. Note that write requests cannot be private in the case where the adversary can make read requests, as the adversary can always read all objects to tell what objects was written to by the honest client. We simply want to hide the contents of the read requests made by the honest client (we do not hide the timing or the number). In our proof, we show that the trace generated by operating on the batch of requests submitted by the adversary is indistinguishable from the trace generated by operating on a random batch of requests, and so the execution trace will not reveal information about the honest client's accesses. Using the modification described above, we also ensure that the correct responses are routed to the correct client, and so the adversary cannot learn information about the honest client's read requests from the returned responses.

4.13 Linearizability

. Snoopy implements a *linearizable key-value store*. We define the following terms:

- An operation o has both a start time o_{start} (the time at which the operation was received by a load balancer), and an end time o_{end} (the time at which the operation was committed by the load balancer).
- Operation o' follows operation o in real-time ($o \underset{rt}{\rightarrow} o'$) if $o_{end} < o'_{start}$.
- o' and o are said to be concurrent if neither o nor o' follow each other.
- Operations can be either reads ($read(x)$, which reads key x), or writes ($write(x, v)$, which writes value v to key x).

. Linearizability requires that for any set of operations, there exists a total ordered sequence of these operations (a linearization – we write $o \rightarrow o'$ if o' follows o in the linearization) such that:

- The linearization respects the real-time order of operations in the set: If $o \underset{rt}{\rightarrow} o'$ then $o \rightarrow o'$ **(C1)**.
- The linearization respects the sequential semantics of the underlying data-structure. Snoopy follows the semantics of a hashmap: given two operations o and o' on the same key, where o is a write $write(x, v)$, and o' is a read $read(x)$, then, if there does not exist an o'' such that $o'' = write(x, v')$ and $o \underset{rt}{\rightarrow} o'' \underset{rt}{\rightarrow} o'$, then $read(x) = v$. In other words, the data structure always returns the value of the latest write to that key **(C2)**.

As in our security proofs, we prove linearizability separately for our subORAM scheme and for Snoopy instantiated with any subORAM.

Theorem 4.4. *Snoopy is linearizable when the subORAM is instantiated with a oblivious storage scheme that is secure according to Definition 4.2.*

Proof. We prove that there exists a linearization that follows the hashmap's sequential specification: each operation is totally ordered according to the (batch commit time *epoch*, load balancer id *lb*, operation type *optype*, batch insertion index *ind*) tuple (sorting first by batch commit time, next by load balancer id, next giving priority to reads over writes, and finally by arrival order) . Let $o_1 \rightarrow o_2 \rightarrow ... \rightarrow ..o_n$ be the resulting linearization. We prove the aforementioned statement in two steps: (1) the statement holds true for $o_n \rightarrow o_{n+1}$, and (2) the statement holds true transitively. Note that we assume load balancers and subORAMs can take a single action per timestep.

1. $o_n \rightarrow o_{n+1}$ We prove this by contradiction. Assume that $o \rightarrow o'$ violates either condition C1 or condition C2.
    - **(C1)** Assume that condition C1 is violated: $o_{end} \geq o'_{start}$. Now, consider $o \rightarrow o'$: it follows by assumption that $(batch_o, lb_o) \leq (batch_{o'}, lb_{o'})$. If $lb_o == lb_{o'}$, o and o' are either in the same epoch or o' is in the epoch that follows o at the same load balancer. In both cases, o' cannot have a start time greater or equal than o's start time: each load

balancer processes each epoch sequentially and waits for all batches to commit. We have a contradiction. Consider next the case in which $batch_o == batch_{o'}$ and $lb_o \leq lb_{o'}$. We have $o_{start} < batch_o < o_{end}$ and $o'_{start} < batch_{o'} < o'_{end}$. As $batch_o == batch_{o'}$, we have $o'_{start} < epoch_o < o_{end}$. We once again have a contradiction.

- **(C2)** Assume that condition C2 is violated: $o = write(x, v)$ and $o' = read(x)$, but o' returns $v \neq v'$ and there does not exist an o' such that $o \underset{rt}{\rightarrow} o'' \underset{rt}{\rightarrow} o'$. We consider two cases: (1) o and o' are in different batches, and (2) o and o' are in the same batch. First, consider the case in which o and o' are in different batches and $batch_o < batch_{o'}$ (if o and o' write to the same key x and are in different batches, then $batch_o \neq batch_{o'}$ as subORAMs processes batches of requests sequentially). It follows that o' executed after o. There are two cases: (1) o is the write in the batch with the highest index, and (2) there exists a write o'' with a higher index. In the latter case, we have a contradiction: our linearization order orders writes by index, as such there exists an intermediate write o'' in the linearization order $o \rightarrow o'' \rightarrow o'$. Instead, consider o to be the write with the highest index. This write gets persisted to the subORAM as part of the batch. By the correctness of the underlying oblivious storage scheme, a read from oblivious storage (instantiated in our system as a subORAM, see Theorem 4.5) returns the latest write to that key. As such, if o' reads x in a batch that follows o's write to x with no intermediate writes to that key, o' will return the value written by o. We have a contradiction once again. (2) If o and o' are instead in the same batch, then $batch_o == batch'_o$. By our linearization order specification, reads are always ordered before writes in a batch, so $o' \rightarrow o$. We have a contradiction.

2. Transitivity. The proof holds trivially for chains of arbitrary length $o_1 \rightarrow .. \rightarrow o_n$ due the transitive nature of inequalities and the pairwise nature of operation correctness on a hashmap.

□

Theorem 4.5. *Our subORAM (Figure 4.18) always returns the value of the latest write to an object, provided that it is instantiated from a two-tiered oblivious hash table [49], an oblivious compare-and-set operator, and an oblivious compaction algorithm.*

Proof. We prove this by contradiction. Assume that the last write to object o was value v and a subsequent read of object o in epoch i returns value v' where $v \neq v'$. Because reads are ordered before writes in the same epoch, a write cannot take place between the end of the end of epoch $i-1$ and a read in epoch i. Then, by the correctness of the oblivious hash table (which we use to retrieve the correct request for an object when scanning through all objects), the oblivious compare-and-set primitive (which copies the object value correctly to the request's response data if the request is a read), and oblivious compaction (which ensures that entries in the hash table corresponding to real requests are returned) it must be the case that the value for object o in the subORAM at the end of epoch $i-1$ is v'. By the correctness of our oblivious hash table (which we use to retrieve the correct request for an object when scanning through all objects) and oblivious compare-and-set primitive (which copies the request value correctly to the object value if the

request is a write) and because write requests in the same batch are distinct (our load balancer deduplicates requests in the same epoch), the last write to object o before epoch i must have been value v'. Thus we have reached a contradiction ($v \neq v'$), completing the proof. □

4.14 Access control

Throughout this work, we assume that all clients are trusted to make any requests for any objects. However, practical applications may require access control. We now (informally) describe how to implement access control for Snoopy. A plaintext system can store an access control matrix and, upon receiving a request, look up the user ID and object ID in the matrix to check if that user has the privileges to make that request. In an oblivious system, the challenge is that the load balancer cannot query the access control matrix directly, as the location in the access control matrix reveals the object ID requested by the client. We instead need to access the access control matrix obliviously.

We can do this using Snoopy recursively. In addition to the objects themselves, the subORAMs now need to store the access control matrix, where each object has the tuple (user ID, object ID, type) as the key (where type is either "read" or "write") and 1 or 0 as the value depending on whether or not the user has permission for that operation. The load balancer then needs to obliviously retrieve the access-control rule pertaining to the requests it received from the clients and apply the access-control rule when generating responses for the clients. Notably, if a client does not have permission to perform a read, Snoopy should return a null value instead of the object value, and if the client does not have permission to perform a write, it should not copy the value from the request to the object. In order to ensure that a user is querying with the correct user ID, users should authenticate to the load balancer using a standard authentication mechanism (e.g. password or digital signature).

Now, upon receiving a request, the load balancer generates a read request to the access control matrix for the tuple (user ID, object ID, type) corresponding to the original request. The load balancer generates batches of access-control read requests that it shards across the subORAMs. This is equivalent to running Snoopy recursively where the load balancer acts as both a client and load balancer for the batch of access-control read requests. When the load balancer receives the results of the access-control read requests, it then matches the access-control responses to the original requests by performing an oblivious sort by (user ID, object ID, type) on both the access-control responses and the original list of requests. The load balancer scans through the lists in tandem (examine both lists at index 0, then at index 1, etc.), copying the bit b returned in the access-control response to the original request. The load balancer then sends the original requests (including this new bit b) to the subORAMs as in the original design of Snoopy.

When executing the requests, the subORAMs additionally check the value of b in the oblivious compare-and-set operation (lines 10 and 11 in Figure 4.18) to ensure that the operation is permitted before performing it. Note that it is critical that we hide which operations are permitted and which are not during execution; otherwise, an attacker can submit requests that aren't permitted and, by observing execution, see where in the sorted list of requests the failed request

was (which leaks information about the permitted requests). Executing requests with access control now requires two epochs of execution (one to query the access control matrix and one to process the client's actual request) to return the response to the user.

4.15 Conclusion

In this chapter, we proposed Snoopy, a high-throughput oblivious storage system that scales like a plaintext storage system. Through techniques that enable every system component to be distributed and parallelized while maintaining security, Snoopy overcomes the scalability bottleneck present in prior work. With 18 machines, Snoopy can scale to a throughput of 92K reqs/sec with average latency under 500ms for 2M 160-byte objects, achieving a 13.7× improvement over Obladi [72].

Chapter 5

SVR3: Secret key recovery in a global-scale end-to-end encryption system

5.1 Introduction

End-to-end encrypted messaging applications like Signal [287], WhatsApp [80], and Messenger [214] are used by hundreds of millions to billions of users. They provide end-to-end encryption: user devices (the "ends") encrypt user messages so application servers receive only encrypted messages without decryption keys. Only the users in a conversation can decrypt the messages locally on their devices. This paradigm protects user messages even if the application provider or cloud infrastructure is compromised.

To provide this guarantee, end-to-end encrypted messaging application providers must ensure that their users' secret keys and data are protected against a wide range of attacks by malicious employees, cloud provider administrators, or other privileged agents. Unfortunately, this creates a usability problem: if a user loses their device, the user loses access to their account information, metadata (e.g. address book, social graph), and message history. The application provider cannot directly store a backup of this information, as this would violate the core principle of end-to-end encryption. Similarly, if the application provider stores an encrypted backup of this information it must not have access to the backup's decryption keys. Users who lose their devices should be able to recover at least their account settings and metadata without the provider gaining access to this protected data.

Shortcomings of many existing key recovery systems. A potential strawman is to allow the user to download their backup encryption key (e.g., print them on a piece of paper) and store them in a safe place [166, 185, 216], but this places extra burden on the user [262]. A more user-friendly approach to this problem is to allow a user to use a password or a PIN to encrypt their key [131]. Unfortunately, these are often vulnerable to brute-force dictionary attacks [281, 283].

Furthermore, standard safeguards (e.g., forcing the attack to be performed online) can easily be circumvented by the application provider.

Current deployed systems [13, 173, 200, 303, 316, 320] prevent brute-force attacks by using secure hardware to limit the number of PIN guesses. This approach provides a strong protection against service provider administrators and cloud providers. While these systems all represent significant advances in password-based key recovery, they rely on the security guarantees of a *single* type of secure hardware. Although secure hardware is a powerful tool for enhancing the security of systems, it can eventually be subverted—attackers have extracted user secrets from secure hardware in the past [56, 59, 126, 154, 227, 257, 274, 297, 309, 310, 313, 314]. In these systems, compromising just one type of secure hardware enables an attacker to recover many users' secret keys, which is a catastrophic scenario for any popular encrypted system.

Key recovery without a single point of security failure. In this chapter, we contribute **S**ecure **V**alue **R**ecovery **3**[1], a PIN-based secret key recovery system that prevents any one type of enclave or cloud provider from becoming a central point of attack. Our security properties are informed by the observation that many vulnerabilities are quickly patched, and so it is challenging for an attacker to find vulnerabilities *on every one* of different enclave architectures within the *same* time period between rekeying events. SVR3 proposes a layered architecture, illustrated in Figure 5.1, consisting of a tailored cryptographic multi-server key recovery protocol that distributes trust across three different enclaves from three distinct hardware vendors on three major clouds: Intel SGX in Microsoft Azure, AMD SEV-SNP in Google Cloud, and Nitro in AWS. SVR3 ensures that even if an attacker compromises two of these enclave types and the respective clouds but not the third, the attacker cannot reconstruct the user's secrets due to the cryptographic protocol. The attacker needs to compromise the security of all of the clouds and all of the enclave types to reach user secrets.

We implemented SVR3 as a production-ready system embedded in Signal Messenger [287], an end-to-end encrypted messaging application with tens of millions of users. We have already deployed an initial version of SVR3's implementation to millions of users globally, and the fully featured system is in the process of deployment at the time of publication. A third-party auditor, NCC Group, audited the deployment of Signal's SVR2, a predecessor system currently in production and using SVR3's consensus protocol on a single trust domain. In production, Signal intends to use SVR3 to improve the protection of data currently protected by Signal's SVR2 service, including account settings, contact lists, and group membership information. SVR3 is open source [286] and can be used by any end-to-end encrypted system that needs secret key recovery (e.g., encrypted messaging [80, 287], email [251, 253], or storage [321]). To the best of our knowledge, SVR3 is the first deployed cross-enclave, cross-cloud secret key recovery system. The servers for SVR3 cost only $0.0025/user/year and it takes 365ms for a user to recover their key, which is a rare operation.

Design decisions. Our design choices were guided by the goal of developing a real-world PIN-

[1]This is the third generation of Signal's SVR service and succeeds SVR1 [200], which did not distribute trust across multiple types of secure hardware. (SVR2 was a transition system consisting of a partial SVR3 design.)

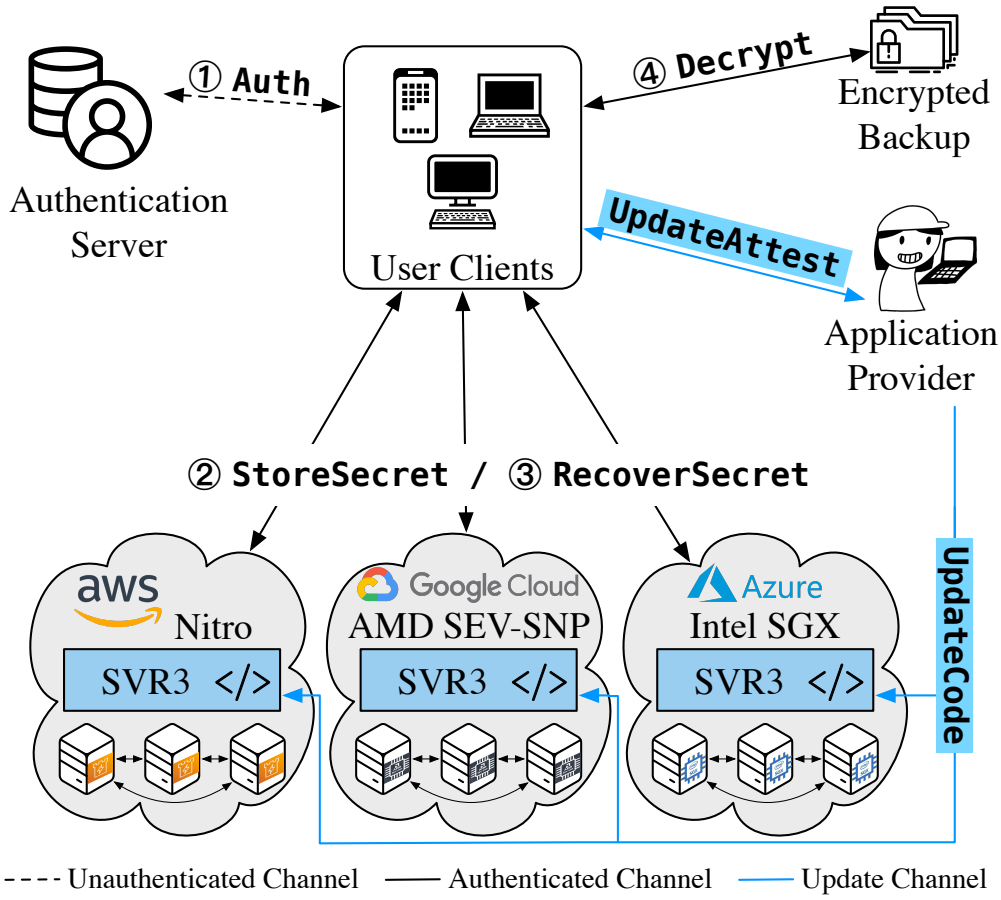


Figure 5.1: System architecture for $n = 3$ enclave clusters, with each cluster using a different type of hardware enclave.

based key recovery system that prevents dictionary attacks, is easy and affordable to maintain, and provides security even if a particular enclave or cloud provider is vulnerable. We summarize the key decisions below.

A layered security architecture (§5.2–§5.3). We aim to protect users' secrets against three major classes of attackers: cloud attackers, an internal application provider attacker, and external hackers. To achieve this, one strawman is to distribute trust across multiple organizations. However, finding reliable and trustworthy such organizations is difficult and expensive [75, 193]. Instead, we introduce an architecture that layers cryptographic security on top of hardware security by using different types of enclaves in different clouds. The hardware enclaves enable creating three separate trust domains, and the cryptographic tools split secret keys across the trust domains.

PPSS to distribute trust (§5.4). Password Protected Secret Sharing (PPSS) [18] provides password-

based key recovery while distributing trust across multiple backends and limiting attackers to online dictionary attacks. Different PPSS schemes have different deployment consequences, and we select the construction by Jarecki et al. [156] primarily because it requires no cross-trust domain communication and the server design enables clients to use different secret sharing schemes if they wish. We use this protocol to construct our one-round key recovery protocol, where the servers receive no information about whether the PIN guess was correct, and the servers unconditionally delete key material after a fixed number of PIN guesses (which can be refreshed by the clients). This is in contrast to existing works [287, 303, 320], which rely on password-based authentication and require multiple communication rounds.

Rollback protection through enclave memory and consensus (§5.5). Like Signal's original SVR1 system [287], SVR3 protects against *software* rollback attacks by keeping all data (e.g., guess counts) inside enclave memory. In order to prevent data loss, we replicate data across multiple enclaves in the same cloud. SVR1 uses the original Raft consensus protocol [235], which is not safe under *physical* rollback attacks. In principle, an attacker with physical access (e.g., a DIMM interposer [304]) to a single server in a vanilla Raft replica group could take control of the group and roll back log entries. To defend against such attacks, we develop a modified Raft [235] protocol, Raft$^{\circlearrowleft}$, that provides safety under physical rollback attacks, as specified in §5.3.2. We prove its safety under a formal TLA+ [180] model in the face of physical rollback attacks.

Secure code updates via auditing (§5.6). To enable code updates while providing strong security, we allow clients to audit the deployed code and explicitly disallow sharing of data between different (server) binary versions. Data migration between binary versions flows through the client, and clients can determine whether or not to store their secret value on each version of the binary.

Limitations. SVR3 relies on the underlying security guarantees of the enclaves it employs; supporting a new enclave or a new version of an existing enclave would require carefully reasoning about how it fits into the threat model. Splitting infrastructure across multiple cloud providers also incurs higher monetary costs than deploying on a single provider, but offers stronger security assurances. Furthermore, SVR3 does not support recovering the user PIN that is used in secret key recovery (i.e., if a user forgets their PIN, they cannot recover their key). We mitigate this in practice by periodically prompting the user to re-enter their PIN on the messaging client to prevent permanent lockout. Finally, we remark that the scope of this work is on how Signal currently implements key recovery, and not the Signal system as a whole (e.g., how the recovered key is used).

5.2 System overview

5.2.1 System architecture

Figure 5.1 shows the system architecture for an SVR3 deployment with three cloud providers, with the following entities:

Enclave clusters. The application owner deploys n enclave clusters (in our deployment, $n = 3$). To strengthen security, each enclave cluster should run on a different type of enclave in a different cloud environment (see §5.3). We will refer to each enclave cluster running on different hardware in a different cloud as a trust domain. Enclave clusters maintain replicated storage and respond to messages from clients. Each enclave cluster consists of a load balancer, a discovery service, and a geographically distributed replica group.

Authentication server. The authentication server establishes authenticated channels between clients and enclave clusters. The authentication server prevents malicious clients from exhausting PIN attempts for honest users because a client needs to authenticate to the authentication server (e.g., via an SMS code) before interacting with the enclave clusters.

Clients. Clients (e.g., mobile phones or laptops) interact with the authentication server and nodes in the enclave clusters in order to back up and recover their secret keys.

Application provider. The application provider will update the software and run monitoring and maintenance to ensure that the system is available and healthy.

5.2.2 System API

As shown in Figure 5.1, SVR3 exposes the following client API:

- $\mathsf{Auth}(\mathsf{client}, \mathsf{client\_cred}) \rightarrow \mathsf{auth\_token}$: Establishes authenticated channel between client and server.
- $\mathsf{StoreSecret}(\mathsf{client}, \mathsf{auth\_token}, \mathsf{val}, \mathsf{pin})$: Backs up a value val for an authenticated client using a human-memorable PIN value pin and an authentication token auth_token.
- $\mathsf{RecoverSecret}(\mathsf{client}, \mathsf{auth\_token}, \mathsf{pin}) \rightarrow \{\mathsf{secret}, \perp\}$: Recovers the value secret for client if (and only if)
  - auth_token is valid for client,
  - pin matches the PIN provided at StoreSecret time for client, and
  - the number of unsuccessful RecoverSecret attempts for client does not exceed a set guess limit.

  Otherwise, outputs $\perp$.

The client can use their recovered secret to locate, authorize access to, and decrypt their encrypted backup.

We describe how the developer updates SVR3 in §5.6.

