Secure Systems from Insecure Components

Emma Dauterman

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-146
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-146.html

July 10, 2024

Copyright © 2024, by the author(s).
All rights reserved.

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

Secure Systems from Insecure Components

by

Emma Dauterman

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Raluca Ada Popa, Co-chair
Professor Ion Stoica, Co-chair
Assistant Professor Natacha Crooks
Assistant Professor Henry Corrigan-Gibbs

Summer 2024

Secure Systems from Insecure Components

Copyright 2024
by
Emma Dauterman

Abstract

Secure Systems from Insecure Components

by

Emma Dauterman

Doctor of Philosophy in Computer Science

University of California, Berkeley

Associate Professor Raluca Ada Popa, Co-chair

Professor Ion Stoica, Co-chair

In many computer systems today, an attacker that compromises just one system component can steal many users' data. Unfortunately, past experience shows that attackers are very effective at compromising system components, whether by exploiting some software vulnerability, compromising hardware, or launching a phishing attack.

In this thesis, we show how to build systems that provide strong security and privacy properties even if the individual components are insecure. This way, even if an attacker compromises any single component in the system, it cannot compromise user security and privacy. While this property is possible to achieve in theory using general-purpose cryptographic techniques, the challenge is to instantiate it efficiently in practice. The key idea is to co-design the system with the cryptography to reduce costs.

We examined two core aspects of this problem: hiding queries and securing accounts. Users who store their data encrypted at servers still need to query their data. We built systems that provide both strong privacy guarantees and good concrete efficiency for keyword search (DORY), time-series analytics queries (Waldo), and object stores (Snoopy). Users also need to protect their accounts in the event of client device loss or compromise, but also in the event of service provider compromise. We built an encrypted backup system that relies on secure hardware without fully trusting it (SafetyPin) and a service that records every authentication without learning private information (larch).

The Signal end-to-end encrypted messaging application uses some of the techniques in Snoopy to scale its private contact discovery service, which privately matches user contacts to Signal users.

To my family.

Contents

Contents ii

List of Figures viii

List of Tables xiii

1 Introduction 1
- 1.1 Approach . . . 1
- 1.2 Private queries . . . 3
  - 1.2.1 DORY: a private search system for end-to-end encrypted filesystems . . . 3
  - 1.2.2 Waldo: a private time-series database . . . 4
  - 1.2.3 Snoopy: an oblivious, scalable object store . . . 5
- 1.3 Secure accounts . . . 6
  - 1.3.1 SafetyPin: an encrypted backup system that resists hardware attacks . . . 7
  - 1.3.2 Larch: an authentication logging system without a single point of security failure . . . 8
- 1.4 Impact and adoption . . . 9

I Private queries 11

2 DORY: A private search system for end-to-end encrypted filesystems 12
- 2.1 Introduction . . . 12
  - 2.1.1 Summary of techniques . . . 14
- 2.2 Identifying a system model . . . 15
  - 2.2.1 System requirements . . . 17
  - 2.2.2 Distributed trust requirements . . . 17
  - 2.2.3 Opportunities . . . 18
  - 2.2.4 Building a distributed trust system . . . 18
  - 2.2.5 Future directions . . . 19
- 2.3 System design overview . . . 19
  - 2.3.1 The underlying filesystem . . . 19

2.3.2 The DORY API . . . 20
2.3.3 System architecture . . . 21
2.3.4 Threat model and security properties . . . 21
2.4 Search design . . . 23
2.4.1 A strawman search index . . . 23
2.4.2 Eliminating search access patterns . . . 24
2.4.3 Protecting against malicious attackers . . . 25
2.4.4 Supporting dynamic membership . . . 26
2.4.5 Final DORY protocol . . . 26
2.5 Replication across trust domains . . . 26
2.5.1 Algorithm . . . 29
2.5.2 Batching . . . 31
2.6 Implementation . . . 31
2.6.1 Parallelism . . . 31
2.6.2 Fast PRF evaluation . . . 32
2.7 Evaluation . . . 32
2.7.1 Baselines . . . 33
2.7.2 Latency . . . 34
2.7.3 Throughput . . . 35
2.7.4 Storage . . . 36
2.7.5 Bandwidth . . . 37
2.7.6 Cost . . . 37
2.8 Related Work . . . 38
2.8.1 Related work before publication . . . 38
2.8.2 Subsequent related work . . . 39
2.9 Security analysis . . . 40
2.9.1 Definitions . . . 42
2.9.2 Proof . . . 45
2.9.3 Our questions for companies . . . 49
2.10 Conclusion . . . 50

3 Waldo: A private time-series database **51**
3.1 Introduction . . . 51
3.1.1 Summary of techniques . . . 52
3.2 System overview . . . 54
3.2.1 Time-series workloads . . . 54
3.2.2 Running example: Remote Patient Monitoring . . . 54
3.2.3 System architecture . . . 55
3.2.4 Waldo API . . . 56
3.2.5 Notation . . . 57
3.3 Threat model and security guarantees . . . 57
3.4 Multi-predicate queries . . . 58

3.4.1 Single predicate with semihonest security 59
3.4.2 Multiple predicates with semihonest security 62
3.4.3 Multiple predicates with malicious security 63
3.4.4 Putting it together 65
3.5 Complex aggregates over time ranges 66
3.6 Implementation 70
3.7 Evaluation 72
3.7.1 Baselines 72
3.7.2 Queries for real-world applications 73
3.7.3 Latency: WaldoTable 74
3.7.4 Latency: WaldoTree 76
3.7.5 Throughput 76
3.7.6 Communication 77
3.7.7 System cost 78
3.8 Limitations and future work 79
3.9 Related work 79
3.9.1 Related work at the time of publication 79
3.9.2 Subsequent related work 80
3.10 Security analysis 81
3.11 Conclusion 84

4 Snoopy: An oblivious, scalable object store 86

4.1 Introduction 86
4.1.1 Summary of techniques 87
4.2 Security and correctness guarantees 89
4.2.1 Formalizing security 91
4.3 System overview 92
4.3.1 System architecture 93
4.3.2 Real-world applications 93
4.4 Oblivious load balancer 94
4.4.1 Setting the batch size 94
4.4.2 Oblivious batch coordination 96
4.4.3 Scaling the load balancer 99
4.5 Throughput-optimized subORAM 99
4.6 Planner 101
4.7 Implementation 102
4.8 Evaluation 103
4.8.1 Baselines 103
4.8.2 Throughput scaling 104
4.8.3 Scaling for latency and data size 106
4.8.4 Microbenchmarks 107
4.8.5 Planner 108

4.9 Discussion . . . 108
4.10 Related work . . . 109
4.10.1 Related work at the time of publication . . . 109
4.10.2 Subsequent related work . . . 111
4.11 Parameter analysis . . . 111
4.12 Security analysis . . . 113
4.12.1 Enclave definition . . . 113
4.12.2 Our model . . . 113
4.12.3 Oblivious storage definitions . . . 114
4.12.4 Oblivious building blocks . . . 116
4.12.5 SubORAM . . . 117
4.12.6 Snoopy . . . 118
4.12.7 Discussion of multiple clients . . . 122
4.13 Linearizability . . . 123
4.14 Access control . . . 129
4.15 Conclusion . . . 130

II Secure accounts 131

5 SafetyPin: An encrypted backup system that resists hardware attacks 132
5.1 Introduction . . . 132
5.2 The setting . . . 134
5.3 System goals . . . 136
5.4 Architecture overview . . . 137
5.4.1 The back-up process . . . 138
5.4.2 The recovery process . . . 139
5.5 Protecting the mapping of users to HSMs with location-hiding encryption . . . 141
5.5.1 Our construction . . . 142
5.6 The distributed log . . . 143
5.6.1 Underlying data structure . . . 144
5.6.2 Building a distributed log . . . 146
5.6.3 Transparency and external auditability . . . 148
5.7 Forward security by puncturable encryption . . . 149
5.7.1 Background: Puncturable encryption . . . 149
5.7.2 Outsourced storage with secure deletion . . . 150
5.7.3 Building secure outsourced storage . . . 151
5.8 Extensions and deployment considerations . . . 152
5.9 Implementation and evaluation . . . 153
5.9.1 Microbenchmarks . . . 154
5.9.2 End-to-end costs . . . 155
5.10 Related work . . . 158

5.10.1 Related work before publication 158
5.10.2 Subsequent related work 159
5.11 Security analysis 160
5.11.1 Syntax 160
5.11.2 Definitions 161
5.11.3 A tedious combinatorial lemma 164
5.11.4 Our construction 165
5.11.5 Proof of correctness 167
5.11.6 Proof of security 167
5.12 Conclusion 173

6 Larch: An authentication logging system without a single point of security failure 174
6.1 Introduction 174
6.2 Design overview 176
6.2.1 Entities 176
6.2.2 Protocol flow 177
6.2.3 System goals 179
6.2.4 Non-goals and extensions 180
6.3 Logging for FIDO2 181
6.3.1 Background 181
6.3.2 Split-secret authentication 182
6.3.3 Two-party ECDSA with preprocessing 183
6.4 Logging for time-based one-time passwords 186
6.4.1 Background: TOTP 186
6.4.2 Split-secret authentication for TOTP 186
6.5 Logging for passwords 188
6.5.1 Protocol overview 188
6.5.2 Split-secret authentication for passwords 188
6.6 Protecting against log misbehavior 189
6.7 Implementation 190
6.8 Evaluation 191
6.8.1 End-user cost 191
6.8.2 Cost to deploy a larch service 192
6.9 Discussion 194
6.10 Related work 197
6.10.1 Related work at the time of publication 197
6.10.2 Subsequent related work 198
6.11 ECDSA with additive key derivation and presignatures 198
6.12 Two-party ECDSA with preprocessing 203
6.12.1 Syntax and construction 203
6.12.2 Malicious security with half-authenticated secure multiplication 203
6.12.3 Security proof for our construction 206

6.13 Protocol details for password-based logging 209
6.13.1 Zero-knowledge proofs for discrete log relations 209
6.13.2 Protocol for passwords 209
6.14 Conclusion 210

III Conclusion 212

Bibliography 214

List of Figures

3 System software architecture. The figure shows the structure of the software rather than the physical system itself, where the server is instantiated across multiple machines. . . . 20