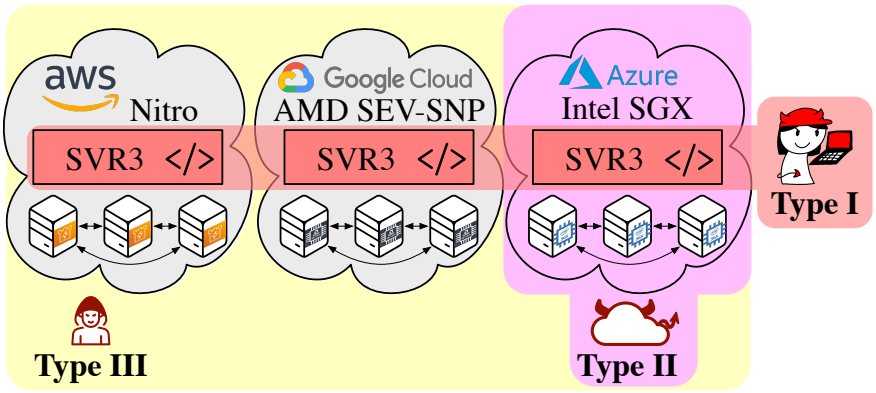


Figure 5.2: Types of attackers SVR3 protects against.

5.3 Threat model and guarantees

SVR3's goal is to protect users's secrets. SVR3 provides different security guarantees against three types of server attackers, shown in Figure 5.2:

- **Type I (Internal).** This attacker compromises the organization deploying SVR3 (e.g., a malicious employee). This attacker does not have physical access to the cloud deployment and has not compromised the clouds, but can freely spin up and bring down machines and modify the software being run.
- **Type II (Cloud).** This attacker represents an entity with control over the physical infrastructure SVR3 is deployed on (e.g., a single cloud provider). While this attacker does not have access to the multi-cloud system deployment, it can leverage physical access and tamper with the hardware running SVR3.
- **Type III (External).** This attacker is external to the deployment of SVR3 (e.g., a hacker), and tries to break-in various parts of an organization's surface.

We express SVR3's security guarantees at two levels: (1) at the level of trust domains (§5.3.1), defining security in terms of which trust domains are not compromised, and (2) at the level of enclaves inside a trust domain (§5.3.2), specifying the conditions under which a trust domain is not compromised.

Like other end-to-end encrypted systems [251, 253, 320], if a user's device is compromised, SVR3 provides no guarantees to that user. For an uncompromised user device, we rely on the trustworthiness of client code released by Signal; we enable the community to scrutinize the client code and build trust in it by making it open-source [211–213].

SVR3 does not hide the identity of clients or the timing of backup and recovery requests.

5.3.1 Security across trust domains

SVR3 protects users' secret keys if at most t out of n trust domains are compromised. We assume that the odds of an attacker identifying and exploiting vulnerabilities across $> t$ trust domains during the *same* time period between rekeys is low, which motivates our threat model. The system enables each user to rekey periodically, and deletes the old secret key.

In our deployment of SVR3, we set $t = 2$ and $n = 3$, so we ensure security as long as ≤ 2 trust domains are compromised (i.e., at least 1 trust domain is uncompromised). We limit PIN guesses by selecting a parameter u, a server *usage limit*.

Theorem 5.1 (Informal)**.** *In an SVR3 deployment configured with n trust domains, threshold t, and a usage limit u, assuming a password-protected secret sharing scheme (defined in §5.4.2), if an attacker compromises $t_c \leq t$ trust domains, then SVR3 ensures that, for each secret key, the attacker only has $\left\lfloor \frac{nu - t_c u}{t+1-t_c} \right\rfloor$ PIN attempts and, after that, cannot recover the secret key.*

We describe how SVR3 achieves Theorem 5.1 in §5.4.2.

5.3.2 Security within a trust domain

We now describe the threat model we consider when instantiating the trust domains assumed in §5.3.1. Recall that each trust domain consists of an enclave cluster and that each trust domain should use a different type of enclave.

Enclave threat model

SVR3's design is not tied to some specific enclave implementations. Different enclaves vary in design, so we abstract out the security properties that we require from the enclaves employed for SVR3's security guarantees (§5.3.2) to hold. An *uncompromised enclave* must provide:

(E1) *Application-level attestation.* The enclave can prove that certain code is running before other systems interact with it, and the attacker cannot alter the code during the enclave's execution.

(E2) *Access control.* Enclave memory is encrypted, and access control is hardware-enforced to prevent all non-enclave access.

(E3) *Page-level rollback granularity.* The attacker can replace pages of data in the enclave's memory with older pages from the same physical location and can mix and match old and new pages, thus violating global memory integrity. We assume that an attacker cannot mount these attacks at a sub-page granularity (e.g., address level) either because the enclave protects this or other protection mechanisms are used in the enclave (see below).

Deviations from enclave threat model. We describe what enclaves SVR3 uses at the time of writing this work and how they fit our threat model in §5.11. Some recent enclaves use AES-XTS, which encrypts in 16B increments [60]. While our design currently targets enclaves that can only be rolled back at the page-level granularity (E3), we can implement atomic regions (regions

that are guaranteed to run without interruption by an attacker) by utilizing the interrupt handler introduced by AEX-Notify [65]. We describe how to do so in §5.5.3. Given the changing landscape of enclave implementations and the possibility that enclaves may not adhere to (E1)–(E3) in the future, we assume that alternative mechanisms like AEX-Notify can be developed to address such discrepancies between real-world enclaves and our enclave threat model.

Attacks on enclaves. Enclaves are susceptible to attacks. We list four categories here and then in §5.3.2, we discuss when SVR3 hardens a trust domain against them.

(A1) *Memory access pattern attacks.* Enclaves do not hide memory access patterns, enabling a large class of side-channel attacks, including but not limited to cache attacks [38, 130, 220, 275], branch prediction [188], paging-based attacks [312, 324], and memory bus snooping [186].

(A2) *Software rollback attacks.* Enclaves are also susceptible to rollback attacks, also referred to as freshness or replay attacks [241]. Software rollback attacks occur from rolling back persisted state outside of the enclave's memory (**Type I** attacker).

(A3) *Hardware rollback attacks.* An attacker with physical access to the system bus can roll back enclave memory at the page level without detection (**Type II** attacker), for example, using a DIMM interposer [304].

(A4) *Other attacks.* Certain physical attacks allow an attacker to break guarantees (E1)–(E3) of enclaves (e.g., leakage due to power consumption [59, 227, 297] or denial-of-service attacks due to memory corruptions [126, 154]). Transient execution attacks [56, 257, 274, 309, 310, 313, 314] exploit speculative execution to leak secret data.

Security guarantees

SVR3 *hardens* a trust domain against a set of attacks, rendering the trust domain uncompromised despite those attacks. We describe the conditions below:

(H1) SVR3's memory-access patterns do not depend on user secret content, and hiding *which* user is recovering their key is a non-goal for SVR3, so it does not suffer from memory-access patterns side-channel attacks (A1).

(H2) SVR3 defends against software rollback attacks (A2).

(H3) SVR3 defends against hardware rollback attacks (A3) as long as $\leq s$ nodes in each cluster are rolled back, where s is a fault-tolerance ("supermajority") parameter defined in §5.5.2. In our production deployment, we set $s = 2$.

(H4) Within a trust domain, SVR3 does not guarantee protection against other attacks (A4), which could render the trust domain compromised. In this case, SVR3 still offers the cross-trust domain security guarantees in §5.3.1.

5.3.3 Availability

Like other end-to-end encrypted systems [251, 320], Signal prioritizes security over availability of secret key recovery because users' secret keys are extremely sensitive and crucial to safeguard in an end-to-end encrypted system. Nevertheless, SVR3 provides availability to clients when at least $t + 1$ trust domains are operating correctly. By correct operation, we mean that enclaves in the trust domain are online and none of the enclaves in the trust domain are under attack. Therefore, we expect the system to be available under normal operation.

SVR3 also does not defend against denial-of-service (DoS) attacks from a **Type I** attacker (since this is the organization that deploys SVR3 itself) or the authentication server.

SVR3 ensures that a malicious client cannot deny availability for an honest user (e.g., by exhausting the number of PIN attempts allowed) assuming that the attacker did not compromise the client credentials or the authentication server (used to Auth in Figure 5.1), and it did not otherwise compromise the servers beyond the availability threshold above.

It is important to consider what users would experience if trust domain(s) were to fail, leading to secret value loss. While this is a significant event when viewed from the perspective of the application provider, it will not lead to secret value loss for the majority of clients in practice: clients cache their SVR3-protected secret locally, and so clients can simply create a backup at the new deployment. Thus data loss is only a concern for users who lose their devices around when the old deployment fails and before migration to the new deployment completes.

5.4 Secret key backup and recovery protocols

We now describe the cryptographic protocols in SVR3.

5.4.1 Establishing enclave sessions

To interact with the SVR3 servers, the client must first authenticate with the authentication server. If the user has lost their devices, then the authentication server sends the client an SMS code, and then the user enters the SMS code to receive a token. This process allows the authentication server to prevent malicious clients from denying service to honest users by exhausting all of their PIN attempts. Notably though, the authentication server does not have any information about user PINs. The client then uses this token to establish a secure channel with a replica in each trust domain. As part of the process of establishing a secure channel, the client runs remote attestation [63] with the enclaves to ensure that it is communicating with the expected enclaves.

5.4.2 PIN-protected secret sharing

In existing deployed PIN-based backup systems [173, 200, 316, 320], a secure hardware device has access to users' secret keys and PINs or PIN-derived information in order to authenticate users. This design means that an attacker that compromises the secure hardware can, either directly

or via a brute-force attack, learn user PINs. This property is particularly problematic when we consider the fact that many users re-use PINs across services.

As a result, when designing our cross-enclave cross-cloud solution, we cannot simply instantiate the above mechanism in each trust domain. Any one compromised trust domain would have access to the PIN, enabling the attacker to recover the user's secret key. Instead, we leverage the class of cryptographic protocols called *password-protected secret sharing (PPSS)* [18] protocols, which ensure that:

- An attacker that compromises $\leq t$ trust domains is still limited to an online dictionary attack.
- If an attacker fully compromises $> t$ trust domains, the attacker does not immediately learn client secrets. The attacker still must perform an offline dictionary attack on user PINs.

Identifying a suitable PPSS scheme for SVR3. Different PPSS schemes have different trade-offs [2,18,155–157], so we worked to identify the most suitable scheme for SVR3 and then tailor it to our setting. Some prior work optimizes for metrics that are not important to our deployment, but sacrifices properties that are important to us.

For example, many of these works aim to reduce the number of exponentiations to improve efficiency [2,155–157]. However, the number of exponentiations is not a bottleneck in our setting, especially because the number of trust domains (3) is small. The scheme with the fewest exponentiations [157] also requires coordinated server initialization and necessitates choosing secret sharing parameters at deployment time. Coordinated initialization could require us to redeploy all trust domains every time a single trust domain requires a security upgrade, and cross-trust-domain communication with security against **Type I** attackers is difficult. Choosing a secret sharing scheme at deployment time tightly couples PPSS parameters with clients and servers, removing the flexibility to modify client PPSS parameters without also changing the servers.

With these priorities in mind, we identified the PPSS from Jarecki et al. [156] as the most suitable because it is particularly simple: each backend generates a new secret key for a client when the client creates a new backup and then uses this key to evaluate an oblivious pseudorandom function (OPRF) [105] during secret reconstruction. Informally, a pseudorandom function (PRF) is a keyed function $F_k(\cdot)$ that, for a randomly chosen key k, appears to be random (indistinguishable from a function chosen uniformly at random from all functions with the same domain and range), even though it is deterministic and efficiently computable. An *oblivious* PRF is a two-party protocol where the server holds k and the client holds some input x. The protocol enables the client to learn $F_k(x)$ without the server learning anything about x or $F_k(x)$.

This PPSS scheme has several properties that are appealing for a real-world deployment:

- The protocol is one-round and concretely efficient.
- Different trust domains do not communicate with each other.
- Servers need minimal configuration. In particular they do not need any information about the threshold scheme being used, and different clients can use the same server with different threshold schemes.
- The protocol can use a standards-track OPRF with optional verifiability [79].

We note that the WhatsApp key recovery system uses a password-authenticated key agreement (PAKE) scheme [80, 320], and SVR3 does not. While PAKE protocols are a commonly cited application for PPSS schemes, we do not need to establish a session between our client and a server. We only need to recover a secret key, which is a simpler problem. Since branching while fetching secret shares is not sensitive, we do not need to layer oblivious data retrieval on top [76, 217].

Augmenting PPSS with usage limiting. Limiting attackers to a fixed number of password guesses is a core requirement for SVR3. While the application provider can use an authentication server for access control and rate limiting, this only restricts external users. SVR3 must limit powerful attackers with full administrative and physical access to the servers to the same finite number of guesses.

We solve this by leveraging our distributed-trust setting to enforce a *usage quota* on OPRF evaluations. A standard OPRF [105] allows a server with a PRF key to evaluate a PRF on a client input without learning the input. SVR3 allows the client to set a usage limit, u, at registration time, and each honest trust domain will delete that client's OPRF key after u OPRF evaluations. In order to instantiate an honest trust domain, we use enclaves to ensure that the server enforces the usage limit. Note that the security guarantees provided by PPSS and the heterogeneous enclaves are tightly coupled: the enclaves are critical for instantiating trust domains, and PPSS enables splitting a secret value across different trust domains.

In the below proposition, we bound the number of total OPRF evaluations based on the threshold t and trust domains n, providing the protection described in Theorem 5.1.

Proposition 5.2. For a (t, n) instance of PPSS [156] with a usage-limited OPRF configured to allow u evaluations, an adversary (that has compromised no trust domains) has at most $\left\lfloor \frac{nu}{t+1} \right\rfloor$ PIN attempts before the secret cannot be recovered.

Proof. Only nu OPRF evaluations are possible in the system. $t + 1$ evaluations are needed to perform one PIN attempt. After $\left\lfloor \frac{nu}{t+1} \right\rfloor$ PIN attempts, $(t + 1) \left\lfloor \frac{nu}{t+1} \right\rfloor$ OPRF evaluations have been used. Only $(t + 1)\{nu/(t + 1)\} < t + 1$ more evaluations are possible, where $\{\}$ denotes the fractional part, that is, $\{x\} = x - \lfloor x \rfloor$. This is not enough to reconstruct the secret. □

When an attacker has compromised $t_c \leq t$ trust domains, we are left with a (n', t') instance of a PPSS system described in Proposition 5.2 where $n' = n - t_c$ and $t = t - t_c$, which results in the bound described in Theorem 5.1.

5.5 Building a SVR3 backend

We now describe SVR3's system design within one trust domain. Per our threat model in §5.3, each uncompromised SVR3 trust domain consists of a cluster of machines, which we assume behave correctly except for possible physical rollback attacks and crash failures within a specified bound.

5.5.1 Design decisions

We first provide an overview of the design decisions behind SVR3's design to ensure fault tolerance and the security guarantees in §5.3.2.

Use of enclaves. In order to protect server secrets and allow clients to check the code that is processing their data, we run the core part of the service in an attested, confidential enclave.

In-memory database to avoid sealing. Data sealing is a mechanism whereby an enclave can encrypt internal state with a key that is unique to the platform and enclave, persist the encrypted data to disk, and then recover it if the enclave is torn down and restarted. As noted in prior work [85,318], applications in commercially available enclaves that use data sealing to store state externally and recover from crashes are vulnerable to simple, software-based rollback attacks. Since a core function of SVR3 is to faithfully maintain a per-user OPRF evaluation count, rollback attacks would undermine the system and could allow an attacker unlimited online password guesses. To prevent this and achieve (H2), the enclave that stores the database of client secrets and usage counters is kept entirely in enclave-protected memory; it is *never* sealed and written to untrusted memory or disk. We show that the database fits entirely in memory without sharding users in §5.8.1.

Distributed consensus. Without a data persistence mechanism (e.g., data sealing), the servers cannot recover from crashes, and data in any failed server will be lost. To ensure that data is not lost, we build the service as a geographically distributed database. To ensure split-brain or other attacks do not allow excess PIN guesses, we use a distributed consensus protocol, modified from Raft [235]. We give a high-level overview of vanilla Raft in §5.5.2. Our modified Raft protocol, Raft$^{\circlearrowleft}$, which we describe in §5.5.2, hardens vanilla Raft against physical rollback attacks and ensures that client requests and usage count changes are committed before responding to client queries. We describe in §5.5.3 how we use Raft$^{\circlearrowleft}$ to achieve global integrity across the database when assuming page-level rollback granularity of enclaves (E3), achieving (H3).

5.5.2 Rollback-resistant consensus protocol

SVR3 already protects against the class of rollback attacks that arise from storing state outside of the enclave by keeping all state in memory. However, as discussed, machines can fail, and so in order to tolerate failures without losing data, we use Raft$^{\circlearrowleft}$, a modified version of vanilla Raft across enclaves from a cloud provider. A full TLA+ description of Raft$^{\circlearrowleft}$ is available in §5.15, and we provide a proof of safety based on the TLA+ specification in §5.14.

In this chapter, we use n to refer to the number of trust domains and m to refer to the number of replica machines *within* a trust domain.

Vanilla Raft background

Raft [235] is a consensus algorithm that manages a replicated log across multiple nodes (replicas). It elects a single leader replica that receives and replicates log entries to the other follower

replicas. The leader handles all client requests by appending new log entries and sending an AppendEntriesRequest to each follower for the duration of its *term*. Follower replicas respond to requests from the leader to replicate log entries. If the leader fails, a new leader is elected through a leader election process. Log entries are identified by `<index, term>`, where `index` is the log position and `term` is the current term number. There is at most one leader in any given term. A leader forces the followers' logs to duplicate its own: conflicting entries in follower logs (with some `term` t) will be overwritten with entries from the leader's log if the leader's term t' is $\geq t$. For f crash failures, Vanilla Raft requires $m \geq 2f + 1$ replicas in order to provide safety and liveness.

The physical rollback problem

While keeping the database in memory protects against software rollback attacks, an attacker with physical access to the system bus could roll back enclave memory at the page level. Since such an attack is more expensive to perform than software-based rollback attacks, we can significantly improve security by requiring an attacker to perform these attacks simultaneously on multiple enclave replicas. With this context, we note that the vanilla Raft protocol [235], as specified, will allow an attacker who can roll back a Raft leader to make an unlimited number of PIN attempts: the Raft protocol does not look at log contents, so if a leader is rolled back and sends an AppendEntriesRequest for a new `<index, term>` log entry at an old log index, followers will accept it and allow the leader to commit.

Prior work [85,318] has addressed a problem close to this one, but with important differences. First, they are designed for data-sealing rollbacks, which do not affect SVR3 because we do not use data sealing. Second, Raft$^{\circlearrowleft}$ also defends against physical rollback attacks, which prior works do not consider in their threat model. Physical rollback attacks are more difficult to detect than data-sealing rollback attacks: after a crash recovery, the new enclave has to execute code that decrypts the sealed data to rebuild the internal state and every data-sealing rollback needs to have the enclave go through this code path. The RR protocol [85] takes advantage of this process to detect data-sealing rollback attacks. Finally, existing protocols aim to ensure liveness in the face of rollback attacks, and this is an explicit non-goal for SVR3 as mentioned in §5.3.3.

Rollback prevention in Raft$^{\circlearrowleft}$

Together, the following additions to the Raft protocol enable us to prove safety of Raft$^{\circlearrowleft}$ in the presence of an attacker who can simultaneously mount physical rollback attacks against $\leq s$ nodes. For m Raft$^{\circlearrowleft}$ servers in a trust domain, s must be strictly smaller than m to ensure safety (§5.5.2). However, to ensure fault tolerance and liveness in the face of crash failures, s should be even smaller (§5.5.2).

Hash chain. Instead of using `<index, term>` to identify a log entry, as in Raft, we use

$$\texttt{<index, term, hash}_{\texttt{index}}\texttt{>}$$

where $\texttt{hash}_{\texttt{index}} = \mathsf{Hash}(\texttt{entrydata, index, term, hash}_{\texttt{index-1}})$, `entrydata` is the contents of the log entry, and Hash is a cryptographic hash function. When a follower receives an AppendEntriesRequest, it computes the expected hash chain value and verifies that it matches the value in the request. If the values do not match, the follower rejects the request.

This prevents the simple rollback attack on Raft described in §5.5.2. However, it is still possible for an attacker who can roll back one server to gain unlimited password guesses by triggering an election with a quorum of servers that did not see the log entry for the first client request.

Supermajority. To ensure that an attacker capable of rolling back a single server cannot gain extra password guesses by triggering an election, we require quorums to have a supermajority of replicas so that the intersection of any two quorums contains more than s replicas, where s is a configurable parameter that is included in the server's attestation. This allows clients to be certain of the value of s used by the service and decide whether to accept it. We prove that an attacker must be able to roll back more than s enclaves to roll back a log entry that was committed by this Raft$^{\circlearrowleft}$. This supermajority parameter is comparable to PBFT's Byzantine nodes value [46].

Promise round. We add a *promise round* to the protocol. Once a quorum of servers acknowledges seeing a log entry, the leader will "promise" this entry by advancing its `promise_idx` to the index of this entry. A promised entry is not committed, but no replica will delete an entry that has been promised. This completes the first round. The leader now sends its `promise_idx` to all followers in its next AppendEntriesRequest, and followers will update their own `promise_idx` to match the leader's when they process the message. From this point, these followers have promised the log entry and will not delete it. The followers send their current `promise_idx` with each AppendEntriesResponse. Once a quorum of replicas has promised an entry, it is committed.

Without the promise round, the attacker could commit a log entry, roll the leader back, send two log entries, have the leader send AppendEntriesRequests to replicas that did not receive the earlier request, and then call an election. Replicas in the original quorum cannot validate the candidate's hash chain and will vote for the longer log, which contains a different entry than the one that was committed. With the promise round, the attacker must roll back all servers that promised the log entry or remove those servers from the group and add new servers in order to perform subsequent attacks and equivocate on the promised log entry.

Safety

In order to achieve safety, the number of machines in the enclave cluster must be larger than the number of rollback attacks we want to tolerate ($m > s$). As liveness under rollback attacks is a non-goal for SVR3 (an attacker with physical access can easily deny service), we decouple the constraints on m with respect to rollback attacks (s) and crash failures (f_c). We describe how s impacts liveness under crash failures in §5.5.2. We prove that Raft$^{\circlearrowleft}$ is safe under a bounded number (s) of physical rollback attacks within a trust domain.

Theorem 5.3 (Informal). *Let M_R be the maximum number of machines in an enclave cluster that can be rolled back and s be our supermajority configuration parameter. If $M_R \leq s$, then under*

standard cryptographic assumptions, for every log entry <index, term, $hash_{index}$> that has been applied to the state machine of a server i, server i will never apply a different log entry at this index.