4 DORY's physical system architecture for a single partition (filesystem server not pictured). Replicas should be deployed in different trust domains, and each holds a copy of the search index. 21

5 Search index layout for n documents with Bloom filters of length m. Updates write *rows* and searches retrieve *columns*. 23

6 Pseudocode for client update protocol. 27

7 Pseudocode for client search protocol. 28

8 Pseudocode for server protocols. 29

9 System architecture and protocol flow for updates (left) and searches (right). ❶ Client sends update to master. ❷ Master propagates updates to replicas. ❸ Client requests version number(s) from master. ❹ Client splits search request across replicas. 30

11 Search latency and update latency. The left and center figures use a logarithmic scale on both axes, and the right figure uses a linear scale on both axes (p denotes server parallelism). The update latency of leaky DORY exactly matches that of DORY, and the search latency of semihonest DORY is slightly less than that of DORY. 35

12 Latency for mobile client and desktop client. Both plots use a logarithmic scale on both axes. 36

13 Throughput under a variety of workloads (U indicates updates, S indicates searches). The performance of semihonest DORY closely matches that of DORY. All plots use a logarithmic scale on both axes. 36

14 Effect of parallelism (p denotes the degree of parallelism) on throughput for different workloads (U indicates updates, S indicates searches). 37

15 Storage space and bandwidth for DORY in comparison to other baselines. The update bandwidth of leaky DORY exactly matches that of DORY, and the search bandwidth of semihonest DORY is slightly less than that of DORY. 38

16 Real world, including the clients C with access to a given folder (all other clients are controlled by $\mathcal{A}$), the honest servers P, and the adversary $\mathcal{A}$, which includes the compromised servers. 41

17 Ideal world, including the client C with access to a given folder (all other clients are controlled by $\mathcal{A}$), the filesystem ideal functionality $\mathcal{F}_{\mathsf{filesys}}$, the consensus ideal functionality $\mathcal{F}_{\mathsf{consensus}}$, the DORY ideal functionality $\mathcal{F}$, the honest servers P, the simulator $\mathcal{S}$, and the adversary $\mathcal{A}$, which includes the compromised servers. 41

1 System architecture. Here the ECG sensor and blood pressure monitor are data producers, and the doctors are queriers. The servers are deployed in different trust domains. . . . 55

2 Query predicate evaluation with FSS and RSS. 61

3 Client $\mathsf{WaldoTable.Query}$ algorithm. $\mathsf{Gen}(1^\lambda, P_i, \beta)$ refers to $\mathsf{Gen}^{=}(1^\lambda, a, \beta)$, $\mathsf{Gen}^{<}(1^\lambda, a, \beta)$, or $\mathsf{Gen}^{\mathsf{IC}}(1^\lambda, a, b, \beta)$ depending on the predicate P_i being $x = a, x < a$, or $a < x < b$, respectively. We denote P_i's feature ID as $\mathsf{idx}(P_i)$, and $\{p_i\}_i$ denotes $\{p_1, \ldots, p_n\}$. . . . 66

4 Server $\mathsf{WaldoTable.Query}$ algorithm. We use N for the window size, n for the number of query predicates, and $\mathsf{ID} \in \{1,2,3\}$ for the server id. Variables with "tilde" denote RSS shares. The variables D, T refer to RSS shares of database and its corresponding shared one-hot index, respectively. We use $(D_p)_i$ to denote the ith record's pth feature in D (not in one-hot form), T_{p_i} refers to the shared one-hot index for p_ith feature, and $\tilde{F}_i$ denotes ith RSS share in $\tilde{F}$. We use $\mathsf{first}, \mathsf{second}$ to access the respective component from an RSS share. For simplicity we assume that predicates are chained via conjunctions; disjunctions can be expressed by adding negations. 67

5 Node activation for the query $0 < x < 7$. Green nodes are activated, and nodes with dashed boundary are in the covering set. 68

6 Client $\mathsf{WaldoTree.Query}$ algorithm. n is the number of leaves in the current shared aggregate tree. Here the aggregate is computed over time range 0 to t. General case follows similarly by using double the keys and servers returning $2\log n + 1$ values. Aggregation function Agg is defined by clients during call to Init procedure; it takes in a list of values and outputs their aggregate. $\{x^i\}_{i=a}^{b}$ denotes $\{x^a, \ldots, x^b\}$. 70

7 Server $\mathsf{WaldoTree.Query}$ algorithm. n is the number of leaves, $\mathsf{ID} \in \{1,2,3\}$ is the server id. T denotes the shared aggregate tree corresponding to $\mathsf{WaldoTree}$ object. $T_d^{(i)}$ denotes the ith node on dth level of the tree and $i.\mathsf{left}, i.\mathsf{right}$ access the value stored on the left and right child of a node i, respectively. $(a,b) \odot (c,d) = ac + bd$. Here the aggregate is computed over time range 0 to t, and so on line 10, the right child's activation status is used at parent. General case follows similarly by using twice as many keys and returning $2\log n + 1$ values. 71

9 Breakdown of $\mathsf{WaldoTable}$ query latency with different numbers of predicates (P). . . . 75

10 $\mathsf{WaldoTable}$ query latency is for 8 predicates, and latency for point and range predicates is almost identical. 76

11 $\mathsf{WaldoTable}$ is configured with 8 predicates and has similar throughput for point and range predicates. ORAM throughput fluctuates due to the fact that it is not fully oblivious (doesn't make the max number of accesses) and we randomly sample the data and queries. 77

12 Overhead and cost of total bandwidth between servers for a $\mathsf{WaldoTable}$ query with 8 predicates. We use the minimum AWS egress bandwidth cost of \$0.05/GB [38] to compute the cost (bandwidth pricing is based on total egress bandwidth per month). . . 78

13 WaldoTable is configured with 2^{10} records and has almost identical bandwidth for point and range predicates. 78

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. 88
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. 93
3 Dummy request overhead. A 50% overhead means for every two real requests there is one dummy request. 95
4 The total real request capacity of our system for an epoch, assuming ≤1K requests per subORAM per epoch. 95
5 Generating batches of requests at the load balancer. 97
6 Mapping subORAM responses to client requests at the load balancer. 98
7 Processing a batch of requests at a subORAM. 101
9 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. 104
10 Throughput of Snoopy using Oblix [361] as a subORAM (2M objects, 160B block size). We measure throughput with different maximum average latencies. 105
11 (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. 106
12 Breakdown of time to process one batch for different data sizes (one load balancer and one subORAM). 107
13 (a) Parallelizing bitonic sort across multiple threads. (b) Parallelizing batch processing at the subORAM across multiple enclave threads (batch size 4K requests). 107
14 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. 108
15 Real experiment for protocol Π running inside the enclave ideal functionality $\mathcal{F}_{\mathsf{Enc}}$ where γ is the trace. 114
16 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. 114
17 Real and ideal experiments for an oblivious storage scheme. 115
18 Ideal functionalities. 116
19 Our subORAM construction. 119

20 Simulator algorithms SimSubORAM = (Initialize, BatchAccess). . . . 120
21 Our Snoopy construction. . . . 123
22 Simulator algorithms SimSnoopy = (Initialize, BatchAccess). . . . 123
23 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. . . . 124
24 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. . . . 125
25 Our load balancer construction. . . . 126
26 Load balancer simulator for SimLoadBalancer.BatchAccess. . . . 127

1 Our cluster of 100 low-cost hardware security modules (SoloKeys [463]) on which we evaluate SafetyPin. . . . 133
3 An overview of the recovery-protocol flow. Each HSM i holds a secret key sk_i. The client holds a vector mpk of all HSMs' public keys. ❶ During backup, the client uses its PIN and the master public key to encrypt its data msg into a recovery ciphertext ct. The client then uploads this recovery ciphertext ct to the service provider. ❷ During recovery, the client downloads its recovery ciphertext. ❸ The client asks the data center to log its recovery attempt. ❹ The service provider collects a batch of client log-insertion requests, updates the log, and aggregates the new log into a Merkle tree. The service provider and HSMs run a log-update protocol. At the end of this protocol, each HSM holds the root of the Merkle tree computed over the latest log. ❺ The service provider sends the client a Merkle proof π that the client's recovery attempt is included in the latest log (i.e., in the latest Merkle root). ❻ The client sends the recovery ciphertext ct and log-inclusion proof π to the subset of HSMs needed to decrypt the recovery ciphertext. ❼ The HSMs check the proof and return shares of the decrypted ciphertext to the client. The client uses these to recover the backed-up data msg. . . . 138
4 Since HSMs in SafetyPin revoke their ability to decrypt a client's recovery ciphertext, SafetyPin protects against HSM compromise attacks that take place before recovery begins and after it completes. An attacker who can compromise HSMs while recovery is in progress can break security. . . . 141
5 The protocol that the service provider and HSMs use to update the HSM's log digest. . 144
6 The outsourced-storage scheme has a tree of keys. An arrow $a \to b$ denotes that value b is stored encrypted under key a. A service provider that stores all values it sees and later compromises the HSM state (sk') still does not learn the deleted data_3 value. . . . 151
8 Log-audit time after inserting 10K recovery attempts for a log with roughly 100M recovery attempts. We only measure the auditing time for 100 HSMs as we only had 100 SoloKeys; we distribute the work as if there were N HSMs. . . . 154
9 Time to run puncturable encryption on a single HSM as the maximum number of allowed punctures (and also secret key size) grows. The cost of our outsourced storage scheme dominates, though the access time is logarithmic in the size of the key. . . . 155

10 Breakdown of time to save (on Android Pixel 4 phone) and recover (using our SoloKey cluster). We do not consider the time to encrypt or decrypt disk images. 156
11 Recovery time grows slowly as cluster size n increases. The bits of security lost refers to the difference between the advantage of an attacker against a SafetyPin deployment with the given value of n and an attacker trying to guess the user's PIN. 157
12 Estimated number of SafetyPin-protected recoveries per year supported by clusters of different HSM models and costs. We use g^x/sec to compute the expected throughput of more powerful HSMs based on our measurements using SoloKeys (Table 2). 157
13 Data center sizes necessary to process different request rates with various 99th-percentile latency requirements. 158
15 Our construction of location-hiding encryption. 166