Proof sketch. The argument follows the proof of safety in Ongaro [234] and relies on the observation that any two quorums will have an intersection that includes at least one server that has not been rolled back. We must address the fact that in the presence of rollbacks, Lemma 3 in Ongaro [234] does not hold. This poses a significant challenge, and forces us to introduce a new concept of *live committed* entries that is subtly different from the prior notion of committed [234]. With our definition, future leaders may not have a live committed entry in their log, but if they do not then they will be unable to commit new entries, so we retain safety at the expense of liveness. The major point where the argument from Ongaro [234] breaks down in our setting is in points 7.c.ii.B and 7.c.iii.B in the proof of their Lemma 8. Our argument uses the hash chain and promise index to show that there is a voter in the intersection of two quorums that has not been rolled back and will not replace the log entry. The complete proof of safety is in §5.14.

Liveness

We do not provide liveness for a trust domain under the setting of an attacker mounting physical rollback attacks, as the attacker could trivially deny client requests by taking the entire enclave cluster offline. When assuming no attacks within a trust domain, Raft$^{\circlearrowleft}$ requires $f_c \leq \lfloor (m-s)/2 \rfloor$ *crash* failures to be live under normal connectivity conditions, where m denotes the number of replicas in a trust domain (enclave cluster) and s denotes the supermajority parameter described in §5.5.2. This is due to the quorum size being $\lfloor (m+s)/2 \rfloor + 1$ enclaves. It remains an open problem to prove liveness of Raft in this setting (e.g., by formal verification [132]). Nevertheless, as discussed in §5.3.3, SVR3 still provides availability to clients when at least $t+1$ trust domains are operating correctly.

Self-healing for simple maintenance

We implement the process for replica group membership changes described in the Raft paper [234] and add a layer of automation. In Raft$^{\circlearrowleft}$, a replica group has a configured target number of voting members. For a healthy configuration, a replica group in our system will have this number of voting members as well as several non-voting members that stay up to date and service client requests. If some voting member is not seen by the leader after a configurable timeout, the leader will initiate a membership change that demotes the missing replica to non-voting status. After an additional timeout, it will remove the replica from the group entirely.

Furthermore, whenever the number of voting members is below the configured target, the leader will check to see if a non-voting member is present and initiate a membership change promoting a non-voting member to voting status.

With these mechanisms in place, administrators simply need to launch new instances and direct them to the discovery service with group information. The new server will then request

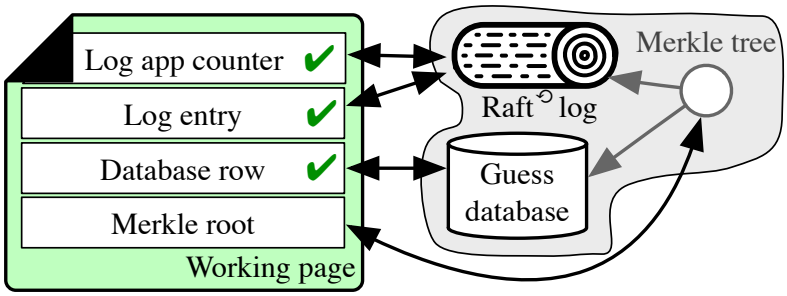


Figure 5.3: Integrity across database. In order to achieve global integrity, updates are only applied when all state on the working page validates under the same Merkle tree root.

to join the group, be brought up to date by a peer, and become a non-voting member. As needed, the voting members may then promote this new replica to voting status.

5.5.3 Integrity across the database

Raft$^{\circlearrowleft}$ provides protection against rollback attacks on the contents of the log. However, our threat model (§5.3) assumes *page-level* rollback granularity on memory inside the enclave, which means that the attacker can replace pages of data in the enclave's memory with older pages from the same physical location and can mix and match old and new pages, thus violating *global* memory integrity.

In order to protect against rollback attacks on the backing in-memory database, SVR3 keeps a Merkle tree across the Raft$^{\circlearrowleft}$ log, database, and log application counter.

Merkle tree

The log application counter keeps track of the latest log entry that has been applied to the database. The Merkle tree contains every database row, the hashchain of the most recently committed log entry, and the log application counter. The hashchain of the last committed log entry, as described in §5.5.2, can be used to verify this entry and earlier entries in the log. As shown in Figure 5.3, the Merkle leaves for database rows and log application counter are updated each time the underlying object changes, and the update only succeeds if the current state of the Merkle tree is consistent with the previous value of that data.

Applying committed log entries

We describe how we process committed log entries in Algorithm 5.1. The executing thread holds a lock on the database, log, and log application counter throughout execution, so no honest process will have a thread outside this process change the Merkle tree during that execution. When applying a committed log to the local database, a replica will begin by reading the log application

counter lac, the log entry at that index entry, and the database row row referenced by that log entry onto a single memory page, which we will call the *working page*. When reading each of these items, it will verify its Merkle proof ($\Pi_{\mathsf{lac}}, \Pi_{\mathsf{entry}}, \Pi_{\mathsf{row}}$) and also copy the root of the Merkle tree for each read onto the working page. After copying this data, we verify that the Merkle roots associated with each read are equal, determine whether the number of uses of this row has surpassed the configured maximum, and update the row by incrementing the usage count and deleting the OPRF secret if the maximum usage count has been exceeded. We then update the row in the database and increment the log application counter, updating the Merkle tree entries for both, then proceed with evaluating the OPRF, if the key is present, and finally respond to the client.

If the attacker rolls back the database row to the contents of a previous timestep, it first has to roll back every entry from the row to the Merkle tree root. However, the root also covers the log entries and log application counter, which are modified when a database row is modified (how SVR3 achieves atomicity of this operation is described above). Thus, the attacker will have to roll back the log as well; rolling back the log is exactly what Raft$^{\circlearrowleft}$ protects against.

Atomic regions. Because all of our working memory fits on a single page, operations are atomic with respect to the attacker's ability to rollback memory at the page granularity. In order to support more modern enclaves that only have cache line granularity (e.g., 16B), we need to implement atomic regions that are guaranteed to run without interruption by an attacker. We describe in detail how to implement atomic regions on SGX and SEV-SNP in §5.13 by utilizing the interrupt handler in AEX-Notify [65]. AEX-Notify mitigates SGX-Step, an attack framework that makes it possible to single-step enclave programs [311]. It does so by introducing an instruction set architecture extension to support a custom handler on interrupt. The SGX-Step mitigation leverages this handler to speed up the next instruction so that the attacker is statistically unlikely to 'hit' the next instruction's execution with an APIC timer. This mechanism also allows us to implement atomic regions, in a similar fashion to restartable sequences [27]. At a high level, we set a flag in a fixed register when an interrupt occurs, and we check this flag at the end of the atomic region to determine whether to restart the atomic region. If the flag is set, we restart and retry until it runs without any interrupt. We leave optimizing this approach in a secure manner to future work.

5.6 Operations

Production systems need upgrades. This is a challenge for us because we want to defend against malicious administrators: a secure system can become completely insecure if a malicious administrator can push arbitrary code to the system. At a high level, we defend against malicious code updates by ensuring that users can audit the code that is running; the code is open source, and enclaves attest to the security-relevant server code and configurations running.

Adding new servers. When a new server is launched in a trust domain, it connects to a discovery service and registers a new group if no replica group is registered. If there is an existing replica

Algorithm 5.1 Applying a committed log entry. We describe in text how we process committed log entries in §5.5.3.

1: $\mathsf{workspace}_R \leftarrow (\mathsf{lac}, \Pi_{\mathsf{lac}}, \mathsf{entry}, \Pi_{\mathsf{entry}}, \mathsf{row}, \Pi_{\mathsf{row}})$

Atomic region.

▷ Abort on any Verify failure.

2: failure $\leftarrow 0$
3: $\mathsf{Verify}(\Pi_{\mathsf{lac}}.\mathsf{root} \stackrel{?}{=} \Pi_{\mathsf{entry}}.\mathsf{root} \stackrel{?}{=} \Pi_{\mathsf{row}}.\mathsf{root})$
4: $\mathsf{Verify}(\mathsf{entry.clientid} \stackrel{?}{=} \mathsf{row.clientid})$
5: $\mathsf{Verify}(\mathsf{lac}, \Pi_{\mathsf{lac}})$; $\mathsf{Verify}(\mathsf{entry}, \Pi_{\mathsf{entry}})$; $\mathsf{Verify}(\mathsf{row}, \Pi_{\mathsf{row}})$
6: **if** row.guess_cnt $<$ max_guesses **then**
7: evaluated $\leftarrow \mathsf{OPRFEval}(\mathsf{row.sk}, \mathsf{blinded})$
8: row.guess_cnt $\leftarrow$ row.guess_cnt $+ 1$
9: **else**
10: failure $\leftarrow 1$
11: row.sk $\leftarrow 0$, row.guess_cnt $\leftarrow$ UINT_MAX
12: **end if**
13: $\mathsf{workspace}_W \leftarrow (\mathsf{row}, \mathsf{UpdatePrf}(\mathsf{row}, \Pi_{\mathsf{row}}))$

14: $\Pi'_{\mathsf{row}} \leftarrow \mathsf{UpdatePrf}(\mathsf{row})$; $\Pi'_{\mathsf{lac}} \leftarrow \mathsf{UpdatePrf}(\mathsf{lac})$
15: Check that leaves on path in $\Pi'_{\mathsf{row}}, \Pi'_{\mathsf{lac}}$ match $\Pi_{\mathsf{row}}, \Pi_{\mathsf{lac}}$.
16: **if** failure **then return** MISSING
17: **else return** (OK, evaluated)
18: **end if**

group, the new server will select a peer in that group, validate that its enclave measurements match, and create an attested connection with that peer. By checking that enclave measurements match, SVR3 ensures that an administrator cannot add a server running different code. The new server then requests to join the group, and the existing server transfers all log entries and database rows to the new server. This is done over a Noise protocol [246] channel with key resetting and hybrid post-quantum forward secrecy [245] to provide robust forward secrecy. Once the transfer is complete, the replica group goes through the membership change process to add the new server (which requires a quorum).

Sometimes security-required microcode updates need to be applied to all servers. Since all data is kept in volatile enclave memory, there is no way to reboot the machine without losing all replica data. In this situation, *all members of the cluster must be replaced.* This can be done by sequentially adding new servers on patched hardware, then terminating old servers.

Clients. Android, iOS, and desktop clients are deployed through app stores with auditable, open-source code. Each client contains hard-coded information about which enclave measurements

(for remote attestation), platform versions, and cluster configurations to accept. If a client attempts to connect to a SVR3 cluster and finds unexpected measurements or configuration, it will abort the connection.

Service upgrades and data migration. Since server enclaves can only communicate with peers that share the same enclave measurements, there is no mechanism to migrate data directly from an old version of an enclave-backed service to a new one. Instead, data migration flows through the client. To accomplish this, when a new version of a client is released that contains measurements for the new enclave, this client will recover its secret from the old servers (if it is not cached in local storage), and then it will back up its secret to the next version of the service. It takes approximately 90 days for a new client software release to fully reach the user base, so the new enclave-backed service must run alongside the older version during this 90-day window.

5.7 Implementation

We implemented SVR3 in ∼8,800 lines of C++ for the enclave and ∼5,300 lines of Go for the untrusted host. For the SGX deployment we use the OpenEnclave framework v0.19 [237] and Intel SGX v2.22. For the Nitro deployment we use the Nitro Security Module library v0.4 [232]. We use a Noise protocol [246] channel on top of TCP for communication between replicas and websockets for communication with clients. We use protobuf [252] to define formats for all wire messages. In addition to handling client and peer requests, the host offers a control interface for administration as well as sophisticated metrics collection that is integrated with our internal monitoring and reporting systems. Our implementation assumes enclave page-level integrity, and we estimate overheads for supporting 16B-level rollback granularity in §5.8.1. The implementation is open source and the consensus system is already in production use. The full system is being deployed to production at the time of publication. Production deployments use 7 geographically distributed servers and a supermajority parameter of 2. Full details about the production deployment are in §5.12.

5.8 Evaluation

We investigate the overheads of running SVR3 (§5.8.1) and the performance perceived by the end user (§5.8.2).

Evaluation setup. For the purposes of this chapter, we evaluate end-to-end performance on our organization's staging system, configured to handle 10 million users. This limit is due to available enclave memory, not compute. Staging clusters are configured with a supermajority parameter of 1 and consist of 3 environments (trust domains), each with 5 replicas deployed in the same region:

- AWS Nitro: `m5.xlarge` instances with 2 cores and 10 GB RAM per enclave ($142/month/server).

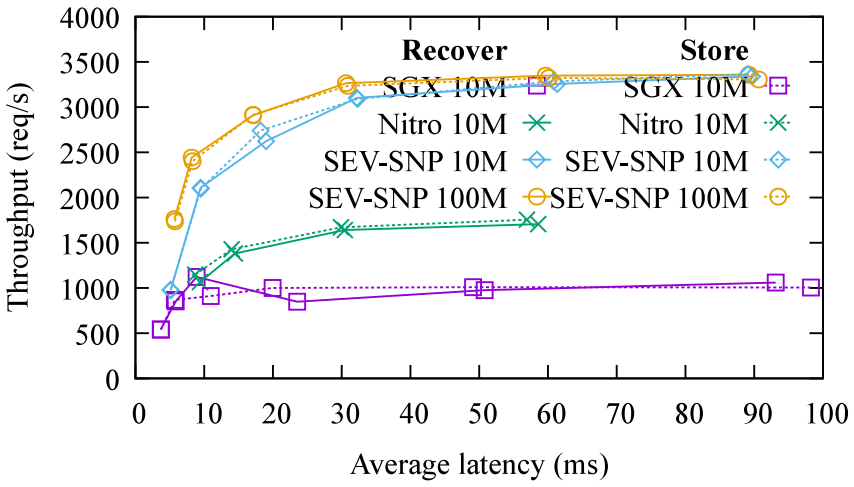


Figure 5.4: Average latency vs. throughput.

- Intel SGX at Azure: `DC2s_v3` instances with 2 cores and 8 GB EPC RAM per enclave ($140/month/server).
- AMD SEV-SNP at GCP: 2 `n2d-standard-2` instances per enclave (one "confidential" and one for the untrusted host) with 2 cores and 8 GB RAM ($2 \cdot (\$70) = \$140$/month/server).

In total, the staging cluster costs $2,110/month to run ($0.0025/user/year). For microbenchmarking, we evaluate on a testing cluster with the same machine types as our staging cluster but with 3 replicas per trust domain instead of 5 and a supermajority parameter of 0 instead of 1.

Our production infrastructure has more replicas (with more cores and RAM per replica) and is set up to handle over 500 million users (more details in §5.12). We provision for 1 req/s/1M users and ∼256B of RAM/user. Our experience operating this system gives us confidence that evaluating on the staging infrastructure is meaningful and that SVR3 scales gracefully. To validate this claim, we also evaluate on an AMD SEV-SNP cluster with 100 million users using `n2d-standard-4` instances (4 cores and 16 GB RAM).

5.8.1 Microbenchmarks

Throughput. We plot an average latency vs. throughput curve for write and recovery requests in Figure 5.4. We generate each point by varying the number of client threads and measuring the average latency and throughput of requests. Requests are spread out across all 3 servers. For the 10M-user deployments, the throughput of recovery requests levels off around 1,700 req/s for Nitro, 1,000 req/s for SGX, and 3,300 req/s for SEV-SNP (for both 10M-user and 100M-user deployments).

Latency. We plot CDFs of the latency of write and recovery requests in Figure 5.5, Figure 5.6, and Figure 5.7 for Nitro, SGX, and SEV-SNP, respectively. Within each figure, we plot the latency

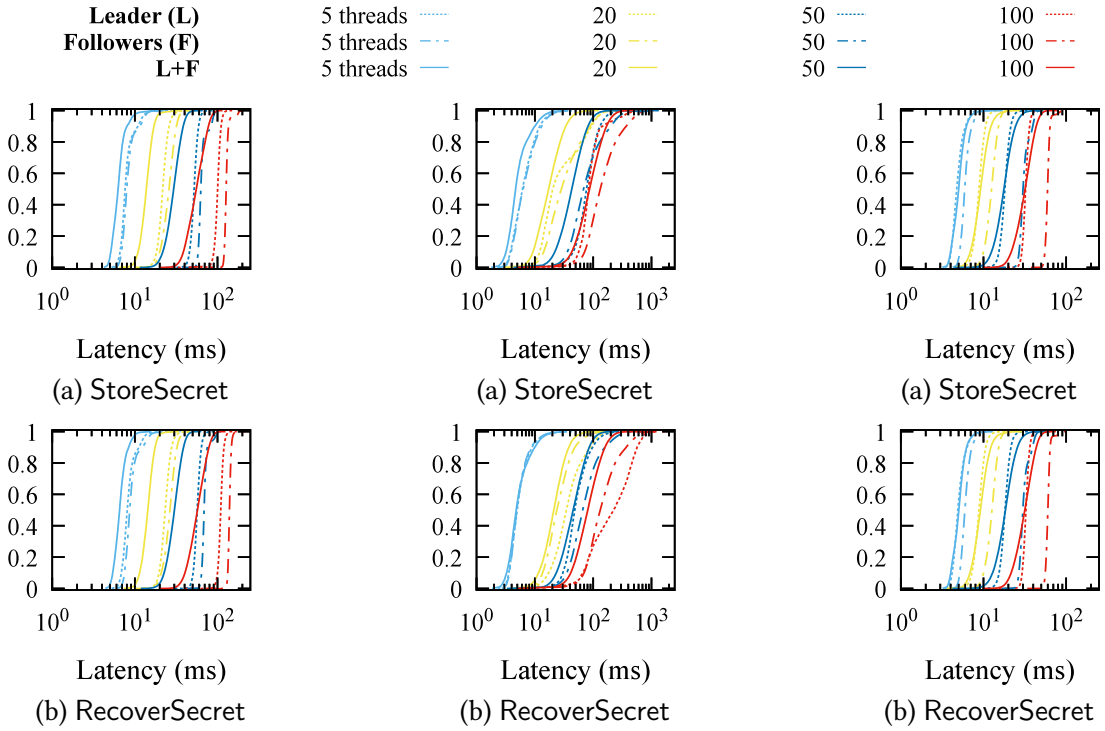


Figure 5.5: Request latency CDF for AWS Nitro, varying client threads, 10M users.

Figure 5.6: Request latency CDF for Intel SGX, 10M users.

Figure 5.7: Request latency CDF for AMD SEV-SNP, 10M users.

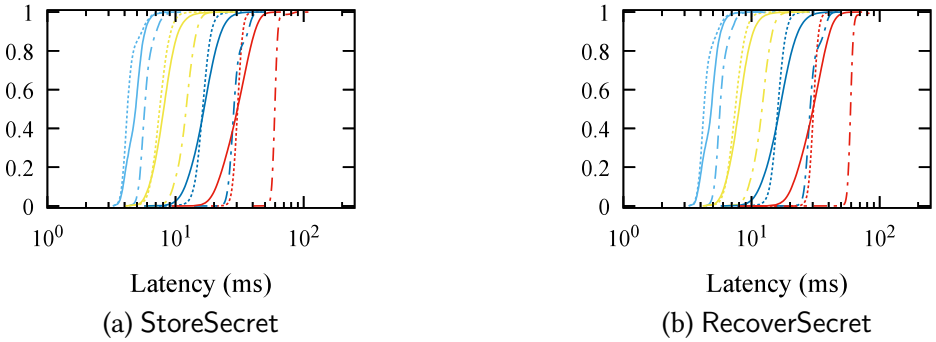
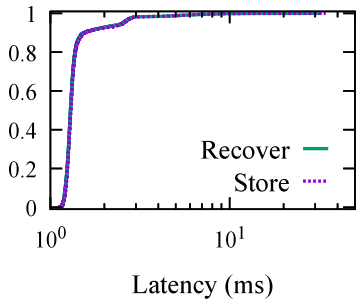
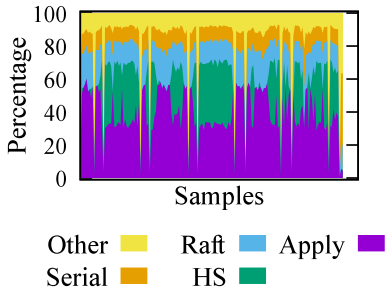


Figure 5.8: Request latency for AMD SEV-SNP, 100M users.



(a) Request latency CDF.



(b) Request latency breakdown. HS = Noise handshake, Serial = serializing/deserializing protobufs, Apply = applying log entry (§5.5.3).

Figure 5.9: SVR3 performance without network latency from Raft↻.

Enclave	**Network (B/user)**			
	StoreSecret		RecoverSecret	
	C ↔ S	S ↔ S	C ↔ S	S ↔ S
SGX	20,717	288–1,276	20,717	224–1,212
SEV-SNP	4,406	288–1,276	4,406	224–1,212
Nitro	4,593	288–1,276	4,593	224–1,212

Table 5.1: Network usage for a single client request to a 3-replica cluster. S=server, C=client. C ↔ S for SEV-SNP is an estimate.

when requests are sent only to the leader, when requests are sent only to followers, and when requests are sent to all 3 servers. Requests sent to followers are forwarded to the leader, so the average latency of requests at followers is higher than at the leader. The latency distribution of requests when sending requests to all 3 servers improves compared to sending requests to only followers. The latency distribution is better than sending requests to only the leader for Nitro and SGX, and the tail latency is worse than sending requests to only the leader for SEV-SNP. At 100 client threads, the average latency for requests sent to all servers for key recovery is 56.9ms for Nitro, 98.3ms for SGX, and 32.3ms for SEV-SNP. We also plot the CDFs of recovery request latency for the 100M-user SEV-SNP deployment in Figure 5.8. The latency distribution of the 100M-user deployment is very similar to the 10M-user deployment and the average latency of the requests sent to all 3 servers for key recovery is 30.9ms.

We note that a majority of the latency is due to network latency when appending to the Raft↻ log, which we validate in Figure 5.9. We run the same experiment as above, but with 1 client thread and 1 SGX node (effectively disabling the network requests of Raft↻). We plot the CDF of request

latencies under this regime in Figure 5.9a, and the average latency of these requests is 1.47ms. We also profile the server and plot the percentage of CPU ticks in Figure 5.9b. On average, the Noise handshake is about 35%, applying the log entry is about 21%, and 13% is encrypting peer messages for Raft$^{\circlearrowleft}$. The yellow spikes are due to periodic updating of environment statistics, which also contributes to the long tail request latencies in SGX (Figure 5.6).