1 The client and log service run split-secret authentication where the client obtains the credential for `amazon.com` and the log service obtains an encryption of `amazon.com` under the client's key. The client's inputs are its share x of the authentication secret, the archive key k, a random nonce r, and the string `amazon.com`. The log's inputs are its shares $y_{\mathtt{amazon}}, \ldots, y_{\mathtt{google}}$ of all the client's authentication secrets and the commitment cm to the archive key generated at enrollment. The $\mathsf{MakeCred}$ function takes extra inputs for FIDO2 and TOTP. 179
2 Larch security goals. 180
3 On the left, larch FIDO2 latency decreases as the number of client cores increases (latency is independent of the number of relying parties). In the center, larch password latency grows with the number of relying parties, with the majority of the time spent on client proof generation. On the right, larch TOTP latency grows with the number of relying parties, with the majority of the time spent in an input-independent "offline" phase as opposed to the input-dependent "online" phase (both phases require network communication). 193
4 On the left, per-client storage overhead at the log decreases as presignatures are replaced with authentication records (client enrolls with 10K presignatures). On the right, minimum cost of supporting more authentications with passwords, (128 relying parties), FIDO2, and TOTP (20 relying parties). Both axes use a logarithmic scale. 194
5 Communication for larch with passwords increases logarithmically with the number of relying parties (both axes use a logarithmic scale). 195
7 Our experiment for security of ECDSA with additive key derivation and presignatures. 201
8 Security experiment for ECDSA with additive key derivation and presignatures from Groth and Shoup [220]. 202
9 Two-party ECDSA signing protocol with preprocessing. 204
10 Π_{HalfMul} protocol. 205
11 The details of the larch protocol for password-based authentication. 211

List of Tables

1 Summary of thesis work. Functionality, security and privacy, and performance properties are described informally at a high level. 2

1 The search use-cases for each of the five companies. 16
2 Survey statistics. In accordance with the companies' confidentiality wishes, we report most fields in aggregate although we report individual responses for maximum permissible search latency (only 4 of the companies responded). 16
10 On the left, Bloom filter sizes (in bytes) necessary for < 1 expected false positive assuming an average of 73.18 keywords per document where each keyword hashes to 7 Bloom filter indexes (Table 10a). On the right, breakdown of search latency without parallelism and end-to-end search latency with parallelism where p is the degree of server parallelism (Table 10b). 32

8 WaldoTable query latency for P predicates and N records. 75

8 Comparison of baselines based on security guarantees (oblivious), setup (no trusted proxy), and performance properties (high throughput and throughput scales). 103

2 HSM 135
7 Microbenchmarks on SoloKey. Pairing is on BLS12-381 curve using the JEDI library [299]. Other public-key operations use NIST P256 curve. 153

6 Costs for larch with FIDO2, TOTP (20 relying parties), and passwords (128 relying parties). We take the cost of one core on a c5 instance to be $0.0425-$0.085/hour (depending on instance size) and data transfer out of AWS to cost $0.05-$0.09/GB (depending on amount of data transferred) [2]. For comparison, the Argon2 password hash function should take 0.5s using 2 cores. 195

Acknowledgments

I am grateful for the many wonderful people I had the privilege of working and interacting with during my time at Berkeley. Thank you all for a great five years.

This thesis would not be possible without my advisors, Raluca Ada Popa and Ion Stoica.

Raluca Ada Popa has been a model of what excellent research looks like throughout my PhD. She gave me both the guidance and freedom to grow as a researcher, and she helped me learn to think through and clearly articulate why certain research problems are important to solve. Throughout my PhD, I could trust that Raluca would always be a strong advocate for me, whether that was encouraging me to apply to a fellowship, connecting me to people in industry, or guiding me through the job search process.

Ion Stoica has continually encouraged me to step back and consider the big picture, both in choosing research problems and considering next steps in my career. He is excellent at identifying and helping me focus on the most important elements at any stage in a project.

I have also been incredibly fortunate to work with Henry Corrigan-Gibbs and David Mazières during my time as an undergraduate, master's, and PhD student. The experience of working on my first research project with them, along with Dan Boneh and Dom Rizzo, sparked my interest in research and led me to apply to PhD programs. They were also a part of two projects in this thesis (SafetyPin and larch). Henry has been a role model in research since we started working together—his thoughtful feedback and attention to detail shaped how I approach each step of the research process, and his advice has been invaluable over the years. David has helped me develop a more ambitious taste in research problems, and his comments on my writing and talks have helped me become a better communicator. They have been very generous with their time and provided excellent guidance throughout my PhD, particularly during the job search process.

I was also lucky to work with Natacha Crooks on two projects, one of which is a part of this thesis. Natacha brought a deep understanding of systems building and a dry sense of humor to our project meetings. She was also a great source of advice throughout the PhD, and I enjoyed sitting in an office right next to hers.

I am grateful to faculty who were a part of my qualifying exam and thesis committees and gave me valuable feedback: Raluca Ada Popa, Ion Stoica, Henry Corrigan-Gibbs, Natacha Crooks, and David Wagner.

Throughout my PhD, I have been lucky to work with an amazing group of collaborators, in addition to those listed above: Weikeng Chen, Alessandro Chiesa, Graeme Connell, Ioannis Demertzis, Vivian Fang, Eric Feng, Alexandra Henzinger, Allison Li, Danny Lin, Ellen Luo, Deevashwer Rathee, Mayank Rathee, Rolfe Schmidt, Miles Wada, Nicholas Ward, and Nickolai Zeldovich. I learned a great deal from working with you all.

Spending time with the students in the Sky/RISE labs, 719/721, and elsewhere has been a highlight of my PhD. Vivian Fang, I am grateful that we started our PhDs together; it's been quite a journey, from long days (and nights) before paper deadlines to seeing tulips in Amsterdam together. Mayank Rathee, I learned a lot from collaborating with you, and I'll miss being able to stop by your desk to hear about the cryptographic problems that you're thinking about or listen to a funny story. Alexandra Henzinger, I've admired your persistence and determination throughout our work

together, and traveling in Japan before Real World Crypto was a lot of fun. Deevashwer Rathee, I'm glad that we've had the chance to start working together, and I'm looking forward to seeing where this project takes us. Sam Kumar, Mae Milano, and Stephanie Wang, thank you for the advice and support throughout the job search process. Many other students helped make my time at Berkeley memorable: Weikeng Chen, Tess Despres, Darya Kaviani, Yuncong Hu, Zhanhao Hu, Norman Mu, Michah Murray, Shishir Patil, Julien Piet, Conor Power, Rishabh Poddar, Chawin Sitwarin, Sijun Tan, Jean-Luc Watson, Samyu Yagati, Wenting Zheng, and Jinhao Zhu. Working in the lab with all of you was never boring.

The RISE, Sky, and EECS staff have helped me navigate the department and helped give me the freedom to focus on my research. I am grateful to Kattt Atchley, Shane Knapp, Jon Kuroda, Jean Nguyen, Ivan Ortega, Dave Shonenberg, Shirley Salanio, Kailee Truong, and Boban Zarkovich, among others.

I don't know if I would have made it through the PhD without the support of my family, friends, and my husband, Louis. My parents, Kent and Catherine Dauterman, and brother, Gordon Dauterman, have always been there for me. Driving to see them in Oregon has been a great way to regroup during my PhD. Barbara and Adam Callaway and Bill, Alice, and Kurt Dauterman have also been a source of support. When I married Louis, I was fortunate to get wonderful in-laws, including Michael, Renee, Becca, and Josh Lafair (and soon, Scott Breece). I cannot imagine my PhD without my husband and best friend, Louis Lafair. Louis has been there for me through the highs and lows of the PhD and believed in me even when I did not believe in myself.

* * *

I am grateful to have been supported by a National Science Foundation Graduate Research Fellowship and a Microsoft Ada Lovelace Research Fellowship. During my PhD, I was a member of the RISE and Sky labs, which received support from NSF CISE Expeditions Award CCF-1730628, the Sloan Foundation, Accenture, Alibaba, AMD, Amazon, Amazon Web Services, Ant Group, Anyscale, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, MBZUAI, Nvidia, Samsung SDS, SAP, Scotiabank, Splunk, Uber, and VMware.

Chapter 1

Introduction

Today's computer systems handle massive amounts of user data, from health records to social security numbers to search queries. All of this information is a valuable target for attackers.

In order to protect this data from attackers, many systems deployed today provide security by hardening a few key system components. These components could be application servers, databases, or even secure hardware devices. As long as an attacker cannot break into one of these components, then the attacker cannot steal user data. In this model, the security of the entire system can reduce to the security of a single component.

However, past experience shows that we cannot rely on completely trustworthy components in order to build secure systems. An attacker might exploit a software vulnerability, launch a phishing attack, or even compromise hardware in order to breach a critical system component. For example, in the 2023 LastPass data breach, attackers broke into an engineer's personal computer by exploiting vulnerable software [514]. By compromising that machine, attackers were ultimately able to steal password information for many users. Human error also allows attackers to gain access to sensitive data, with an estimated 68% of data breaches involving a human element such as stolen credentials [501]. And even hardware designed to resist attack is not immune: researchers have found vulnerabilities in secure enclaves [401, 495, 520], hardware security modules [190], and trusted platform modules [209].

Attackers often compromise critical system components in order to steal user data and profit. In 2023 alone, there were 3,122 data breaches affecting roughly 349M individuals [259].

1.1 Approach

In this thesis, we take a different approach: systems should provide strong security even if some individual components are insecure. This design principle ensures that the systems we build do not have single points of security failure. In other words, an attacker that compromises some but not all system components cannot steal user data.

While existing general-purpose cryptographic tools can realize this approach in theory [205, 525], the challenge is to build systems that are concretely efficient and frictionless to use in practice.

System	Functionality properties	Security & privacy properties	Performance properties
DORY	Keyword search for end-to-end encrypted filesystems	Hides queries, data, and search access patterns from attacker controlling 1 of 2 servers	Outperforms oblivious RAM baseline
Waldo	Time-series analytics queries	Hides query filter values, data, and search access patterns from attacker controlling 1 of 3 servers	Outperforms oblivious RAM and multi-party computation baselines
Snoopy	Object store	Oblivious (hides access patterns from servers) using secure enclaves	Horizontally scalable
SafetyPin	Data backups with short (6-digit) PINs	Protects backups from compromise of a fraction of hardware security modules	Horizontally scalable
Larch	Every authentication is correctly logged; compatible with existing web servers	Hides credentials and authentication records from log service	Adds $\leq$ 150ms at authentication time

Table 1: Summary of thesis work. Functionality, security and privacy, and performance properties are described informally at a high level.