Impact of supporting 16B-granularity. Informed by latency measurements, we can upper-bound the impact of latency from achieving page-level integrity from 16B-granularity using atomic regions (§5.5.3). Applying the log entry (which we will conservatively make an entire atomic region) takes $1.47 \cdot 0.21 = 0.3$ms. We could be interrupted by the APIC timer, the end of a thread scheduling quantum, or by a page fault from a memory access, of which there are $5 \cdot \log_2(100,000,000) = 120$ (from the Merkle tree accesses in algorithm 5.1). In the worst case, we would repeat execution of the atomic region 122 times, resulting in a worst-case additional latency of 36.6ms. Note that this is a (very) loose upper bound and is still below user perceptibility.

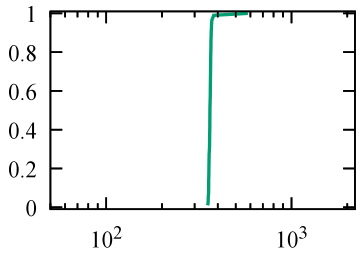
Network usage. We measure the network usage of SVR3 running on each enclave type for a 3-replica cluster in Table 5.1. There is a range of network usage for Server $\leftrightarrow$ Server because it depends on how many requests have been batched into a single Raft$^{\circlearrowleft}$ append request. The network usage between servers also depends on the number of servers in the cluster, growing proportionally to $m-1$ for m servers. From a deployment perspective, we are more concerned with the Client $\leftrightarrow$ Server bandwidth, which is under 20KB for all enclave types. This is because exchanging more data between the client and the server can become a usability issue for users with limited data plans.

Memory usage. We measured the memory usage of SVR3 on SGX, varying the number of users in the system. Note that we expect the memory usage to be similar for all enclave types, since they are storing the same amount of data for each user. We find that memory usage grows by ~450B/user until we start truncating the log at 100MB and then settles into a steady 170B/user added. At 100 million users, SVR3 uses 18.5GB of memory on each server, which is 185B/user/server.

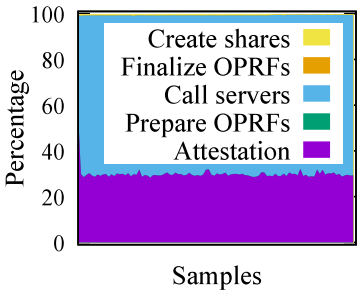
5.8.2 End-to-end performance

We measure the end-to-end performance of SVR3 by running a client that stores its secret key by sending a (sequential) request to a server in each enclave cluster. For a more representative deployment, we geographically distribute the clusters as follows:

- SGX cluster: `centralus`, `eastus`, `eastus2`, `southcentralus`, and `westus`
- SEV-SNP cluster: `us-central1`, `europe-west3`, `asia-southeast1`, `europe-west4`, and `europe-west3`
- Nitro cluster: `us-east-1`, `us-east-2`, `us-west-1`, `us-west-2`, and `eu-north-1`



(a) Request (StoreSecret) latency CDF.



(b) Client request latency breakdown.

Figure 5.10: End-to-end performance.

The performance for recovering a key is almost identical to the performance for storing a key, so we only report the performance for storing a key. We plot the CDF of the latency of these requests in Figure 5.10a. The average end-to-end latency is 365ms, which is reasonable for a user to wait for a key recovery or key backup request. We plot the breakdown of the latency in Figure 5.10b. The majority of the latency is from waiting for servers to respond (69.3%), followed by remote attestation with the servers (29.9%).

5.9 Discussion

Consensus in the enclave. Nimble [9] is able to maintain a secure log while removing the consensus mechanism from the TCB, and an important future direction for SVR3 would be to similarly minimize its TCB. However, it is not entirely straightforward, and there are interesting design and engineering challenges to address. First, Nimble will need to be hardened against physical rollback attacks, which seems straightforward to do. More significant is that since this log—which contains OPRF secrets—will be held in untrusted storage, it must be encrypted. This has important consequences for our system as we describe below, and addressing them may result in significant additional complexity (and thus increase the TCB).

First, we note that we will need enclaves similar to the ones we have today to handle client requests. These enclaves will now need to share a common encryption key to encrypt and decrypt these log messages. This shared key becomes a new single point of failure for the system. To maintain the forward secrecy we have today due to our use of Noise protocol [246] channels with rekeying between enclaves, it seems the enclaves will need to participate in some sort of continuous group key agreement (CGKA) [5] to rotate the key periodically and on membership changes.

Second, if this new system aims to keep the TCB small by maintaining the database state outside of the enclave, as with Juicebox [303] or WhatsApp [320], then the encryption key for

the database becomes another single point of failure, but in this case it is not clear how we can achieve forward secrecy without periodically re-encrypting the entire database. If, on the other hand, we maintain the database in enclave memory, as we do now, then the use of CGKA to protect the encrypted log means that new members of a replica group will not be able to read old log messages to construct the database state. While we have a state transfer mechanism in our current system to handle truncated logs, we will need to refine it to ensure that new members are correctly initialized.

Taken together, we see removal of the consensus mechanism from the TCB as a project that requires careful design and analysis and significant engineering work that adds its own complexity. We note that the consensus protocol is a relatively small (1,541 LOC in C++) and well-understood part of our current codebase, so we need—and hope to find—clear rationale for its removal.

In-memory vs. disk-based storage. While disk-based storage solutions are cheaper than keeping the entire database of key recovery shares in memory, they are more susceptible to rollback attacks because the secrets are taken out of the enclave, and even enable rollback attacks that are software-based and can be performed without physical access.

Data privacy compliance. In general, a multi-cloud deployment may complicate compliance with data privacy laws. The design of SVR3, however, keeps compliance simple since by preventing any user data from being processed by our servers and blocking our administrators from accessing sensitive keys.

Malicious clients. SVR3 provides security guarantees for users using our clients, which we assume are well-behaved. Our client code is open source [211–213], and scrutinized by the community. If the user's client is compromised and malicious (e.g., the user has malware), it can affect the security of that user, but not the security or experience of other users with uncompromised clients.

Honest cloud providers?. If we could assume that most cloud operators are honest, then that could change the parameterization of SVR3 (e.g., setting the number of trust domains that can be compromised t to 1), though this would also require assuming that the enclaves were not susceptible to any future vulnerabilities that could be exploited remotely. We would still use enclaves to prevent malicious system administrators from running arbitrary server code.

Future and ongoing work. SVR3 could be modified to support a transparency log so that users have a means of monitoring key recovery requests (similar to SafetyPin [74]) and changes in replica group membership. Currently, clients can rekey in SVR3 by reentering the user's PIN. We will eliminate the requirement for user interaction and explore an approach closer to proactive security [43], where keys can be rotated more frequently without client involvement. The OPRFs that SVR3's cryptographic protocol relies on are not quantum-safe; hardening SVR3 against an attacker that will have eventual access to a quantum computer and can harvest now, decrypt later (HNDL) [302] is also ongoing work.

5.10 Related work

Secret recovery systems. A number of companies have deployed secret recovery systems using secure hardware: Apple protects user iCloud data using hardware security modules (HSMs) [13, 173], Google protects Android backups using secure microcontrollers [316], and WhatsApp protects message histories using HSMs [320]. WhatsApp runs vanilla Raft [234] on a geographically distributed cluster of HSMs and uses OPAQUE [158] for key recovery. WhatsApp's consensus only requires one round trip between the leader and the replicas while SVR3 requires an extra round of communication (to guarantee safety in the face of rollbacks). Davies et al. analyzed the security of the WhatsApp encrypted backup protocol [80]. Like SVR3, all of these systems use secure hardware to allow a user to recover a cryptographic secret using a low-entropy secret (e.g., a 4-digit PIN). Unlike SVR3, they rely on a single type of secure hardware: the compromise of one secure hardware device can compromise many users' secrets.

Juicebox [303] is a key recovery protocol that distributes trust across one type of secure hardware and multiple trust domains in the traditional manner (across organizations). SVR3 has a simpler protocol that is not a multi-round PAKE as our servers never learn whether the PIN is guessed correctly or not (keys are deleted unconditionally when guesses run out). Secret shares are also stored directly on the servers in Juicebox. Thus, to prevent an attacker who compromises a threshold number of trust domains from reconstructing all the secrets without needing to mount a dictionary attack, they must mix the reconstructed secret with the PIN to create an encryption key that is then used to encrypt the target secret.

SafetyPin [74] is a PIN-based end-to-end encrypted backup system that defends against an attacker that can adaptively compromise some percent of HSMs. While SafetyPin protects against a more powerful attacker model, it requires a comparatively large number of HSMs.

Tutamen [271], Acsesor [52], and CanDID [203] split trust across multiple entities to allow users to recover their secrets (among other operations). Chen et al. [57] use cloud storage for secret recovery. These systems do not use secure hardware; the use of enclaves in SVR3 provides additional security and requires us to design for their limitations (e.g., rollback attacks). CALYPSO [170] also shards user secrets across different entities but, unlike SVR3, uses a blockchain. PreVeil [251] shards secret keys across other peers in a social or work graph, but requires manual setup from the user.

Another line of work has taken a more theoretical approach to the problem of secret key backups. Benhamouda et al. [25] use a proof-of-stake blockchain to allow users to store secrets while protecting against an attacker that can adaptively compromise a percent of the stake. Subsequent work improves efficiency in this model via batching [122].

Orisini et al. [239] also describe a scheme for end-to-end encrypted backups, but in their scheme, the user does not need to remember a PIN or something similar. Instead, clients must refute illegitimate recovery attempts. While this approach is appealing in that it eliminates the PIN, it does not work for our setting where clients may go offline for extended periods of time.

End-to-end encrypted backups can be vulnerable to injection attacks where changes in the backup size can allow the attacker to infer information from sensitive metadata [96]. This work

focuses on backing up cryptographic keys, and these type of injection attacks are important to consider in the context of the larger system using SVR3.

Multi-party computation and secure hardware. Cryptocurrency wallets protect user secrets by distributing them across hardware enclaves or HSMs [100, 107, 168, 266, 278]. Cryptocurrency wallets are designed to avoid materializing the key in a single location rather than to enable users to recover secrets. Myst provides security by splitting trust across many hardware devices and operations like signing and decryption [207]. More broadly, prior work has examined composing multi-party computation and secure hardware for efficiency [19, 83, 175, 233]. Our use of secure hardware with multi-party computation is tailored to encrypted backups and, while this line of work uses secure hardware to reduce the costs of multi-party computation, we use it to augment the security of the system. In prior work [75], we observed that heterogeneous secure hardware hosted by different clouds can be useful for deploying systems that split user secrets, including encrypted backups, but we had not yet worked through and built out such a deployment.

Rollback prevention in enclaves. There has been a rich line of work on preventing rollback attacks in enclaves. Memoir [241] and Ariadne [295] store a small amount of state inside non-volatile memory (NVRAM) and use that to reconstruct application state during recovery. Both approaches are scoped to single machines, and do not provide availability in the event of a machine permanently failing. ROTE [205] uses a broadcast algorithm across enclaves to maintain a distributed counter, but requires NVRAM to update group membership, whereas we use our Raft↻ log to update membership. Additionally, the abstraction that ROTE offers is one of a counter instead of generic log entries. Engraft [318] examines the safety issues of running off-the-shelf consensus inside enclaves. They use an underlying broadcast protocol similar to ROTE to maintain a distributed counter and introduce additional mechanisms to support node recoverability. However, in our setting, we can simply start a new node in the event of a node failure, so we do not need to support node recoverability.

Nimble [9] is a lightweight replication protocol that provides a freshness-guaranteed ledger. The ledger can be used to keep track of the state of untrusted storage, enabling applications that run on enclaves to persist their state to external (untrusted) storage and detect potential rollbacks on that storage. Note that our system is already protected against the class of rollback attacks on external storage described in §1 of [9] because *all data is stored and maintained in memory*. Nimble's threat model does not include physical rollback attacks on the enclave (both endorser and application). However, minimizing SVR3's trusted computing base (TCB) is an interesting and important future direction, and we discuss potential design decisions and open challenges in §5.9.

TrInc [192] shows that a secure log can be implemented with a secure counter. However, realizing a secure counter on enclaves is difficult. We cannot write PCRs to the TPM from inside an SGX enclave, and additionally, TPMs can limit the speed of counter updates (§6.1.1, [295]). CPU registers are written to the SSA, which can be rolled back. On SGX there is no CPU register where only an enclave can write to it. We are unaware of an (efficient) secure counter primitive on newer enclaves after consulting with Intel.

Consensus protocols. As Dinis et al. [85] point out, rollback behavior can be considered a subset of Byzantine behavior, so the Byzantine fault tolerant (BFT) model is stronger than necessary for our setting. Consequently, Raft$^{\circlearrowleft}$ is lighter weight than BFT flavors of Raft protocols like Tangoroa [66] which requires $O(m^2)$ communication scaling in the number of replicas. The supermajority parameter in Raft$^{\circlearrowleft}$, which increases the quorum size, is comparable to PBFT's [46] Byzantine nodes value. Engraft [318] and RR (TEEMS) [85] address data-sealing (software) rollback attacks. SVR3 not only defends against these data-sealing rollback attacks, but also defends against physical rollback attacks.

5.11 Properties of different enclaves

Intel SGX. Intel Scalable SGX (also known as SGX) attains confidentiality through hardware-based access control and encryption. The access control is obtained by placing all enclave memory inside processor reserved memory that cannot be accessed by software outside the enclave, including the OS and hypervisor. Additionally, enclave data memory is encrypted using Intel TME, which employs hardware-based AES-XTS encryption to all data pages before they leave the processor [167]. The access control provides protection for enclave data on-chip and the encryption provides protection from cold-boot and other attacks. SGX guarantees integrity in the presence of software-based attacks across the entire memory region, but does not provide this guarantee in the presence of physical attacks [159]. The use of hardware-based AES-XTS encryption of all memory pages yields ciphertexts at the 16B block level that cannot be moved but can be replayed by attackers with physical access to the system bus.

SGX provides application-level attestation. When creating an SGX enclave the system loads a dynamic library into protected memory and measures the layout of this memory, along with security flags associated with these memory pages. This measurement is provided to clients in a signed document that allows clients to confirm that the enclave is running the code expected by the client on an up-to-date platform. Thus the TCB of an SGX application includes the application library and the platform firmware. As of June 2024, SVR3 is deployed on `DCsv3` instances at Microsoft Azure, which use Intel Icelake processors.

AMD SEV-SNP. AMD SEV-SNP has memory protection that is similar to Intel SGX. All varieties of AMD SEV use hardware-based AES encryption to protect memory off chip. Additionally, with SEV-SNP, AMD adds hardware based access control and integrity and freshness guarantees. SEV uses AES-XEX memory encryption that, like Intel SGX, produces ciphertexts at the block level that cannot be moved but could be replayed [6].

SEV-SNP provides attestation at the VM level, so to obtain application-level attestation engineers must produce a restricted VM image that can only run the target application code. Thus the attested code base includes an entire VM image and hence is much larger than the attested code base for a Scalable SGX enclave running the same application. As of June 2024, SVR3 is deployed on `n2d-highmem-16` at GCP, which use AMD Rome or AMD Milan processors.

AWS Nitro. AWS Nitro enclaves run on dedicated cores and use hardware-based access control to protect enclave memory. The use of dedicated cores differs from SGX and SEV-SNP, reducing exposure to some side channel attacks. The memory protection provides integrity in the presence of software-based attacks across the entire memory region. Nitro enclaves running on Graviton 2 and 3 chips provide memory encryption [39]. While the details of this memory encryption are not public, it claims to guard against cold-boot attacks but makes no claims about security in the presence of physical attackers. Thus we expect that the implementation is similar to those of Scalable SGX and AMD SEV-SNP.

AWS Nitro has a larger TCB (the Nitro cards, security chip, and hypervisor) than Intel SGX and AMD SEV-SNP. While it is designed for application level attestation and does not present the engineering challenges that SEV-SNP does, it still requires attestation of an entire Docker image rather than the single application library attested by SGX. As of June 2024, SVR3 is deployed on `m5` instances at AWS, which use either Intel Skylake-SP or Cascade Lake processors, and we are evaluating a move to Graviton-based instances.

5.12 Production deployment

Production clusters will use 7 replicas with at least 128 GB of enclave memory and a supermajority parameter of 2. We will estimate bandwidth costs assuming 500 requests per second, a reasonable estimate for 500 million users. To deploy this at published rates will cost:

- AWS Nitro: `m5.12xlarge` (48 cores, 192 GiB memory) $1,535.62/month:
  - Compute: $10,749.34 /month
  - Bandwidth: 6150 GB client-server at $0.09/GB = $553/month
- SGX at Azure: `DC24sv3` (24 cores, 192 GiB memory) $1,681.92/month
  - Compute: $11,773.44/month
  - Bandwidth: 27,744 GB at $0.087/GB = $2,414/month
- SEV at GCP: `n2d-highmem-16` x 2 (32 cores total, 256 GiB RAM total, 128 for the TEE) $1,528.76 / month:
  - Compute: $10,701.32/month
  - Bandwidth: 13,392 GB at $0.11/GB = $1,473/month
- **Total cost: $37,663/month**. This deployment will comfortably support over 500 million users, giving an operating cost of $0.0009/user/year.

5.13 Atomic regions

To prevent attackers from exploiting gaps between time-of-check and time-of-use data, we need a way to guarantee that a segment of code runs without interruption and that certain working data is non-volatile during its execution. We accomplish this on the SGX and SEV-SNP platforms

using custom interrupt handlers, but we do not currently have a means to implement atomic regions for AWS Nitro.

SGX Implementation. The key observation that allows us to implement atomic regions on the SGX platform is that the AEX-Notify ISA extensions [16] let us implement a custom AEX-Notify handler that performs the SGX-Step mitigation of [65] and also sets a flag in a fixed register which we will denote `IR` to notify the application that it was interrupted. We can then implement atomic regions as follows:

1. Enable AEX-Notify and register a custom AEX Notification handler that performs the single-step mitigations of [65], sets the value of `IR` to `0x1` in the atomic prefetching phase, and loads two arrays of workspace data - one for reading and one for writing - into L1 cache.
2. Begin an atomic region by setting `IR` to `0x0` and setting the memory of the read/write workspace array to zero.
3. Implement the functionality of the atomic region in a way that does not modify `IR` and that only reads memory from the workspace arrays, and only writes to registers or to the read-/write workspace array.
4. At the end of the atomic region, check `IR`. If it is set, then jump back to step 2, otherwise leave the atomic block and continue execution.

Thus the atomic block only completes if no interruption occurs during its execution, and the data in the read-only workspace array will be unchanged throughout an uninterrupted execution. Note, however, that an attacker could modify the workspace data between execution attempts so there is no guarantee that the atomic region will process the same input data on each execution attempt.

With simultaneous multithreading (SMT) disabled, an attacker cannot evict workspace data from the cache and force a read from the DIMMs without interrupting the process. Thus even if an attacker attempts to rollback memory in the DIMMs during execution of the atomic region, it will not be seen in the processing.

An attacker is capable of rolling back registers by interrupting the process, rolling back the SSA to an earlier version, then resuming the process. Note that the attacker cannot use this to clear `IR` since our handler will reset it after every interruption.

SEV-SNP Implementation. The TCB for SEV-SNP includes the operating system (OS), as attestation is at the VM level. To implement a AEX-Notify style handler on SEV-SNP, we can modify the trusted OS to handle APIC interrupts and carry out the steps described above.

5.14 Raft$^{\circlearrowleft}$ safety proofs

Lemma 5.4 (Fundamental Lemma)**.** If $\mathtt{Len}(RollbackServer) \leq s$, where s is the rollback tolerance parameter, then the intersection of any two quorums contains at least one non-rolled-back server.

Proof. A quorum is comprised of $\lfloor (m+s)/2 \rfloor + 1$ servers. Two quorums have a total of $> m+s$ servers, so they must overlap in more than s servers. At most s of these servers can be rolled back, so the intersection of these two quorums must contain at least one non-rolled-back server. □

Lemma 5.5. A server's $currentTerm$ monotonically increases if it is not rolled back in this transition:

$$\begin{array}{l}\forall i \in Server : \\ \quad \forall s : \neg \texttt{Rollback}(i, s) \implies \\ \quad currentTerm[i] \leq currentTerm'[i]\end{array}$$

Proof. This follows from the specification. □

Lemma 5.6. There is at most one leader per term:

$$\begin{array}{l}\forall e, f \in elections : \\ \quad e.eterm = f.eterm \implies e.eleader = f.eleader\end{array}$$

Proof sketch. This follows from Lemma 5.4. It takes votes from a quorum to become leader, voters may only vote once per term, and any two quorums overlap in a non-rolled-back voter.

Lemma 5.7. A non-rolled-back leader's log monotonically grows during its term:

$$\begin{array}{l}\forall e \in elections \\ \quad \land\ e.leader \notin RollbackServer \\ \quad \land\ currentTerm[e.leader] = e.term \implies \\ \quad\quad \forall \texttt{index} \in 1 \ldots \texttt{Len}(log[e.leader]) : \\ \quad\quad\quad log'[e.leader][index] = log[e.leader][index]\end{array}$$

Proof. The proof corresponds exactly to the proof of Lemma 3 in [234]. □

Lemma 5.8. Assume that the hash function used is a collision-resistant hash function [115]. Then, there exists a negligible function $\nu(\cdot)$ such that the hash of an $\langle index, term, value, hash \rangle$ tuple identifies a log prefix with probability $1 - \nu(\lambda)$:

$$\begin{array}{l}\forall l, m \in allLogs \\ \quad \langle index, term, value, hash \rangle \in l : \\ \quad\quad \langle index, term', value' hash \rangle \in m : \\ \quad\quad\quad \forall pindex \in 1 \ldots index : \\ \quad\quad\quad\quad l[pindex] = m[pindex]\end{array}$$

Proof sketch. Only leaders create entries, and they assign the new entries term numbers that will never be assigned again by other leaders (Lemma 5.6).

When followers accept `AppendEntriesRequest` from the leader, they check that the values of $hash$ match. The probability of a collision for some other $index'$, $term'$, i.e., the follower appends a different entry with the same hash to its log is $\nu(\lambda)$.

Proof. We prove this inductively on an upper bound for $index$. For $index \leq 1$ violating the property requires finding $\langle index, term, value, hash\rangle$, $\langle index, term', value' hash\rangle$ such that

$$\texttt{Hash}(index, term, value, 0) = \texttt{Hash}(index, term', value', 0)$$

Since the hash function is collision resistant this implies $term = term'$ and $value = value'$ with high probability, proving our base case.

Now assume that for some N the result is true whenever $index < N$. A server only appends $\langle index, term, value, hash\rangle$ to its log l if $\texttt{Hash}(index, term, value, l[index - 1].hash) = hash$. Hence if $\langle index, term, value, hash\rangle \in l$ and $\langle index, term', value', hash\rangle \in m$ then $\texttt{Hash}(index, term, value, l[index - 1].hash) = hash = \texttt{Hash}(index, term', value', m[index - 1].hash)$.

This is a negligible probability unless $term = term'$, $value = value'$, and $l[index - 1].hash = m[index - 1].hash$. Thus $l[index] = m[index]$ with high probability. Furthermore, since $l[index - 1].hash = m[index - 1].hash$ with high probability, the inductive hypothesis implies $\forall pindex \in 1 \ldots index : l[pindex] = m[pindex]$, completing the induction. □

Lemma 5.9. When a follower processes an `AppendEntriesRequest` and does not reject it, then after processing, part of its log is a prefix of the leader's log at the time the leader sent the `AppendEntriesRequest`:

$$\begin{array}{l}
\forall i, j \in Server, \forall m \in \text{DOMAIN } messages : \\
\quad \land\ \texttt{HandleAppendEntriesRequest}(\texttt{i}, \texttt{j}, \texttt{m}) \\
\quad \land\ \exists\, rsp \in \text{DOMAIN } messages : \\
\qquad \land\ \texttt{Reply}(rsp, m) \\
\qquad \land\ rsp.msuccess = \text{TRUE} \implies \\
\qquad\quad \forall index \in 1 \ldots m.mcommitIndex : \\
\qquad\quad\ \land\ log'[i][index] = m.mlog[index]
\end{array}$$

Proof sketch. The follower only appends $\langle index, term, value, hash\rangle$ if its hash chain is consistent with the follower's current log. Similarly the leader computed $hash$ in $\langle index, term, value, hash\rangle$ to be consistent with its own log. We can use Lemma 5.8 to $m.mlog$ and $log'[i]$ since both are in $allLogs$.

Proof. Since $rsp.msuccess = \text{TRUE}$ it follows that the intermediate expression $logOk$ in the definition of `HandleAppendEntriesRequest` evaluates to TRUE.

If no entries were added by this request then $m.mprevLogIndex \geq m.mcommitIndex$. Further, $logOk = \text{TRUE}$ implies that

$$m.mlog[m.mprevLogIndex].hash = log[i][m.mprevLogIndex].hash,$$

thus Lemma 5.8 implies $m.mlog$ matches $log[i]$ up to $m.mprevLogIndex > m.mprevIndex \geq m.mcommitIndex$.

If entries were added to $log[i]$, then $logOk = \text{TRUE}$ implies that the hash chain value of the added log value matches the hash chain value corresponding entry in $m.mlog$. Applying Lemma 5.8 now shows that $log'[i]$ is now a prefix of $m.mlog$ and the result follows.

□

Lemma 5.10. A server's $currentTerm$ is at least as large as the terms in its log:

$$\begin{array}{l}\forall i \in Server : \\ \quad \langle index, term, value, hash \rangle \in log[i] : \\ \qquad term \leq currentTerm[i]\end{array}$$

Proof sketch. Without rollbacks, prove by induction in Lemma 6 of [234]. A server can only be rolled back into a state where the inductive hypothesis is true.

Lemma 5.11. Servers never remove promised entries without rollback:

$$\begin{array}{l}\forall i \in Server \setminus RollbackServer : \\ \quad \wedge \langle index, term, value, hash \rangle \in log[i] \\ \quad \wedge step = s_0 \\ \quad \wedge index \leq promiseIndex[i] \implies \\ \qquad \forall s_1 \geq s_0 : \\ \qquad\quad \wedge step = s_1 \\ \qquad\quad \wedge \langle index, term, value, hash \rangle \in logs[i]\end{array}$$

Proof. This follows immediately from the specification, since the promise index increases monotonically and promised entries are not removed. □

Lemma 5.12. If an entry is not in a leader's log, then there is an earlier election for this leader and this term that does not have the entry in the election log.

$$\begin{array}{l}\wedge \texttt{isLeader}(leader) \\ \wedge \langle i, t, v, h \rangle \notin logs[leader] \implies \\ \quad \exists e \in elections : \\ \qquad \wedge e.estep \leq step \\ \qquad \wedge e.eterm = currentTerm[leader] \\ \qquad \wedge \langle i, t, v, h \rangle \notin e.elog\end{array}$$

Proof.

1. Assume

   $$\begin{array}{l}\wedge \texttt{isLeader}(leader) \\ \wedge \langle idx, t, v, h \rangle \notin logs[leader]\end{array}$$

2. Define

a) $goodSteps \triangleq \{s : \texttt{isLeader}(leader, s) \land \langle idx, t, v, h\rangle \notin states[s][leader].log \land states[s][leader].term = currentTerm[leader]\}$

This is the set of all steps (state transitions) within a term where the leader of that term does not have $\langle idx, t, v, h\rangle$ in its log.

b) $step1 \triangleq \min(goodSteps)$. This is well defined since by our assumption $step \in goodSteps$ so $goodSteps \neq \varnothing$. Furthermore $step1 \leq step$.

3. It follows that $\lor \neg\, isLeader(leader, step1 - 1) \lor states[step1 - 1][leader].term < currentTerm[leader] \lor \langle idx, t, v, h\rangle \in states[step1 - 1][leader].log$

In order for $step1$ to be the minimal $goodStep$, one of the above clauses must be true about $step1 - 1$ because $step1 - 1 \notin goodSteps$.

a) If $\langle idx, t, v, h\rangle \in states[step1 - 1][leader].log$ then the action that led to $step1$ removed $\langle idx, t, v, h\rangle$ from the log. Thus it was either a rollback or a `HandleAppendEntriesRequest`.

i. If $leader$ processed a `HandleAppendEntriesRequest` at this step then it was not a leader at $step1 - 1$ since leaders do not process these. Furthermore since it processed a `HandleAppendEntriesRequest` and not a `BecomeLeader` or rollback, it could not become leader at $step1$. This is a contradiction.

ii. If the action from $step1 - 1$ to $step1$ was a rollback to an earlier state at some $step0$ then we must have $step0 \in goodSteps \land step0 < step1$. This is a contradiction.

iii. Thus $\langle idx, t, v, h\rangle \notin states[step1 - 1][leader].log$.

b) If $states[step1 - 1][leader].term < currentTerm[leader]$ then the action that led to $step1$ was either a rollback to an earlier $goodStep$, which is impossible since $step1$ is the earliest $goodStep$, or an $\texttt{ElectionTimeout}(leader)$ which would imply that $\neg\, \texttt{isLeader}(leader, step1)$. This is a contradiction.

c) Thus it must be $\neg\, \texttt{isLeader}(leader, step1 - 1)$.

4. There are two actions that could allow $leader$ to become a leader at $step1$:

a) A rollback to an earlier $goodStep$, but this is impossible because $step1$ is the earliest $goodStep$.

b) $\texttt{BecomeLeader}(leader)$ could occur. This does not change the log and it adds an election e' to $elections$ with:

$$\begin{aligned}&\land\ e'.estep < step\\ &\land\ e'.eterm = currentTerm[leader]\\ &\land\ e'.log = states[step1 - 1][leader].log\end{aligned}$$

Since $\langle i, t, v, h\rangle \notin states[step1 - 1][leader].log$ it follows that $\langle i, t, v, h\rangle \notin e'.elog$. This proves the result.

□

Definition 5.1. An entry $\langle index, term, value, hash \rangle$ is **immediately committed** if it is acknowledged by a quorum (including the leader) during $term$ and all members of that quorum have the same value for $hash$.

$$
\begin{aligned}
&immediatelyCommitted \triangleq \{\langle index, term, value, hash \rangle \in anyLog : \\
&\quad \land\ anyLog \in allLogs \\
&\quad \land\ \exists leader \in Server, subquorum \in \text{SUBSET } Server : \\
&\qquad \land\ subquorum \cup \{leader\} \in Quorum \\
&\qquad \land\ \forall i \in subquorum : \\
&\qquad\quad \exists m \in messages: \\
&\qquad\qquad \land\ m.mtype = \texttt{AppendEntriesResponse} \\
&\qquad\qquad \land\ m.msource = i \\
&\qquad\qquad \land\ m.dest = leader \\
&\qquad\qquad \land\ m.term = term \\
&\qquad\qquad \land\ m.mPromiseIndex \geq index \\
&\qquad\qquad \land\ log[leader][m.mMatchIndex].hashChain = \\
&\qquad\qquad\quad m.mMatchHashChain \\
&\qquad\qquad \land\ \langle index, term, value, hash \rangle \in log[leader]\ \}
\end{aligned}
$$

Note that $\langle index, term, value, hash \rangle \in log[leader]$ enforces that $\langle index, term, value, hash \rangle$ is indeed in the log instead of some $\langle index, term, value, hash' \rangle$. Our definition of immediately committed differs from [234]. In particular, we introduce a promise index and a hash chain. We also prove committed under *live* terms, which we define next.

Definition 5.2. A **live term** is a term in which some log entry is immediately committed:

$$
\begin{aligned}
&liveTerms \triangleq \{term : \\
&\quad \exists \langle index, term, value, hash \rangle \in immediatelyCommitted\}
\end{aligned}
$$

Definition 5.3. An entry $\langle index, term, value, hash \rangle$ is **live committed at term** $term$ if it is present in every leader's log in live terms following $term$:

$$
\begin{aligned}
&liveCommitted(term) \triangleq \{\langle index, term, value, hash \rangle : \\
&\quad \forall election \in elections : \\
&\qquad \land\ election.eterm > term \\
&\qquad \land\ election.eterm \in liveTerms \implies \\
&\qquad\quad \langle index, term, value, hash \rangle \in election.elog\}
\end{aligned}
$$

Lemma 5.13. Immediately-committed entries are live committed:

$$
\begin{aligned}
&\forall \langle index, term, value, hash \rangle \in immediatelyCommitted : \\
&\ \langle index, term, value, hash \rangle \in liveCommitted(term)
\end{aligned}
$$

Proof.

1. Let $\langle index, term, value, hash \rangle$ be an entry that is immediately committed.
2. Define

$$\begin{aligned} &BadElections \triangleq \{\ election \in elections : \\ &\quad \land\ election.eterm > term \\ &\quad \land\ \langle index, term, value, hash \rangle \notin election.log\ \} \end{aligned}$$

3. Let $election$ be an element in $BadElections$ with a minimal $eterm$ field. If there is more than one election in the same term, choose the election with the minimal $estep$ field.
4. WTS $BadElections$ does not contain any elections e with $e.eterm \in liveTerms$.
5. Let $voter$ be any server that both votes in $election$, contains $\langle index, term, value, hash \rangle$ in its log during $term$, and has not been rolled back. Such a server must exist since:
   a) A quorum of servers voted in $election$ for it to succeed.
   b) A quorum contains $\langle index, term, value, hash \rangle$ in its log during $term$ since it is immediately committed. Because $m.mMatchHashChain$ must match across all servers in this quorum, all quorum members agree with the leader (and each other) w.h.p. at $index$ (Lemma 5.8).
   c) Any two quorums overlap in a server that has not been rolled back (Lemma 5.4).
6. Let $voterLog \triangleq election.evoterLog[voter]$, the voter's log at the time it cast its vote.
7. WTS $\langle index, term, value, hash \rangle \in voterLog$:
   a) $\langle index, term, value, hash \rangle$ was in the voter's log during $term$.
   b) The voter must have stored the entry in $term$ before voting in $election.term$ since:
      i. $election.eterm > term$
      ii. The voter rejects requests for $index$ with terms smaller than its current term, and its current term monotonically increases (Lemma 5.5).
   c) The voter couldn't have removed the entry before voting:
      C-1: No `AppendEntriesRequest` with $mterm \leq term$ removes the entry from the voter's log, since $currentTerm[voter] \geq term$ upon storing the entry (Lemma 5.10) and the voter does not remove entries from requests with terms $\leq currentTerm[voter]$.
      C-2: No `AppendEntriesRequest` with $mterm > term$ removes the entry from the voter's log, since:
         C-A: $mterm > election.eterm$: the voter would have been prevented in voting in $election.eterm$.
         C-B: $mterm = election.eterm$: In order for the leader to have sent an `AppendEntriesRequest` for $\langle index, term, value, hash' \rangle$, the leader did not have $\langle index, term, value, hash \rangle$ in its log, so there was an earlier election that did not have $\langle index, term, value, hash \rangle$ in its $elog$ (Lemma 5.12), which is a contradiction.

C-C: $mterm < election.eterm$: The leader of $mterm$ must have the entry, otherwise by Lemma 5.12 it has an earlier election that does not have the entry in its log. This contradicts the assumption that e is the minimal elelection in $BadElections$.

8. Since $voter$ voted during $election$:

$$\begin{array}{l} \lor\ \texttt{LastTerm}(election.elog) > \texttt{LastTerm}(voterLog) \\ \lor\ \land\ \texttt{LastTerm}(election.elog) = \texttt{LastTerm}(voterLog) \\ \quad \land\ \texttt{Len}(election.elog) \geq \texttt{Len}(voterLog) \end{array}$$

9. Case: $\texttt{LastTerm}(election.elog) = \texttt{LastTerm}(voterLog) \land \texttt{Len}(election.elog) \geq \texttt{Len}(voterLog)$:
   a) Let Q denote the quorum of servers that immediately committed $\langle index, term, value, hash \rangle$.
   b) Consider a live term $t > term$ with election e, $e.eterm = t$, and an entry $\langle i, t, v, h \rangle$ immediately committed by quorum Q'. We prove that $\langle index, term, value, hash \rangle \in e.elog$:
      i. By Lemma 5.4 there $\exists server \in Q \cap Q' : server \notin RollbackServer$.
      ii. Let s_0 denote the step at which $server$ created the `AppendEntriesResponse` involved in the commitment of $\langle index, term, value, hash \rangle$. Let s_1 denote the step at which $server$ created the `AppendEntriesResponse` involved in the commitment of $\langle i, t, v, h \rangle$.
      iii. Since $t > term$ Lemma 5.5 implies $s_1 > s_0$.
      iv. Since $server$ had $promiseIndex[server] \geq index$ at some step before s_0 and $server \notin RollbackServer$, Lemma 5.11 implies that $\langle index, term, value, hash \rangle \in log[server]$ at all steps after s_0.
      v. Thus when $server$ created its message immediately committing $\langle i, t, v, h \rangle$ at step s_1 it had $\langle index, term, value, hash \rangle$ in its log.
      vi. Since every member of quorum Q' had $\langle i, t, v, h \rangle$ in its log in term t, Lemma 5.8 implies that every member of quorum Q' had $\langle index, term, value, hash \rangle$ in its log in term t.
      vii. Since $e.eleader \in Q'$ we know that $\langle index, term, value, hash \rangle$ was in $logs[e.eleader]$ during term t.
      viii. Since $term < t$, $\langle index, term, value, hash \rangle$ could not be added during term t it follows that $\langle index, term, value, hash \rangle \in e.elog$.
10. Case: $\texttt{LastTerm}(election.elog) > \texttt{LastTerm}(voterLog)$:
   a) $\texttt{LastTerm}(voterLog) \geq term$ since $\langle index, term, value, hash \rangle \in voterLog$ and terms in non-rolled-back servers grow monotonically (Lemma 5.5).
   b) $election.eterm > \texttt{LastTerm}(election.elog)$ since servers increment their $currentTerm$ when starting an election and by Lemma 5.10 a server's current term is at least as large as the terms in its log.
   c) Let $prior$ be the last election with $prior.eterm = \texttt{LastTerm}(election.elog)$. Such an election must exist since $\texttt{LastTerm}(election.elog) > 0$ and a server must win an election

before creating an entry.

d) By transitivity we have $term \leq \texttt{LastTerm}(voterLog) < \texttt{LastTerm}(election.elog) = prior.eterm < election.eterm$.

e) $\texttt{LastTerm}(election.elog) = prior.eterm$ implies $\exists \langle i_0, prior.eterm, v_0, h_0 \rangle \in election.elog$.

f) Since $\langle index, term, value, hash \rangle \notin election.elog$, Lemma 5.8 implies that $\langle index, term, value, hash \rangle$ was not in the log of $prior.eleader$ when it created the `AppendEntriesRequest` that added $\langle i_0, prior.eterm, v_0, h_0 \rangle$.

g) Lemma 5.12 implies that there was an earlier election, $badElection$, with $badElection.eleader = prior.eleader$ and $badElection.eterm = prior.eterm$ such that $\langle index, term, value, hash \rangle \notin badElection.elog$.

h) Thus $badElection \in BadElections$ and is earlier than $election$, a contradiction.

□

Definition 5.4. An entry $\langle index, term, value, hash \rangle$ is **prefix committed at term** t if there is another entry that is *live committed at term* t following it in some log.

$$
\begin{aligned}
&prefixCommitted(term) \triangleq \{\langle index, term, value, hash \rangle \in anyLog : \\
&\quad \land\ anyLog \in allLogs \\
&\quad \land\ \exists \langle rindex, rterm, rvalue, rhash \rangle \in anyLog: \\
&\qquad \land\ index < rindex \\
&\qquad \land\ \langle rindex, rterm, rvalue, rhash \rangle \in liveCommitted(t)\}
\end{aligned}
$$

Lemma 5.14. Prefix committed entries are live committed in the same term.

Proof. The argument is identical to the proof of Lemma 9 in Appendix B [234], mutatis mutandis. □

Theorem 5.15. *Servers only apply entries that are committed in their current term:*

$$
\begin{aligned}
&\forall i \in Server : \\
&\quad \land\ commitIndex[i] \leq \texttt{Len}(log[i]) \\
&\quad \land\ \forall \langle index, term, value, hash \rangle \in log[i] : \\
&\qquad index \leq commitIndex[i] \implies \\
&\qquad\quad \langle \texttt{index}, \texttt{term}, \texttt{value}, \texttt{hash} \rangle \in \texttt{liveCommitted}(currentTerm[i])
\end{aligned}
$$

This is a restatement of Theorem 5.3.

Proof. The proof closely follows the proof of the State Machine Safety Property in [234]. We first note that for an infinite execution which has no *liveTerms* after *step*, all entries are trivially *liveCommitted* at *step* making the result trivial. So we may assume that there exists a live term after the current *step*.

We prove by induction on an execution.

1. Initial state: the property trivially holds for empty logs and $commitIndex[i] = 0$.

2. Inductive step: A rollback occurs:
   a) Once an entry is live committed at $currentTerm[i]$, all leaders of subsequent live terms will have the entry in their log.
   b) Thus the set of live-committed entries at $currentTerm[i]$ grows monotonically and the rollback cannot shrink this set.
   c) A rollback can only decrease $commitIndex[i]$, thus the inductive hypothesis implies that the invariant holds.
   d) *In the remainder of this proof we will now assume that the transition was not due to a rollback.*
3. Inductive step: The set of entries live committed at $currentTerm[i]$ changes:
   a) As shown above the set of committed entries at $currentTerm[i]$ grows monotonically.
   b) So no entry with $index \leq commitIndex[i]$ could be removed from $\texttt{committed}(currentTerm[i])$ in this step, and the inductive hypothesis remains true.
4. Inductive step: $commitIndex[i]$ changes:
   a) If $commitIndex[i]$ decreases, the inductive hypothesis suffices to show the invariant holds.
   b) When $commitIndex[i]$ increases, it covers entries present in i's log that are committed:
      i. Case: Follower completes accepting `AppendEntriesRequest` m:
         A. Upon processing m the follower's log is a prefix of a prior version of the leader's log $m.mlog$ by Lemma 5.9.
         B. Every entry through $commitIndex'[i]$ in $m.mlog$ is committed by the inductive hypothesis since they were committed in the leader's log when it sent the request.
      ii. Case: leader i processes an `AppendEntriesResponse`:
         A. If the leader sets a new $commitIndex$ then the conditions in the specification ensure that $logs[i][commitIndex'[i]] \in immediatelyCommitted$.
         B. Every entry in the leader's log up to $CommitIndex'[i]$ is prefix committed.
         C. Lemma 5.13 and Lemma 5.14 imply that all entries in the leader's log up to $commitIndex'[i]$ are live committed.
5. Inductive step: $currentTerm[i]$ changes:
   a) Since this is not a rollback, by Lemma 5.5 $currentTerm'[i] \geq currentTerm[i]$.
   b) $\texttt{liveCommitted}(currentTerm[i]) \subseteq \texttt{liveCommitted}(currentTerm'[i])$ by the definition of `liveCommitted`.
   c) Thus the inductive hypothesis suffices to show that the invariant holds.
6. Inductive step: logs change in one of the following ways:
   a) Case: A leader adds one entry due to `ClientRequest`:

i. Newly created entries are not marked committed, so the invariant holds.

b) Case: a follower removes one entry due to `AppendEntriesRequest` m:

i. Assume that $\langle index, term, value, hash \rangle$ was removed from $logs[i]$.

ii. By Lemma 5.11 and the fact that this transition is not a rollback we conclude that $index > promiseIndex[i]$.

iii. Since $promiseIndex[i] \geq commitIndex[i]$ it follows that $index > commitIndex[i]$.

iv. Hence the entry was not committed and the invariant holds.

c) Case: a follower adds one entry due to `AppendEntriesRequest` m:

i. Case: the new entry is not marked committed on the follower: The inductive hypothesis suffices to show the invariant holds.

ii. Case: the new entry is marked committed on the follower: $commitIndex[i]$ must increase, which was handled above.