In particular, these systems should respect the constraints and expectations of existing software systems, hardware, and users.

To achieve this, we use a co-design of cryptography and systems techniques. The key idea is to use cryptographic tools that provide only what is necessary and to take advantage of the system model to reduce costs. This approach allows us to improve performance in comparison to general-purpose solutions.

This thesis focuses on protecting two sensitive user assets:

- **Queries (Part I).** Users want to store their data encrypted at a server, but still retain the ability to query their data. The technical challenge is enabling servers to execute a query without learning any information, in a cryptographic sense, about the contents of the query or the outsourced data.
- **Accounts (Part II).** User accounts often safeguard personal data and sensitive actions (e.g., wire transfers or publishing data). The technical challenge is essentially protecting the user from a compromised or lost client device without creating a single point of security failure in the system.

1.2 Private queries

A substantial body of prior work has explored how to execute private queries on encrypted data [72, 411, 464]. However, many existing constructions achieve good performance at the cost of privacy: how the server accesses memory reveals some information about the query and/or the data [269]. For example, even if the contents of a medical database are encrypted, if an attacker sees which parts of the medical database are accessed for each query, this could reveal information about patient medical conditions. The systems we built achieve good concrete efficiency without sacrificing privacy—informally, the attacker learns no information about the data or query from watching the server access memory in response to a query.

This thesis contributes techniques for hiding three classes of queries: keyword queries in DORY (Chapter 2), time-series analytics queries in Waldo (Chapter 3), and object-store queries in Snoopy (Chapter 4). While these three workloads seem quite different, the systems we built all achieve strong privacy guarantees by employing the same design idea: scanning over all the data is asymptotically slow, but, for some applications, can be concretely efficient with the right cryptographic tools.

1.2.1 DORY: a private search system for end-to-end encrypted filesystems

Users can protect their files by using an end-to-end encrypted filesystem: the client encrypts data before storing it a storage server so that even if an attacker compromises the server, it does not learn the contents of user files. However, users still expect to be able to search over their data. One class of solutions for encrypted keyword search based on searchable encryption [72, 411, 464] achieves good efficiency at the expense of strong privacy guarantees [269]: how these constructions access memory and the number of results can reveal private information. Another approach based on oblivious RAM (ORAM) [206, 473] provides strong privacy, but incurs high costs for encrypted keyword search.

DORY shows that it is possible to have encrypted keyword search with both strong privacy and good concrete efficiency [140]. In addition, we show that, for expected workloads, our system has lower concrete overheads than an asymptotically faster ORAM-based solution. Achieving strong privacy with good concrete efficiency in DORY requires distributing trust between two servers; if at least one server is honest, then the attacker learns no private information. DORY is a step towards ensuring that users do not need to choose between privacy and functionality. Users can have the strong privacy of end-to-end encrypted filesystems while being able to search over their documents.

We built a prototype and show that it performs orders of magnitude better than a baseline built on ORAM. Parallelized across 8 servers, each with 16 CPUs, DORY takes 116ms to search roughly 50K documents and 862ms to search over roughly 1M documents. We describe DORY's design and evaluation in more detail in Chapter 2.

Technical challenges and contributions. The first challenge was identifying a cryptographic primitive that allowed us to hide access patterns, but did not incur the overheads of ORAM for encrypted search. By distributing trust, we were able to use a distributed point function (DPF) [78,201] for the task of private information retrieval [112]. Evaluating a DPF over a search index incurs costs

linear in the index size (rather than logarithmic, as in ORAM), but this scan be concretely efficient because it only relies on AES evaluations, which are fast in hardware.

Because evaluating a DPF requires scanning over the entire search index, the next challenge was to design a compact index that permits concretely efficient updates and searches. The key idea is to build a table of bits where the ith row corresponds to a bitmap of words for document i. This way, updates simply require writing a row, and searches require privately retrieving a column with a DPF. To reduce the cost of searches, which require a linear scan over the index, we use Bloom filters to compress document contents without losing column alignment.

Another challenge was to protect the privacy and integrity of the search index contents without substantially increasing the index size and thus the cost of searches. Authenticated encryption would provide the necessary properties, but applied naïvely, would require encrypting and authenticating every bit of the search index individually, as updates and searches are performed along different axes of the table. This approach would dramatically increase the index size. Instead, we show how to use a random mask to protect privacy while keeping the index compact, and we leverage aggregate MACs in order to provide integrity at a low cost [280].

Finally, distributing trust across servers naturally requires more resources, especially when we consider that each server needs to be replicated. We show how to provide low-overhead fault tolerance by taking advantage of the properties of our search index.

1.2.2 Waldo: a private time-series database

End-to-end encrypted filesystems are only one setting where users and organizations need to outsource sensitive data to servers and then later query it. For example, organizations store and query time-series data, such as data produced by remote patient monitoring systems and smart homes. In these applications, users and organizations need to hide data and query parameters from attackers. For example, a doctor's query to a remote patient monitoring system could reveal information about a patient's medical condition. ORAM and multi-party computation (MPC) [205, 525] are natural tools for providing strong privacy and protecting access patterns, but they incur high overheads in the time-series setting.