□

5.15 TLA+ specification of Raft↺

MODULE *Raft*↺

Based on is the formal specification for the Raft consensus algorithm
(Diego Ongaro, 2014) which is licensed under the Creative Commons Attribution-4.0
International License https://creativecommons.org/licenses/by/4.0/

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*, *Randomization*

The set of server IDs
CONSTANTS *Server*

The set of IDs of servers that are rolled back
CONSTANTS *RollbackServer*

Server states.
CONSTANTS *Follower*, *Candidate*, *Leader*

A reserved value.
CONSTANTS *Nil*

Message types:
CONSTANTS *RequestVoteRequest*, *RequestVoteResponse*,
AppendEntriesRequest, *AppendEntriesResponse*

Maximum number of client requests
CONSTANTS *MaxClientRequests*

CONSTANTS *MaxSteps*

CONSTANTS *RollbackTolerance*

Global variables

A bag of records representing requests and responses sent from one server
to another. TLAPS doesn't support the Bags module, so this is a function
mapping Message to Nat.
VARIABLE *messages*

A history variable used in the proof. This would not be present in an
implementation.
Keeps track of successful elections, including the initial logs of the

leader and voters' logs. Set of functions containing various things about
successful elections (see BecomeLeader).
VARIABLE $elections$

A history variable used in the proof. This would not be present in an
implementation.
Keeps track of every log ever in the system (set of logs).
VARIABLE $allLogs$

a step counter used to model Rollback
VARIABLE $step$

a map from Server to a sequence of server states - one for each step.
VARIABLE $serverStates$

A hash function used to compute a hash chain
VARIABLE $hash$

The following variables are all per server (functions with domain Server).

The server's term number.
VARIABLE $currentTerm$
The server's state (Follower, Candidate, or Leader).
VARIABLE $state$
The candidate the server voted for in its current term, or
Nil if it hasn't voted for any.
VARIABLE $votedFor$
$serverVars \triangleq \langle currentTerm, state, votedFor \rangle$

The set of requests that can go into the log
VARIABLE $clientRequests$

A Sequence of log entries. The index into this sequence is the index of the
log entry. Unfortunately, the Sequence module defines Head(s) as the entry
with index 1, so be careful not to use that!
VARIABLE log
The latest entry that each follower has promised the leader to commit.
This is used to calculate commitIndex on the leader.
VARIABLE $promiseIndex$
The index of the latest entry in the log the state machine may apply.
VARIABLE $commitIndex$
VARIABLE $promisedLog$
VARIABLE $promisedLogDecrease$
The index that gets committed

VARIABLE $committedLog$

Does the commited Index decrease

VARIABLE $committedLogDecrease$

$logVars \triangleq \langle log, commitIndex, promiseIndex, clientRequests, committedLog, committedLogDecrease, promisedLog, promisedLogDecrease \rangle$

The following variables are used only on candidates:

The set of servers from which the candidate has received a RequestVote response in its currentTerm.

VARIABLE $votesSent$

The set of servers from which the candidate has received a vote in its currentTerm.

VARIABLE $votesGranted$

A history variable used in the proof. This would not be present in an implementation.

Function from each server that voted for this candidate in its currentTerm to that voter's log.

VARIABLE $voterLog$

$candidateVars \triangleq \langle votesSent, votesGranted, voterLog \rangle$

The following variables are used only on leaders:

The next entry to send to each follower.

VARIABLE $nextIndex$

The latest entry that each follower has acknowledged is the same as the leader's. This is used to calculate promiseIndex on the leader.

VARIABLE $matchIndex$

VARIABLE $ackedPromiseIndex$

$leaderVars \triangleq \langle nextIndex, matchIndex, ackedPromiseIndex, elections \rangle$

End of per server variables.

All variables; used for stuttering (asserting state hasn't changed).

$vars \triangleq \langle messages, allLogs, serverVars, candidateVars, leaderVars, logVars, hash, serverStates, step \rangle$

Hash function setup

$BitString256 \triangleq [1 \,..\, 256 \to \text{BOOLEAN}\,]$

Helpers

The set of all quorums. This just calculates simple majorities, but the only important property is that every quorum overlaps with every other.

$Quorum \triangleq \{i \in \text{SUBSET}\ (Server) : Cardinality(i) * 2 > RollbackTolerance + Cardinality(Server)\}$

The term of the last entry in a log, or 0 if the log is empty.

$$LastTerm(xlog) \triangleq \text{IF } Len(xlog) = 0 \text{ THEN } 0 \text{ ELSE } xlog[Len(xlog)].term$$

Helper for Send and Reply. Given a message m and bag of messages, return a new bag of messages with one more m in it.

$$WithMessage(m,\ msgs) \triangleq$$
$$\text{IF } m \in \text{DOMAIN } msgs \text{ THEN}$$
$$[msgs \text{ EXCEPT } ![m] = \text{IF } msgs[m] < 2 \text{ THEN } msgs[m] + 1 \text{ ELSE } 2]$$
$$\text{ELSE}$$
$$msgs \mathbin{@@} (m :> 1)$$

Helper for Discard and Reply. Given a message m and bag of messages, return a new bag of messages with one less m in it.

$$WithoutMessage(m,\ msgs) \triangleq$$
$$\text{IF } m \in \text{DOMAIN } msgs \text{ THEN}$$
$$[msgs \text{ EXCEPT } ![m] = \text{IF } msgs[m] > 0 \text{ THEN } msgs[m] - 1 \text{ ELSE } 0]$$
$$\text{ELSE}$$
$$msgs$$

$$ValidMessage(msgs) \triangleq$$
$$\{m \in \text{DOMAIN } messages : msgs[m] > 0\}$$

$$SingleMessage(msgs) \triangleq$$
$$\{m \in \text{DOMAIN } messages : msgs[m] = 1\}$$

Add a message to the bag of messages.

$$Send(m) \triangleq messages' = WithMessage(m,\ messages)$$

Remove a message from the bag of messages. Used when a server is done processing a message.

$$Discard(m) \triangleq messages' = WithoutMessage(m,\ messages)$$

Combination of Send and Discard

$$Reply(response,\ request) \triangleq$$
$$messages' = WithoutMessage(request,\ WithMessage(response,\ messages))$$

Return the minimum value from a set, or undefined if the set is empty.

$$Min(s) \triangleq \text{CHOOSE } x \in s : \mathsf{Adv}\ y \in s : x \leq y$$

Return the maximum value from a set, or undefined if the set is empty.

$$Max(s) \triangleq \text{CHOOSE } x \in s : \mathsf{Adv}\ y \in s : x \geq y$$

The current state of server i

$$CurrentFollowerState(i) \triangleq [$$
$$sslog \mapsto log[i],$$
$$sscurrentTerm \mapsto currentTerm[i],$$
$$ssvotedFor \mapsto votedFor[i],$$

$ssstate \mapsto state[i],$
$sspromiseIndex \mapsto promiseIndex[i],$
$sscommitIndex \mapsto commitIndex[i]]$
$CurrentLeaderState(i) \triangleq [$
$sslog \mapsto log[i],$
$sscurrentTerm \mapsto currentTerm[i],$
$ssvotedFor \mapsto votedFor[i],$
$ssstate \mapsto state[i],$
$sspromiseIndex \mapsto promiseIndex[i],$
$sscommitIndex \mapsto commitIndex[i],$
$ssnextIndex \mapsto nextIndex[i],$
$ssmatchIndex \mapsto matchIndex[i],$
$ssackedPromiseIndex \mapsto ackedPromiseIndex[i]]$
$CurrentCandidateState(i) \triangleq [$
$sslog \mapsto log[i],$
$sscurrentTerm \mapsto currentTerm[i],$
$ssvotedFor \mapsto votedFor[i],$
$ssstate \mapsto state[i],$
$sspromiseIndex \mapsto promiseIndex[i],$
$sscommitIndex \mapsto commitIndex[i],$
$ssvotesSent \mapsto votesSent[i],$
$ssvotesGranted \mapsto votesGranted[i]]$

$CurrentState(i) \triangleq$ IF $state[i] = Follower$ THEN $CurrentFollowerState(i)$
ELSE IF $state[i] = Candidate$ THEN $CurrentCandidateState(i)$
ELSE $CurrentLeaderState(i)$

$RecordStates \triangleq$ LET $currentState \triangleq [i \in Server \mapsto CurrentState(i)]$
IN $serverStates' = [serverStates$ EXCEPT $![step] = currentState]$

Define initial values for all variables

$InitHistoryVars \triangleq \land elections = \{\}$
$\land allLogs = \{\}$
$\land voterLog = [i \in Server \mapsto [j \in \{\} \mapsto \langle\rangle]]$
$\land serverStates = [s \in 0 \,..\, MaxSteps \mapsto [i \in Server \mapsto \langle\rangle]]$
$InitServerVars \triangleq \land currentTerm = [i \in Server \mapsto 1]$
$\land state = [i \in Server \mapsto Follower]$
$\land votedFor = [i \in Server \mapsto Nil]$
$InitCandidateVars \triangleq \land votesSent = [i \in Server \mapsto$ FALSE$]$
$\land votesGranted = [i \in Server \mapsto \{\}]$

The values nextIndex[i][i] and matchIndex[i][i] are never read, since the

leader does not send itself messages. It's still easier to include these
in the functions.

$$\begin{aligned}
&InitLeaderVars \triangleq \ \land nextIndex \ = [i \in Server \mapsto [j \in Server \mapsto 1]] \\
&\land matchIndex \ = [i \in Server \mapsto [j \in Server \mapsto 0]] \\
&\land ackedPromiseIndex = [i \in Server \mapsto [j \in Server \mapsto 0]]
\end{aligned}$$

$$\begin{aligned}
&InitLogVars \triangleq \ \land log \ = [i \in Server \mapsto \langle\rangle] \\
&\land commitIndex \ = [i \in Server \mapsto 0] \\
&\land promiseIndex \ = [i \in Server \mapsto 0] \\
&\land clientRequests = 1 \\
&\land committedLog = \langle\rangle \\
&\land committedLogDecrease = \textsc{false} \\
&\land promisedLog = \langle\rangle \\
&\land promisedLogDecrease = \textsc{false}
\end{aligned}$$

$$\begin{aligned}
&RollbackServersAreServers \triangleq \\
&\quad \land IsFiniteSet(RollbackServer) \\
&\quad \land RollbackServer \subseteq Server
\end{aligned}$$

$$\begin{aligned}
&Init \triangleq \ \land messages = [m \in \{\} \mapsto 0] \\
&\land InitHistoryVars \\
&\land InitServerVars \\
&\land InitCandidateVars \\
&\land InitLeaderVars \\
&\land InitLogVars \\
&\land step = 0 \\
&\land hash = [x \in \{\} \mapsto Nil] \\
&\land RollbackServersAreServers
\end{aligned}$$

Define state transitions

Server i times out and starts a new election.

$$\begin{aligned}
&Timeout(i) \triangleq \ \land state[i] \in \{Follower, \ Candidate\} \\
&\land state' = [state \ \textsc{except} \ ![i] = Candidate] \\
&\land currentTerm' = [currentTerm \ \textsc{except} \ ![i] = currentTerm[i] + 1]
\end{aligned}$$

Most implementations would probably just set the local vote
atomically, but messaging localhost for it is weaker.

$$\begin{aligned}
&\land votedFor' = [votedFor \ \textsc{except} \ ![i] = Nil] \\
&\land votesSent' = [votesSent \ \textsc{except} \ ![i] = \textsc{false}] \\
&\land votesGranted' \ = [votesGranted \ \textsc{except} \ ![i] = \{\}] \\
&\land voterLog' \ = [voterLog \ \textsc{except} \ ![i] = [j \in \{\} \mapsto \langle\rangle]] \\
&\land \textsc{unchanged} \ \langle messages, \ leaderVars, \ logVars, \ hash\rangle
\end{aligned}$$

Rollback server i to its state at step s

$$
\begin{array}{l}
Rollback(i,\ s) \triangleq \text{LET } restoreState \triangleq serverStates[s][i] \\
\quad \text{IN} \quad \land\ i \in RollbackServer \\
\quad \land\ log' = [log \text{ EXCEPT } ![i] = restoreState.sslog] \\
\quad \land\ currentTerm' = [currentTerm \text{ EXCEPT } ![i] = restoreState.sscurrentTerm] \\
\quad \land\ votedFor' = [votedFor \text{ EXCEPT } ![i] = restoreState.ssvotedFor] \\
\quad \land\ state' = [state \text{ EXCEPT } ![i] = restoreState.ssstate] \\
\quad \land\ promiseIndex' = [promiseIndex \text{ EXCEPT } ![i] = restoreState.sspromiseIndex] \\
\quad \land\ commitIndex' = [commitIndex \text{ EXCEPT } ![i] = restoreState.sscommitIndex] \\
\quad \land\ \lor\ \land\ restoreState.ssstate = Follower \\
\quad \lor\ \land\ restoreState.ssstate = Candidate \\
\quad \land\ votesSent' = [votesSent \text{ EXCEPT } ![i] = restoreState.ssvotesSent] \\
\quad \land\ votesGranted' = [votesGranted \text{ EXCEPT } ![i] = restoreState.ssvotesGranted] \\
\quad \lor\ \land\ restoreState.ssstate = Leader \\
\quad \land\ nextIndex' = [nextIndex \text{ EXCEPT } ![i] = restoreState.ssnextIndex] \\
\quad \land\ matchIndex' = [matchIndex \text{ EXCEPT } ![i] = restoreState.ssmatchIndex] \\
\quad \land\ ackedPromiseIndex' = [ackedPromiseIndex \text{ EXCEPT } ![i] = restoreState.ssackedPromiseIndex] \\
\quad \land\ \text{UNCHANGED } \langle messages,\ elections,\ clientRequests,\ committedLog,\ committedLogDecrease, \\
\qquad\qquad promisedLog,\ promisedLogDecrease,\ ackedPromiseIndex,\ matchIndex,\ nextIndex, \\
\qquad\qquad voterLog,\ votesGranted,\ votesSent,\ hash \rangle
\end{array}
$$

Candidate i sends j a RequestVote request.

$$
\begin{array}{l}
RequestVote(i,\ j) \triangleq \\
\quad \land\ state[i] = Candidate \\
\quad \land\ Send([mtype \mapsto RequestVoteRequest, \\
\quad mterm \mapsto currentTerm[i], \\
\quad mlastLogTerm \mapsto LastTerm(log[i]), \\
\quad mlastLogIndex \mapsto Len(log[i]), \\
\quad msource \mapsto i, \\
\quad mdest \mapsto j]) \\
\quad \land\ \text{UNCHANGED } \langle serverVars,\ votesGranted,\ voterLog,\ leaderVars,\ logVars,\ votesSent,\ hash \rangle
\end{array}
$$

Leader i sends j an AppendEntries request containing up to 1 entry.
While implementations may want to send more than 1 at a time, this spec uses
just 1 because it minimizes atomic regions without loss of generality.

$$
\begin{array}{l}
AppendEntries(i,\ j) \triangleq \\
\quad \land\ i \neq j \\
\quad \land\ state[i] = Leader \\
\quad \land\ \text{LET } prevLogIndex \triangleq nextIndex[i][j] - 1 \\
\quad prevLogTerm \triangleq \text{IF } prevLogIndex > 0 \text{ THEN} \\
\quad\ log[i][prevLogIndex].term \\
\quad \text{ELSE}
\end{array}
$$

```
        0
    prevLogHash ≜ IF prevLogIndex > 0 THEN
     log[i][prevLogIndex].hashChain
    ELSE
        0
    Send up to 1 entry, constrained by the end of the log.
    lastEntry ≜ Min({Len(log[i]), nextIndex[i][j]})
    entries ≜ SubSeq(log[i], nextIndex[i][j], lastEntry)
    IN Send([mtype          ↦ AppendEntriesRequest,
    mterm          ↦ currentTerm[i],
    mprevLogIndex          ↦ prevLogIndex,
    mprevLogTerm          ↦ prevLogTerm,
    mprevLogHash          ↦ prevLogHash,
    mentries          ↦ entries,
    mlog is used as a history variable for the proof.
    It would not exist in a real implementation.
    mlog          ↦ log[i],
    mcommitIndex          ↦ Min({commitIndex[i], lastEntry}),
    mpromiseIndex          ↦ Min({promiseIndex[i], lastEntry}),
    msource          ↦ i,
    mdest          ↦ j])
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars, hash⟩

Candidate i transitions to leader.
BecomeLeader(i) ≜
    ∧ state[i] = Candidate
    ∧ votesGranted[i] ∈ Quorum
    ∧ state'      = [state EXCEPT ![i] = Leader]
    ∧ nextIndex'      = [nextIndex EXCEPT ![i] =
          [j ∈ Server ↦ Len(log[i]) + 1]]
    ∧ matchIndex'      = [matchIndex EXCEPT ![i] =
          [j ∈ Server ↦ 0]]
    ∧ ackedPromiseIndex' = [ackedPromiseIndex EXCEPT ![i] =
    [j ∈ Server ↦ 0]]
    ∧ elections'  = elections ∪
        {[eterm     ↦ currentTerm[i],
        eleader     ↦ i,
        elog     ↦ log[i],
        evotes     ↦ votesGranted[i],
        evoterLog     ↦ voterLog[i],
        estep     ↦ step]}
    ∧ UNCHANGED ⟨messages, currentTerm, votedFor, candidateVars, logVars, hash⟩
```

```
Leader i receives a client request to add v to the log.
ClientRequest(i) ≜
    ∧ state[i] = Leader
    ∧ clientRequests < MaxClientRequests
    ∧ LET index ≜ Len(log[i])
      hashInput ≜ [hiindex ↦ index, hiterm ↦ currentTerm[i], hivalue ↦ clientRequests,
                   hilastHash ↦ log[i][Len(log[i])]]
      hashValue ≜ IF [hiindex ↦ index, hiterm ↦ currentTerm[i], hivalue ↦ clientRequests,
                      hilastHash ↦ log[i][Len(log[i])]] ∈ DOMAIN hash THEN
                    hash[[hiindex ↦ index, hiterm ↦ currentTerm[i], hivalue ↦ clientRequests,
                          hilastHash ↦ log[i][Len(log[i])]]]
                  ELSE
                    RandomElement(BitString256)
      entry ≜ [term ↦ currentTerm[i],
               hashChain ↦ hash[hashInput],
               value ↦ clientRequests]
      newLog ≜ Append(log[i], entry)
    IN ∧ log' = [log EXCEPT ![i] = newLog]
       Make sure that each request is unique, reduce state space to be explored
       ∧ clientRequests' = clientRequests + 1
       ∧ hash' = [hash EXCEPT ![hashInput] = hashValue]
    ∧ UNCHANGED ⟨messages, serverVars, candidateVars, leaderVars, commitIndex,
                 promiseIndex, committedLog, committedLogDecrease, promisedLog, promisedLogDecrease⟩

Leader i advances its promiseIndex.
This is done as a separate step from handling AppendEntries responses,
in part to minimize atomic regions, and in part so that leaders of
single-server clusters are able to mark entries committed.
AdvancePromiseIndex(i) ≜
    ∧ state[i] = Leader
    ∧ LET The set of servers that agree up through index.
      Agree(index) ≜ {i} ∪ {k ∈ Server :
        matchIndex[i][k] ≥ index}
      The maximum indexes for which a quorum agrees
      agreeIndexes ≜ {index ∈ 1 .. Len(log[i]) :
          Agree(index) ∈ Quorum}
      New value for commitIndex'[i]
      newPromiseIndex ≜
        IF ∧ agreeIndexes ≠ {}
           ∧ log[i][Max(agreeIndexes)].term = currentTerm[i]
        THEN
           Max(agreeIndexes ∪ {promiseIndex[i]})
```

```
        ELSE
            promiseIndex[i]
    newPromisedLog ≜
        IF newPromiseIndex > 1 THEN
            [j ∈ 1 .. newPromiseIndex ↦ log[i][j]]
        ELSE
            ⟨⟩
    IN  ∧ promiseIndex' = [promiseIndex EXCEPT ![i] = newPromiseIndex]
        ∧ promisedLogDecrease' = ∨ (newPromiseIndex < Len(promisedLog))
          ∨ ∃ j ∈ 1 .. Len(promisedLog) : promisedLog[j] ≠ newPromisedLog[j]
        ∧ promisedLog' = newPromisedLog
        ∧ UNCHANGED ⟨messages, serverVars, candidateVars, leaderVars, log, clientRequests,
                     commitIndex, committedLog, committedLogDecrease, hash⟩

Leader i advances its commitIndex.
This is done as a separate step from handling AppendEntries responses,
in part to minimize atomic regions, and in part so that leaders of
single-server clusters are able to mark entries committed.
AdvanceCommitIndex(i) ≜
    ∧ state[i] = Leader
    ∧ LET The set of servers that agree up through index.
      Agree(index) ≜ {i} ∪ {k ∈ Server :
        ackedPromiseIndex[i][k] ≥ index}
        The maximum indexes for which a quorum agrees
      agreeIndexes ≜ {index ∈ 1 .. Len(log[i]) :
          Agree(index) ∈ Quorum}
        New value for commitIndex'[i]
      newCommitIndex ≜
        IF ∧ agreeIndexes ≠ {}
           ∧ log[i][Max(agreeIndexes)].term = currentTerm[i]
        THEN
            Max(agreeIndexes)
        ELSE
            commitIndex[i]
      newCommittedLog ≜
        IF newCommitIndex > 1 THEN
            [j ∈ 1 .. newCommitIndex ↦ log[i][j]]
        ELSE
            ⟨⟩
    IN  ∧ commitIndex' = [commitIndex EXCEPT ![i] = newCommitIndex]
        ∧ committedLogDecrease' = ∨ (newCommitIndex < Len(committedLog))
```

$\lor \exists j \in 1 \,..\, Len(committedLog) : committedLog[j] \neq newCommittedLog[j]$
$\land committedLog' = newCommittedLog$
$\land$ UNCHANGED $\langle messages,\ serverVars,\ candidateVars,\ leaderVars,\ log,\ clientRequests \rangle$
$\land$ UNCHANGED $\langle promiseIndex,\ promisedLog,\ promisedLogDecrease,\ hash \rangle$

Message handlers
i = recipient, j = sender, m = message

Server i receives a RequestVote request from server j with
m.mterm <= currentTerm[i].

$HandleRequestVoteRequest(i,\ j,\ m) \triangleq$
LET $logOk \triangleq \lor m.mlastLogTerm > LastTerm(log[i])$
$\lor \land m.mlastLogTerm = LastTerm(log[i])$
$\land m.mlastLogIndex \geq Len(log[i])$
$grant \triangleq \land m.mterm = currentTerm[i]$
$\land logOk$
$\land votedFor[i] \in \{Nil,\ j\}$
IN $\land m.mterm \leq currentTerm[i]$
$\land \lor grant \land votedFor' = [votedFor$ EXCEPT $![i] = j]$
$\lor \neg grant \land$ UNCHANGED $votedFor$
$\land Reply([mtype \mapsto RequestVoteResponse,$
$mterm \mapsto currentTerm[i],$
$mvoteGranted \mapsto grant,$
mlog is used just for the 'elections' history variable for
the proof. It would not exist in a real implementation.
$mlog \mapsto log[i],$
$msource \mapsto i,$
$mdest \mapsto j],$
$m)$
$\land$ UNCHANGED $\langle state,\ currentTerm,\ candidateVars,\ leaderVars,\ logVars,\ hash \rangle$

Server i receives a RequestVote response from server j with
m.mterm = currentTerm[i].
$HandleRequestVoteResponse(i,\ j,\ m) \triangleq$
This tallies votes even when the current state is not Candidate, but
they won't be looked at, so it doesn't matter.
$\land m.mterm = currentTerm[i]$
$\land \lor \land m.mvoteGranted$
$\land votesGranted' = [votesGranted$ EXCEPT $![i] =$
$votesGranted[i] \cup \{j\}]$
$\land voterLog' = [voterLog$ EXCEPT $![i] =$

$$
\begin{array}{l}
\quad voterLog[i] \mathbin{@@} (j :> m.mlog)] \\
\land \textsc{unchanged}\ \langle votesSent \rangle \\
\lor\ \land \neg m.mvoteGranted \\
\land \textsc{unchanged}\ \langle votesSent,\ votesGranted,\ voterLog \rangle \\
\land Discard(m) \\
\land \textsc{unchanged}\ \langle serverVars,\ votedFor,\ leaderVars,\ logVars,\ hash \rangle
\end{array}
$$

Server i receives an AppendEntries request from server j with
m.mterm <= currentTerm[i]. This just handles m.entries of length 0 or 1, but
implementations could safely accept more by treating them the same as
multiple independent requests of 1 entry.

$$
\begin{array}{l}
HandleAppendEntriesRequest(i,\ j,\ m) \triangleq \\
\quad \textsc{let}\ hashInput \triangleq [hiindex \mapsto m.mprevLogIndex + 1, \\
\quad hiterm \mapsto m.mentries[1].term, \\
\quad hivalue \mapsto m.mentries[1].value, \\
\quad hilastHash \mapsto log[i][m.mprevLogIndex].hashChain] \\
\qquad hashValue \triangleq \textsc{if}\ hashInput \in \textsc{domain}\ hash\ \textsc{then} \\
\qquad\ hash[hashInput] \\
\qquad \textsc{else} \\
\qquad\quad RandomElement(BitString256) \\
\qquad logOk \triangleq\ \lor m.mprevLogIndex = 0 \\
\qquad\ \lor\ \land m.mprevLogIndex > 0 \\
\qquad\ \land m.mprevLogIndex \leq Len(log[i]) \\
\qquad\ \land m.mprevLogTerm = log[i][m.mprevLogIndex].term \\
\qquad\ \land m.mprevLogHash = log[i][m.mprevLogIndex].hashChain \\
\qquad\ \land\ \lor\ \land Len(m.mentries) = 0 \\
\qquad\ \land \textsc{unchanged}\ hash \\
\qquad\ \lor\ \land m.mprevLogIndex < Len(log[i]) \\
\qquad\ \land \textsc{unchanged}\ hash \\
\qquad\ \land\ \lor m.mentries[1].hashChain = log[i][m.mprevLogIndex + 1].hashChain \\
\qquad\ \lor\ \text{there's a conflict on a promised entry} \\
\qquad\ \land Len(m.mentries) > 0 \\
\qquad\ \land log[i][m.mprevLogIndex + 1].term \neq m.mentries[1].term \\
\qquad\ \land promiseIndex[i] = Len(log[i]) \\
\qquad\ \lor\ \land m.mprevLogIndex = Len(log[i]) \\
\qquad\ \land m.mentries[1].hashChain = hashValue \\
\qquad\ \land hash' = [hash\ \textsc{except}\ ![hashInput] = hashValue] \\
\quad \textsc{in}\ \ \land m.mterm \leq currentTerm[i] \\
\quad\ \land\ \lor\ \land\ \text{reject request} \\
\quad\ \lor m.mterm < currentTerm[i] \\
\quad\ \lor\ \land m.mterm = currentTerm[i] \\
\quad\ \land state[i] = Follower
\end{array}
$$

$\land \neg logOk$
$\land Reply([mtype \mapsto AppendEntriesResponse,$
$mterm \mapsto currentTerm[i],$
$msuccess \mapsto \text{FALSE},$
$mackedPromiseIndex \mapsto 0,$
$mmatchIndex \mapsto 0,$
$msource \mapsto i,$
$mdest \mapsto j],$
$m)$
$\land \text{UNCHANGED } \langle serverVars, logVars \rangle$
$\lor$ return to follower state
$\land m.mterm = currentTerm[i]$
$\land state[i] = Candidate$
$\land state' = [state \text{ EXCEPT } ![i] = Follower]$
$\land \text{UNCHANGED } \langle currentTerm, votedFor, logVars, messages \rangle$
$\lor$ accept request
$\land m.mterm = currentTerm[i]$
$\land state[i] = Follower$
$\land logOk$
$\land \text{LET } index \triangleq m.mprevLogIndex + 1$
IN $\lor$ already done with request
$\land \lor m.mentries = \langle\rangle$
$\lor \land m.mentries \neq \langle\rangle$
$\land Len(log[i]) \geq index$
$\land log[i][index].term = m.mentries[1].term$
This could make our commitIndex decrease (for
example if we process an old, duplicated request),
but that doesn't really affect anything.
$\land commitIndex' = [commitIndex \text{ EXCEPT } ![i] =$
$m.mcommitIndex]$
$\land promiseIndex' = [promiseIndex \text{ EXCEPT } ![i] =$
$Max(\{m.mpromiseIndex, promiseIndex[i]\})]$
$\land Reply([mtype \mapsto AppendEntriesResponse,$
$mterm \mapsto currentTerm[i],$
$msuccess \mapsto \text{TRUE},$
$mmatchIndex \mapsto m.mprevLogIndex +$
$Len(m.mentries),$
$mmatchHash \mapsto log[i][m.mprevLogIndex + Len(m.mentries)].hashChain,$
$mpromiseIndex \mapsto m.mpromiseIndex,$
$msource \mapsto i,$
$mdest \mapsto j],$
$m)$

$\land$ UNCHANGED $\langle serverVars,\ log,\ clientRequests,\ committedLog,\ promisedLog,$
$committedLogDecrease,\ promisedLogDecrease\rangle$
$\lor$ conflict: remove 1 entry
$\land\ m.mentries \neq \langle\rangle$
$\land\ Len(log[i]) \geq index$
$\land\ log[i][index].term \neq m.mentries[1].term$
$\land\ promiseIndex[i] < Len(log[i])$
$\land$ LET $new \triangleq [index2 \in 1\ ..\ (Len(log[i]) - 1) \mapsto$
$log[i][index2]]$
IN $log' = [log$ EXCEPT $![i] = new]$
$\land$ UNCHANGED $\langle serverVars,\ commitIndex,\ promiseIndex,\ messages,\ clientRequests,$
$commitedLog,\ committedLogDecrease\rangle$
$\land$ UNCHANGED $\langle promisedLog,\ promisedLogDecrease\rangle$
$\lor$ no conflict: append entry
$\land\ m.mentries \neq \langle\rangle$
$\land\ Len(log[i]) = m.mprevLogIndex$
$\land\ log' = [log$ EXCEPT $![i] =$
$Append(log[i],\ m.mentries[1])]$
$\land$ UNCHANGED $\langle serverVars,\ commitIndex,\ promiseIndex,\ messages,\ clientRequests,$
$committedLog,\ committedLogDecrease\rangle$
$\land$ UNCHANGED $\langle promisedLog,\ promisedLogDecrease\rangle$
$\land$ UNCHANGED $\langle candidateVars,\ leaderVars\rangle$

Server i receives an AppendEntries response from server j with
m.mterm = currentTerm[i].
$HandleAppendEntriesResponse(i,\ j,\ m) \triangleq$
$\land\ m.mterm = currentTerm[i]$
$\land\ \lor\ \land\ m.msuccess$ successful
$\land\ m.mmatchHash = log[i][m.mmatchIndex].hashChain$
$\land\ nextIndex' = [nextIndex$ EXCEPT $![i][j] = m.mmatchIndex + 1]$
$\land\ matchIndex' = [matchIndex$ EXCEPT $![i][j] = m.mmatchIndex]$
$\land\ ackedPromiseIndex' = [ackedPromiseIndex$ EXCEPT $![i][j] = Max(\{m.mpromiseIndex,\ @\})]$
$\lor\ \land \neg m.msuccess$ not successful
$\land\ nextIndex' = [nextIndex$ EXCEPT $![i][j] =$
$Max(\{nextIndex[i][j] - 1,\ 1\})]$
$\land$ UNCHANGED $\langle matchIndex\rangle$
$\land\ Discard(m)$
$\land$ UNCHANGED $\langle serverVars,\ candidateVars,\ logVars,\ elections,\ hash\rangle$

Any RPC with a newer term causes the recipient to advance its term first.
$UpdateTerm(i,\ j,\ m) \triangleq$
$\land\ m.mterm > currentTerm[i]$

$\land\ currentTerm' = [currentTerm \text{ EXCEPT } ![i] = m.mterm]$
$\land\ state' = [state \text{ EXCEPT } ![i] = Follower]$
$\land\ votedFor' = [votedFor \text{ EXCEPT } ![i] = Nil]$
messages is unchanged so m can be processed further.
$\land$ UNCHANGED $\langle messages, candidateVars, leaderVars, logVars, hash\rangle$

Responses with stale terms are ignored.
$DropStaleResponse(i, j, m) \triangleq$
$\land\ m.mterm < currentTerm[i]$
$\land\ Discard(m)$
$\land$ UNCHANGED $\langle serverVars, candidateVars, leaderVars, logVars, hash\rangle$

Receive a message.
$Receive(m) \triangleq$
LET $i \triangleq m.mdest$
$j \triangleq m.msource$
IN Any RPC with a newer term causes the recipient to advance
its term first. Responses with stale terms are ignored.
$\lor\ UpdateTerm(i, j, m)$
$\lor\ \land\ m.mtype = RequestVoteRequest$
$\land\ HandleRequestVoteRequest(i, j, m)$
$\lor\ \land\ m.mtype = RequestVoteResponse$
$\land\ \lor\ DropStaleResponse(i, j, m)$
$\lor\ HandleRequestVoteResponse(i, j, m)$
$\lor\ \land\ m.mtype = AppendEntriesRequest$
$\land\ HandleAppendEntriesRequest(i, j, m)$
$\lor\ \land\ m.mtype = AppendEntriesResponse$
$\land\ \lor\ DropStaleResponse(i, j, m)$
$\lor\ HandleAppendEntriesResponse(i, j, m)$

End of message handlers.

Network state transitions

The network duplicates a message
$DuplicateMessage(m) \triangleq$
$\land\ Send(m)$
$\land$ UNCHANGED $\langle serverVars, candidateVars, leaderVars, logVars, hash\rangle$

The network drops a message
$DropMessage(m) \triangleq$
$\land\ Discard(m)$
$\land$ UNCHANGED $\langle serverVars, candidateVars, leaderVars, logVars, hash\rangle$

Defines how the variables may transition.

$$
\begin{aligned}
&Next \triangleq \ \land \lor \exists\, i \in Server : Timeout(i) \\
&\lor \exists\, i,\, j \in Server : RequestVote(i,\, j) \\
&\lor \exists\, i \in Server : BecomeLeader(i) \\
&\lor \exists\, i \in Server : ClientRequest(i) \\
&\lor \exists\, i \in Server : AdvancePromiseIndex(i) \\
&\lor \exists\, i \in Server : AdvanceCommitIndex(i) \\
&\lor \exists\, i,\, j \in Server : AppendEntries(i,\, j) \\
&\lor \exists\, i \in Server : \exists\, s \in 1\,..\,(step - 1) : Rollback(i,\, s) \\
&\lor \exists\, m \in ValidMessage(messages) : Receive(m) \\
&\lor \exists\, m \in SingleMessage(messages) : DuplicateMessage(m) \\
&\lor \exists\, m \in ValidMessage(messages) : DropMessage(m)
\end{aligned}
$$

History variable that tracks every log ever:

$$
\begin{aligned}
&\land allLogs' = allLogs \cup \{log[i] : i \in Server\} \\
&\land RecordStates \\
&\land step' = step + 1
\end{aligned}
$$

The specification must start with the initial state and transition according to Next.

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

5.16 Conclusion

SVR3 demonstrates the potential of systems that provide security through a combination of cryptography and a diverse set of hardware enclaves and clouds, without putting trust in any single hardware component. Using different types of enclaves leads to an array of deployment challenges stemming from heterogeneous attacker models. SVR3 is a powerful defense against the evolving landscape of enclave security: by distributing trust across enclaves and clouds through a cryptographic protocol, even if a new threat arises in one type of enclave, user secrets are still secure. SVR3 costs $0.0025/user/year and takes 365ms for a user to recover their key, which is a rare operation.

Chapter 6

Conclusion

6.1 Summary

In this dissertation, we explored addressing the challenges of deploying distributed-trust systems. We focused on two main themes: **adopting** and **scaling** distributed-trust systems. On improving the adoption of distributed-trust systems, we proposed **CostCO**, a cost modeling tool that helps developers understand the cost of secure multi-party computation (MPC) protocols, and **LegoLog**, a configurable transparency log that allows developers to choose the right trade-offs for their applications. On improving the scalability of distributed-trust systems, we introduced **Snoopy**, an oblivious storage system that scales like plaintext storage systems and allows users to store and retrieve data without revealing the data or its access patterns, and **SVR3**, a secret key recovery system for end-to-end encrypted messaging applications that allows users to securely recover their secret keys.

6.2 Future work

There are many open questions and future work directions still left to explore in improving the adoption and scalability of distributed-trust systems. Here are a few for the interested reader:

→ **Concrete bounds for the k-choice case in balls-into-bins.** In Snoopy (Chapter 4), we were able to securely and efficiently distribute requests across subORAMs by deriving a concrete bound for the 1-choice case in the balls-into-bins problem. However, we were not able to derive a concrete bound for the k-choice case, where the ball selects the least loaded bin out of k randomly selected bins. Asymptotically, the k-choice case is *exponentially* more efficient than the 1-choice case. If this is also true in practice, it would be a significant improvement for the overhead of securely distributing requests, and could lead to systems that leverage this bound, that also have to account for the item being stored in any of the k bins.

→ **Bridging the gap between theoretical security guarantees of TEEs and what we have today.** In SVR3 (Chapter 5), we proposed a system that leverages TEEs to provide strong se-

curity guarantees for key recovery in end-to-end encrypted messaging applications. When using these TEEs, we found that there are varying security guarantees depending on the TEE implementation and vendor. While we unify a threat model for TEEs in SVR3, and show how three such TEEs fit in our model, this required a lot of work to understand the security guarantees of each TEE implementation and vendor. In the cases where the TEE implementation did not provide the security guarantees we needed, we had to rely on other techniques to provide the necessary security guarantees. Continuing to bridge the gap between the theoretical security guarantees of TEEs and what we have today is important future work.

→ **Post-quantum cryptography.** As quantum computers become more powerful, they will be able to break many of the cryptographic primitives used in distributed-trust systems. A particularly relevant threat model is the *harvest now, decrypt later* model, where an adversary collects encrypted data now and waits until they have a quantum computer powerful enough to decrypt it. Developing and deploying distributed-trust systems that rely on cryptographic primitives that can withstand attacks in this model is necessary for long-term security.

Bibliography

[1] John Aas. Project update and new name for ISRG Prio services: Introducing Divvi Up, 2021. https://divviup.org/blog/prio-services-update/.

[2] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. Robust password-protected secret sharing. In *ESORICS*, 2016.

[3] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for Intel SGX. In *NDSS*, 2018.

[4] Mustafa Al-Bassam and Sarah Meiklejohn. Contour: A practical system for binary transparency. In *DPM/CBT@ESORICS*, 2018.

[5] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In *TCC*, 2020.

[6] AMD SEV-SNP: Strengthening VM isolation with integrity protection and more, 2020. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[7] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E Culler, and Raluca Ada Popa. WAVE: A decentralized authorization framework with transitive delegation. In *USENIX Security*, 2019.

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *EuroSys*, 2018.

[9] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. Nimble: Rollback protection for confidential cloud services. In *OSDI*, 2023.

[10] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, 2016.

[11] Apple. Advancing iMessage security: iMessage contact key verification. https://security.apple.com/blog/imessage-contact-key-verification/.

[12] Apple. Energy efficiency guide for iOS apps. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/index.html.

[13] Apple. iCloud Keychain security overview, 2021. https://support.apple.com/guide/security/icloud-keychain-security-overview-sec1c89c6f3b/.

[14] Apple and Google. Exposure notification privacy-preserving analytics (ENPA) white paper, 2021. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

[15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*, 2013.

[16] Asynchronous Enclave Exit Notify and the EDECCSSA user leaf function. https://www.intel.com/content/www/us/en/content-details/736463/white-paper-asynchronous-enclave-exit-notify-and-the-edeccssa-user-leaf-function.html.

[17] Attestation Service for Intel SGX. https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf.

[18] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *CCS*, 2011.

[19] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from SGX. In *FC*, 2017.

[20] Kenneth E Batcher. Sorting networks and their applications. In *AFIPS*, 1968.

[21] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the Libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

[22] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[23] Donald Beaver, Shafi Micali, and Phillip Rogaway. The round complexity of secure protocols. In *STOC*, 1990.

[24] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A system for secure multi-party computation. In *CCS*, 2008.

[25] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In *TCC*, 2020.

[26] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. In *STOC*, 2000.

[27] Brian N Bershad, David D Redell, and John R Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS*, 1992.

[28] Binary Transparency. Building trust in the software supply chain. https://binary.transparency.dev.

[29] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *CCS*. ACM, 2015.

[30] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.

[31] 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.

[32] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *IEEE S&P*, 2021.

[33] Joseph Bonneau. EthIKS: Using Ethereum to audit a CONIKS key transparency log. In *FC*, 2016.

[34] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In *MICRO*, 2019.

[35] George EP Box and Kenneth B Wilson. On the experimental attainment of optimum conditions. *Journal of the Royal wtatistical Society: Series B (Methodological)*, 1951.

[36] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *TCC*, 2016.

[37] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *DSN*, 2017.

[38] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*. USENIX, 2017.

[39] David Brown. Confidential computing: an AWS perspective, 2021. https://aws.amazon.com/blogs/security/confidential-computing-an-aws-perspective/.

[40] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security*, 2010.

[41] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. https://ia.cr/2015/472.

[42] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *CCS*, 2018.

[43] Ran Canetti, Rosario Gennaro, Amir Herzberg, and Dalit Naor. Proactive security: Long-term protection against break-ins. *RSA Laboratories' CryptoBytes*, 1997.

[44] Cape privacy: Privacy & trust management for machine learning. https://capeprivacy.com/.

[45] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[46] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[47] Anrin Chakraborti and Radu Sion. ConcurORAM: High-throughput stateless parallel multi-client ORAM. In *NDSS*, 2019.

[48] T-H Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel ram. In *ASIACRYPT*. IACR, 2017.

[49] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In *Asiacrypt*, 2017.

[50] T-H Hubert Chan and Elaine Shi. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In *TCC*, 2017.

[51] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: programmable, efficient, and scalable secure two-party computation for machine learning. In *EuroS&P*, 2019.

[52] Melissa Chase, Hannah Davis, Esha Ghosh, and Kim Laine. Acsesor: A new framework for auditable custodial secret storage and recovery. *Cryptology ePrint Archive 2022/1729*, 2022.

[53] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. SEEMless: Secure end-to-end encrypted messaging with less trust. In *CCS*, 2019.

[54] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1):65–75, 1988.

[55] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel ram: improved efficiency and generic constructions. In *TCC*, 2016.

[56] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE: Stealing intel secrets from SGX enclaves via speculative execution. In *EuroS&P*, 2019.

[57] Long Chen, Ya-Nan Li, Qiang Tang, and Moti Yung. End-to-same-end encryption: Modularly augmenting an app with an efficient, portable, and blind cloud storage. In *USENIX Security*, 2022.

[58] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P Ward. Reducing participation costs via incremental verification for ledger systems. *Cryptology ePrint Archive*, 2020.

[59] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. VoltPillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *USENIX Security*, 2021.

[60] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. *arXiv preprint arXiv:2303.15540*, 2023.

[61] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, 1995.

[62] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6), 1998.

[63] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. In *ISeCure*, 2011.

[64] Graeme Connell*, Vivian Fang*, Rolfe Schmidt*, Emma Dauterman, and Raluca Ada Popa. Secret key recovery in a global-scale end-to-end encryption system. In *OSDI*, 2024.

[65] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting precise single-stepping attacks through interrupt awareness for Intel SGX enclaves. In *USENIX Security*, 2023.

[66] Christopher Copeland and Hongxia Zhong. Tangaroa: a Byzantine fault tolerant Raft, 2016. https://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf.

[67] Robert M Corless, Gaston H Gonnet, David EG Hare, David J Jeffrey, and Donald E Knuth. On the Lambert W function. *Advances in Computational mathematics*, 1996.

[68] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

[69] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE S&P*, 2015.

[70] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. The pyramid scheme: Oblivious RAM for trusted processors. *arXiv preprint arXiv:1712.07882*, 2017.

[71] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[72] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, 2018.

[73] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[74] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. SafetyPin: Encrypted backups with human-memorable secrets. In *OSDI*, 2020.

[75] Emma Dauterman, Vivian Fang, Natacha Crooks, and Raluca Ada Popa. Reflections on trusting distributed trust. In *HotNets*, 2022.

[76] Emma Dauterman*, Vivian Fang*, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the bottlneck of scalable oblivious storage. In *SOSP*, 2021.

[77] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An encrypted search system with distributed trust. In *OSDI*, 2020.

[78] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *IEEE S&P*, 2022.

[79] Alex Davidson, Armando Faz-Hernandez, Nick Sullivan, and Christopher A. Wood. Oblivious pseudorandom functions (OPRFs) using prime-order groups. https://www.ietf.org/id/draft-irtf-cfrg-voprf-21.html.

[80] Gareth T. Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horvárth, and Tibor Jager. Security analysis of the WhatsApp end-to-end encrypted backup protocol. *Cryptology ePrint Archive 2023/843*, 2023.

[81] Antoine Delignat-Lavaud, Cédric Fournet, Kapil Vaswani, Sylvan Clebsch, Maik Riechert, Manuel Costa, and Mark Russinovich. Why should I trust your code? Confidential computing enables users to authenticate code running in TEEs, but users also need evidence this code is trustworthy. *ACM Queue*, 2023.

[82] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. {SEAL}: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, 2020.

[83] Daniel Demmler, Thomas Schneider, and Michael Zohner. Ad-hoc secure two-party computation on mobile devices using hardware tokens. In *USENIX Security*, 2014.

[84] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[85] Baltasar Dinis, Peter Druschel, and Rodrigo Rodrigues. RR: A fault model for efficient TEE replication. In *NDSS*, 2023.

[86] Yevgeniy Dodis, Shai Halevi, Ron D Rothblum, and Daniel Wichs. Spooky encryption and its applications. In *CRYPTO*, 2016.

[87] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *CCS*, 2017.

[88] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *STOC*, 1986.

[89] DualityTechnologies: Data encryption technology and secure collaboration. https://dualitytech.com/.

[90] Sisi Duan, Sean Peisert, and Karl N Levitt. hBFT: speculative Byzantine fault tolerance with minimum cost. *TDSC*, 2014.

[91] Steven Englehardt. Next steps in privacy-preserving telemetry with Prio, 2019. https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/.

[92] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *USENIX Security*, 2021.

[93] Saba Eskandarian and Matei Zaharia. ObliDB: oblivious query processing for secure databases. *VLDB*, 2019.

[94] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party oram for secure computation. In *ASIACRYPT*. IACR, 2015.

[95] Andrés Fábrega, Jack Cable, Michael A Specter, and Sunoo Park. Cryptographic verifiability for voter registration systems. *arXiv preprint arXiv:2503.03974*, 2025.

[96] Andrés Fábrega, Carolina Ortega Pérez, Armin Namavari, Ben Nassi, Rachit Agarwal, and Thomas Ristenpart. Injection attacks against end-to-end encrypted applications. In *IEEE S&P*, 2023.

[97] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://ia.cr/2012/144.

[98] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. CostCO: An automatic cost modeling framework for secure multi-party computation. In *EuroS&P*, 2022.

[99] Vivian Fang, Emma Dauterman, Akshat Ravoor, Akshit Dewan, and Raluca Ada Popa. LegoLog: A configurable transparency log. In *EuroS&P*, 2025.

[100] Fireblocks. https://www.fireblocks.com/platforms/mpc-wallet/.

[101] 5 advantages of a cloud-based EHR. https://www.carecloud.com/continuum/5-advantages-of-a-cloud-based-ehr-for-large-practices/.

[102] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. A low-latency, low-area hardware oblivious RAM controller. In *FCCM*, 2015.

[103] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C compiler for secure two-party computations. In *CC*, 2014.

[104] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In *Asiacrypt*, 2015.

[105] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[106] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. *Proceedings on Privacy Enhancing Technologies (PETS)*, 2017.

[107] Gemini. Cold storage, keys & crypto: How Gemini keeps assets safe. https://www.gemini.com/blog/cold-storage-keys-crypto-how-gemini-keeps-assets-safe.

[108] Tim Geoghegan. Exposure notifications private analytics: Lessons learned from running secure MPC at scale, 2022. https://divviup.org/blog/lessons-from-running-mpc-at-scale/.

[109] Tim Geoghegan, Mariana Raykova, and Frederic Jacobs. Exposure notifications private analytics. In *Real World Crypto*, 2022.

[110] EP George, J Stuart Hunter, William Gordon Hunter, Roma Bins, Kay Kirlin IV, and Destiny Carroll. *Statistics for experimenters: design, innovation, and discovery*. Wiley New York, NY, USA:, 2005.

[111] Esha Ghosh and Melissa Chase. Weak consistency mode in key transparency: Optiks. *Cryptology ePrint Archive*, 2024.

[112] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.

[113] Golang. go.sum database tree. https://sum.golang.org/latest.

[114] Golang. index.golang.org. https://index.golang.org/index?since=2022-03-03T19:08:52.997264Z.

[115] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, 2006.

[116] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 1996.

[117] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. Measuring empirical computational complexity. In *ESEC/FSE*, 2007.

[118] Michael T Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, 2011.

[119] Google. Key transparency design doc. https://github.com/google/keytransparency/blob/master/docs/design_new.md.

[120] Google. Trillian. https://github.com/google/trillian.

[121] Google. Trillian. https://github.com/google/trillian.

[122] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. In *PKC*, 2022.

[123] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*, 2020.

[124] 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.

[125] 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.

[126] 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*, 2018.

[127] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *EuroSys*, 2010.

[128] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable and decentralized trust infrastructure. In *IEEE DSN*, 2019.

[129] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *NSDI*, 2016.

[130] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[131] Feng Hao and Paul C van Oorschot. SoK: Password-authenticated key exchange–theory, practice, standardization and real-world lessons. In *AsiaCCS*, 2022.

[132] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *SOSP*, 2015.

[133] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In *PKC*, 2017.

[134] Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *CCS*, 2010.

[135] Wilko Henecka and Thomas Schneider. Faster secure two-party computation with less memory. In *CCS*, 2013.

[136] Ryan Henry. Polynomial batch codes for efficient it-pir. *PETS Symposium*, 2016.

[137] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.

[138] William J Hill and William G Hunter. A review of response surface methodology: a literature survey. *Technometrics*, 1966.

[139] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A Yavuz. Mose: Practical multi-user oblivious storage via secure enclaves. In *CODASPY*, 2020.

[140] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *PETS*, 2019.

[141] Katie Hockman. Module mirror and checksum database launched. https://go.dev/blog/module-mirror-launch.

[142] Benjamin Hof and Georg Carle. Software distribution transparency and auditability. *arXiv preprint arXiv:1711.07278*, 2017.

[143] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle^2: A low-latency transparency log system. In *IEEE S&P*, 2021.

[144] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2010.

[145] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[146] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[147] IETF. Key transparency (keytrans). https://datatracker.ietf.org/wg/keytrans/about/.

[148] Inpher: Secret computing and privacy-preserving analytics. https://www.inpher.io/.

[149] Intel. Intel xeon scalable platform built for most sensitive workloads. https://www.intel.com/content/www/us/en/newsroom/news/xeon-scalable-platform-built-sensitive-workloads.html.

[150] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, 2004.

[151] Muhammad Ishaq, Ana Milanova, and Vassilis Zikas. Efficient mpc via program analysis: A framework for efficient optimal mixing. In *CCS*, 2019.

[152] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *NDSS*, 2012.

[153] ISRG. Introducing ISRG Prio services for privacy respecting metrics. https://www.abetterinternet.org/post/introducing-prio-services/.

[154] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking down the processor via Rowhammer attack. In *SysTEX@EuroSys*, 2017.

[155] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT*, 2014.

[156] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *EuroS&P*, 2016.

[157] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In *ACNS*, 2017.

[158] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, 2018.

[159] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. Supporting Intel SGX on multi-socket platforms. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf.

[160] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security*, 2018.

[161] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[162] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, 2016.

[163] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Eurocrypt*, 2018.

[164] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *ACNS*, 2014.

[165] Key transparency new design. github.com/google/keytransparency/blob/master/docs/design_new.md.

[166] Keybase. https://keybase.io/.

[167] Hormuzd Khosravi. Runtime encryption of memory with Intel Total Memory Encryption - Multi-Key, 2022. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-10/intel-total-memory-encryption-multi-key-whitepaper.pdf.

[168] Knox. Knox custody. https://www.knoxcustody.com/security.

[169] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *USENIX Security*, 2021.

[170] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Private data management for decentralized ledgers. *Cryptology ePrint Archive 2018/209*, 2018.

[171] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *IEEE S&P*, 2019.

[172] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *TOCS*, 2010.

[173] Ivan Krstic. Behind the scenes with iOS security, 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf.

[174] Robert O Kuehl and RO Kuehl. Design of experiments: statistical principles of research design and analysis. 2000.

[175] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure TensorFlow inference. In *IEEE S&P*, 2020.

[176] Mu-Hsing Kuo. Opportunities and challenges of cloud computing to improve health care services. *Journal of medical Internet research*, 2011.

[177] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *SODA*, 2012.

[178] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *SOSP*, 2017.

[179] Albert Hyukjae Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. In *PoPETs*, 2016.

[180] Leslie Lamport. Specifying systems: The TLA+ language and tools for hardware and software engineers. 2002.

[181] Leslie Lamport. Byzantizing Paxos by refinement. In *DISC*, 2011.

[182] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962.

[183] Sean Lawlor and Kevin Lewi. Deploying key transparency at WhatsApp. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/.

[184] Sean Lawlor and Kevin Lewi. Deploying key transparency at whatsapp. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/.

[185] Ledger. How Ledger device generates 24-word recovery phrase. https://support.ledger.com/hc/en-us/articles/4415198323089-How-Ledger-device-generates-24-word-recovery-phrase, November 2023.

[186] 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.

[187] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *EuroSys*, 2020.

[188] 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.

[189] Julia Len, Melissa Chase, Esha Ghosh, Daniel Jost, Balachandar Kesavan, and Antonio Marcedone. ELEKTRA: Efficient lightweight multi-device key transparency. In *CCS*, 2023.

[190] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. OPTIKS: An optimized key transparency system. Cryptology ePrint Archive, Paper 2023/1515, 2023. https://eprint.iacr.org/2023/1515.

[191] Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, and Nickolai Zeldovich. Aardvark: A concurrent authenticated dictionary with short proofs. *IACR Cryptol. ePrint Arch.*, 2020:975, 2020.

[192] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, 2009.

[193] Yehuda Lindell, David Cook, Tim Geoghegan, Sarah Gran, Rolfe Schmidt, Ehren Kret, Darya Kaviani, and Raluca Ada Popa. The deployment dilemma: Merits & challenges of deploying MPC, 2023. https://mpc.cs.berkeley.edu/blog/deployment-dilemma.html.

[194] Yehuda Lindell, Benny Pinkas, Nigel P Smart, and Avishay Yanai. Efficient constant-round multi-party computation combining BMR and SPDZ. *Journal of Cryptology*, 2019.

[195] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. GhostRider: A hardware-software system for memory trace oblivious computation. *ASPLOS*, 2015.

[196] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *IEEE S&P*, 2015.

[197] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *CCS*, 2017.

[198] Stuart Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 1982.

[199] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *FAST*, 2013.

[200] Joshua Lund. Technology preview for secure value recovery, 2019. https://signal.org/blog/secure-value-recovery/.

[201] Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible Byzantine fault tolerance. In *SIGSAC*, 2019.

[202] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. 2023. https://eprint.iacr.org/2023/081.

[203] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. Candid: Can-do decentralized identity with legacy compatibility, Sybil-resistance, and accountability. In *IEEE S&P*, 2021.

[204] Moxie Marlinspike. The difficulty of private contact discovery, 2014. https://signal.org/blog/contact-discovery/.

[205] 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.

[206] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. BITE: Bitcoin lightweight client privacy using trusted execution. In *USENIX Security*, 2019.

[207] Vasilios Mavroudis, Andrea Cerulli, Petr Svenda, Dan Cvrcek, Dusan Klinec, and George Danezis. A touch of evil: High-assurance cryptographic hardware from untrusted components. In *CCS*, 2017.

[208] Marcela S Melara, Joseph Blankstein, Aaron aind Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.

[209] Memtier benchmark. https://github.com/RedisLabs/memtier_benchmark.

[210] Ralph Merkle. Method of providing digital signatures, 1979.

[211] Signal Messenger. Signal Android client. https://github.com/signalapp/Signal-Android.

[212] Signal Messenger. Signal desktop client. https://github.com/signalapp/Signal-Desktop.

[213] Signal Messenger. Signal iOS client. https://github.com/signalapp/Signal-iOS.

[214] Meta. End-to-end encryption on Messenger explained, 2024. https://about.fb.com/news/2024/03/end-to-end-encryption-on-messenger-explained/.

[215] Thibault Meunier and Mari Galicer. Cloudflare helps verify the security of end-to-end encrypted messages by auditing key transparency for WhatsApp. https://blog.cloudflare.com/key-transparency/.

[216] Microsoft. Bitlocker whitepaper Windows 10. https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_firmware/pdf_3/Bitlocker_White_Paper_Windows_10.pdf, 2018.

[217] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *IEEE S&P*, 2018.

[218] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE TPDS*, 2001.

[219] Model transparency. https://github.com/google/model-transparency.

[220] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[221] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, 2017.

[222] Payman Mohassel and Peter Rindal. ABY3: a mixed protocol framework for machine learning. In *CCS*, 2018.

[223] MPC alliance. https://www.mpcalliance.org/.

[224] Graham Mudd. Privacy-enhancing technologies and building for the future, 2022. https://www.facebook.com/business/news/building-for-the-future.

[225] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In *Eurocrypt*, 2016.

[226] Multi-party computation ceremonies in Zcash. https://z.cash/technology/paramgen/.

[227] 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.

[228] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008.

[229] 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.

[230] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: Proactive Software-Update transparency via collectively signed skipchains and verified builds. In *USENIX Security*, 2017.

[231] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In *USENIX Security*, 1998.

[232] Nitro secure module. https://github.com/aws/aws-nitro-enclaves-nsm-api/tree/v0.4.0.

[233] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[234] Diego Ongaro. *Consensus: Bridging theory and practice*. Stanford University, 2014.

[235] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[236] Taku Onodera and Tetsuo Shibuya. Succinct oblivious ram. *arXiv preprint arXiv:1804.08285*, 2018.

[237] OpenEnclave. https://github.com/openenclave/openenclave.

[238] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for SGX enclaves. In *EuroSys*, 2017.

[239] Chris Orsini, Alessandra Scafuro, and Tanner Verber. How to recover a cryptographic secret from the cloud. *Cryptology ePrint Archive 2023/1308*, 2023.

[240] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, 1990.

[241] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *IEEE S&P*, 2011.

[242] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. Cryptology ePrint Archive, Report 2020/1225, 2020. https://eprint.iacr.org/2020/1225.

[243] Erman Pattuk, Murat Kantarcioglu, Huseyin Ulusoy, and Bradley Malin. Cheapsmc: A framework to minimize secure multiparty computation cost in the cloud. In *DBSec*, 2016.

[244] Paxos. https://paxos.com/crypto-brokerage/.

[245] Trevor Perrin. KEM-based hybrid forward secrecy for Noise. 2018. https://github.com/noiseprotocol/noise_hfs_spec/blob/master/output/noise_hfs.pdf.

[246] Trevor Perrin. The Noise protocol framework. 2018.

[247] Robin L Plackett and J Peter Burman. The design of optimum multifactorial experiments. *Biometrika*, 1946.

[248] 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.

[249] 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.

[250] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *NSDI*, 2018.

[251] Preveil. https://www.preveil.com/.

[252] Protocol buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf.

[253] Proton Mail. https://proton.me/mail.

[254] Friedrich Pukelsheim. *Optimal design of experiments*. SIAM, 2006.

[255] Martin Raab and Angelika Steger. Balls into bins–a simple and tight analysis. In *RANDOM*, 1998.

[256] Martin Raab and Angelika Steger. "balls into bins"—a simple and tight analysis. In *RANDOM*, 1998.

[257] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE S&P*, 2021.

[258] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. *arXiv preprint arXiv:2008.00332*, 2020.

[259] MV Ramakrishna. Computing the probability of hash table/urn overflow. *Communications in Statistics-Theory and Methods*, 16(11):3343–3353, 1987.

[260] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Knowledge inference for optimizing secure multi-party computation. In *IEEE S&P*, 2014.

[261] Redis. https://redis.io/.

[262] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, and Kent Seamons. A usability study of five two-factor authentication methods. In *SOUPS*, 2019.

[263] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security*, 2015.

[264] Pedro Reviriego, Lars Holst, and Juan Antonio Maestro. On the expected longest length probe sequence for hashing with separate chaining. *Journal of Discrete Algorithms*, 9(3):307–312, 2011.

[265] 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.

[266] Riddle and code. https://www.riddleandcode.com/blog-posts/hardware-security-modules-vs-secure-multi-party-computation-in-digital-asset-custody.

[267] Ripple. https://ripple.com/xrp/.

[268] Mark Dermot Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*, 2014.

[269] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. TaoStore: Overcoming asynchronicity in oblivious data storage. In *IEEE S&P*, 2016.

[270] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. In *NDSS*, 2018.

[271] Andy Sayler, Taylor Andrews, Matt Monaco, and Dirk Grunwald. Tutamen: A next-generation secret-storage platform. In *SoCC*, 2016.

[272] SCALE-MAMBA. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[273] Axel Schroepfer and Florian Kerschbaum. Forecasting run-times of secure two-party computation. In *QEST*, 2011.

[274] 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.

[275] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, 2017.

[276] Scotiabank's chief risk officer on the state of anti–money laundering. https://mck.co/2ATh2IU, October 2019.

[277] Microsoft SEAL (release 3.4). https://github.com/Microsoft/SEAL, 2019.

[278] Sepior. https://sepior.com/products/advanced-mpc-wallet.

[279] Mary Shacklett. Financial services companies are starting to use the cloud for big data and ai processing. https://www.techrepublic.com/article/financial-services-companies-are-starting-to-use-the-cloud-for-big-data-and-ai-processing/, 2020.

[280] Hossein Shafagh, Lukas Burkhalter, Sylvia Ratnasamy, and Anwar Hithnawi. Droplet: Decentralized authorization and access control for encrypted data streams. In *USENIX Security*, 2020.

[281] Pavitra Shankdhar. Popular tools for brute-force attacks. https://resources.infosecinstitute.com/topics/hacking/popular-tools-for-brute-force-attacks/, 2020.

[282] Shared machine learning: Ant financial's solution for data privacy. https://link.medium.com/CgDVD0mtbab.

[283] Rob Shirley. Internet security glossary, version 2. https://datatracker.ietf.org/doc/html/rfc4949.

[284] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *ESEC/FSE*, 2015.

[285] Signal. The difficulty of private contact discovery, 2014. https://signal.org/blog/contact-discovery/.

[286] Signal Messenger. Secure Value Recovery Service v2/3. https://github.com/signalapp/SecureValueRecovery2.

[287] Signal Messenger. https://signal.org/.

[288] Signal revenue & usage statistics. https://www.businessofapps.com/data/signal-statistics/.

[289] Sudheesh Singanamalla, Suphanat Chunhapanya, Marek Vavruša, Tanya Verma, Peter Wu, Marwan Fayed, Kurtis Heimerl, Nick Sullivan, and Christopher Wood. Oblivious DNS over HTTPS (ODoH): A practical privacy enhancement to DNS. *PoPETs*, 2021.

[290] Solana. Solana decentralized exchange. https://soldex.ai/wp-content/uploads/2021/07/Soldex.ai-whitepaper-.pdf.

[291] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *IEEE S&P*, 2015.

[292] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious cloud storage. In *IEEE S&P*, 2013.

[293] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. In *NDSS*, 2012.

[294] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, 2013.

[295] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security*, 2016.

[296] Roberto Tamassia. Authenticated data structures. In *ESA*, 2003.

[297] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *USENIX Security*, 2017.

[298] The battle inside Signal. https://www.theverge.com/22249391/signal-app-abuse-messaging-employees-violence-misinformation.

[299] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *CCS*, 2019.

[300] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via Bitcoin. In *IEEE S&P*, 2017.

[301] Shruti Tople, Yaoqi Jia, and Prateek Saxena. PRO-ORAM: Practical read-only oblivious RAM. In *RAID*, 2019.

[302] Kevin Townsend. Solving the quantum decryption 'harvest now, decrypt later' problem. 2022. https://www.securityweek.com/solving-quantum-decryption-harvest-now-decrypt-later-problem/.

[303] Nora Trapp. Key to simplicity: Squeezing the hassle out of encryption key recovery, 2024. https://www.juicebox.xyz/blog/key-to-simplicity-squeezing-the-hassle-out-of-encryption-key-recovery.

[304] Anna Trikalinou and Dan Lake. Taking DMA attacks to the next level. 2017.

[305] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. VeRSA: Verifiable registries with efficient client audits from RSA authenticated dictionaries. In *CCS*, 2022.

[306] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency dictionaries with succinct proofs of correct operation. In *NDSS*, 2022.

[307] Unbound tech. https://www.unboundtech.com/.

[308] Unbound Security. The Unbound CORE MPC key vault.

[309] 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.

[310] 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.

[311] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX*, 2017.

[312] 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.

[313] 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.

[314] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. *arXiv preprint arXiv:2006.13353*, 2020.

[315] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, 2016.

[316] Shabsi Walfish. Google Cloud Key Vault Service. Google, 2018. https://developer.android.com/about/versions/pie/security/ckv-whitepaper.

[317] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *NSDI*, 2017.

[318] Weili Wang, Sen Deng, Jianyu Niu, Michael K Reiter, and Yinqian Zhang. Engraft: Enclave-guarded Raft on Byzantine faulty nodes. In *CCS*, 2022.

[319] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, 2017.

[320] WhatsApp. Security of end-to-end encrypted backups, 2021. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf.

[321] Kyle Wiggers. Apple launches end-to-end encryption for iCloud data. *TechCrunch*, 2022.

[322] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.

[323] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, 2012.

[324] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[325] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *SoCC*, 2017.

[326] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *SFCS*, 1986.

[327] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *ACM PODC*, 2019.

[328] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. https://ia.cr/2015/1153.

[329] Tong Zhang. Adaptive forward-backward greedy algorithm for learning sparse representations. *IEEE Transactions on Information Theory*, 2011.

[330] Yihua Zhang, Aaron Steele, and Marina Blanton. Picco: A general-purpose compiler for private distributed computation. In *CCS*, 2013.

[331] Ekaterina Zharova. The state of Go. https://blog.jetbrains.com/go/2021/02/03/the-state-of-go/.

[332] 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.

[333] Ruiyu Zhu, Darion Cassel, Amr Sabry, and Yan Huang. Nanopi: extreme-scale actively-secure multi-party computation. In *CCS*, 2018.