We built Waldo, a system that provides strong privacy for time-series analytics queries [142]. Like DORY, Waldo also splits trust, but it requires three servers and provides strong security and privacy if an attacker compromises at most one of the servers. Waldo offers a path towards allowing users and organizations to protect their privacy while still retaining the ability to extract valuable insights from outsourced data.

We implemented Waldo and show performance improvements over ORAM and MPC baselines. With 32-core machines and features modulo 256, Waldo runs a query with 8 range predicates for 2^{10} records in 0.22s and 2^{20} records in 11.82s. We include more details about Waldo's design and performance in Chapter 3.

Technical challenges and contributions. The high-level challenge in Waldo was extending the core design principles in DORY to aggregation queries with multi-predicate filtering while keeping concrete costs low. Waldo takes advantage of function secret sharing (FSS), which is a generalization

of DPFs (used in DORY) that support more functionality [78]. While existing FSS constructions offer a mechanism for filtering based on equality and range predicates, there is still a large gap between what FSS provides and the properties that we want Waldo to achieve. While FSS is designed for honest-but-curious servers with public inputs, Waldo needs to support malicious servers, secret inputs, and chained predicates.

The first challenge was how to filter when both the query filter values and data should remain private. FSS correctness requires that both servers have the same inputs, but providing the inputs in plaintext would violate privacy. To solve this problem, we developed a shared one-hot index that provides strong privacy while enabling appends and queries. To protect the contents of the index, we use replicated secret sharing [31] with FSS, an approach that is inspired by Bunn et al. [82]. The structure of the shared one-hot index together with the replicated secret sharing make it possible to combine multiple predicates using Boolean operators via existing techniques for three-party multiplication [31, 479].

The shared one-hot index allows us to filter based on a combination of multiple predicates, but the aggregation operations that it supports are restricted to count, sum, and, by extension, mean, variance, and standard deviation. To support more complex aggregates, we constructed a shared aggregate tree, which only supports aggregation over time, but makes it possible to compute functions like min, max, and top-k.

Finally, we needed to provide integrity against a malicious attacker that can cause a server to behave arbitrarily. This requires linking the FSS queries sent by the client to the final result. To do this, we draw inspiration from prior work on information-theoretic MACs for authenticating FSS outputs [75], and we show how to securely combine FSS outputs while minimizing the verification work done by the client: the client only needs to check the integrity of the final result and a random linear combination of intermediate results.

1.2.3 Snoopy: an oblivious, scalable object store

Many applications managing sensitive data rely on object stores. In some applications, such as medical databases, not only are the object contents sensitive, but also the access patterns made by the application. Oblivious object stores hide access patterns from attackers observing external storage [206]. However, many existing constructions have scalability bottlenecks [63, 97, 128, 206, 361, 432, 442, 471–473], which are a critical barrier to deployment. A scalability bottleneck, a single point in the system that all requests must pass through, limits the ability of the system to handle more requests.

We showed that it is possible to build an object store that is both oblivious and scalable in our system Snoopy [139]. Vivian Fang is a co-first-author on this project. In order to scale in the cloud and support multiple clients, Snoopy leverages secure enclaves. (As enclaves have known limitations, we explored how to ensure that secure hardware is not a single point of security failure in an encrypted backup system in Chapter 5).

We built a Snoopy prototype, and it achieves 13.7× higher throughput than Obladi [128], the prior state-of-the-art high-throughput oblivious object store. With two million 160-byte objects, Obladi reaches a max throughput of 6.7K requests/s with a proxy machine and server machine,

whereas Snoopy can use 18 machines to scale to 92K requests/s with average latency under 500ms. Snoopy makes it possible to support oblivious object-store queries in high-throughput applications. For example, the end-to-end encrypted messaging application Signal leverages techniques from Snoopy in their private contact discovery system (see Section 1.4). We describe Snoopy's design and how it scales in more detail in Chapter 4.

Technical challenges and contributions. To illustrate why horizontal scaling is hard, consider two common characteristics of existing oblivious storage systems. First, they often have some sort of dynamic mapping of identifier to location, and this mapping must be checked and updated after every access, creating a scalability bottleneck. Second, they are often based on a hierarchical or tree-like structure for efficiency, and this leads to a scalability bottleneck at the root [472, 473].

To avoid these pitfalls, we based the design of Snoopy on a balls-into-bins argument: if there are enough requests ("balls"), then we can distribute them across shards ("bins") in a way that hides information about the request distribution. This insight makes it possible to scale across shards, but introduces new challenges. We needed techniques for generating batches of requests obliviously, as well as a mechanism for efficiently executing batches of requests at each shard.

The first challenge was to group requests by shard without revealing any information about the request contents. To do this, we ensure that the number of requests is based only on public information (not the underlying request distribution), and we guarantee that, with high probability, no requests are dropped. We also use an oblivious algorithm in order to hide the mapping of incoming requests to shards. Notably, we can add more machines that perform this request grouping without additional coordination.

Next, we needed to execute large batches of requests at individual shards. We could have used an existing oblivious storage system [361] as a shard, but because we are executing a large batch of requests instead of individual requests, we tailored the design to our setting. We use a linear scan over the contents of the data, which we show can be concretely efficient for some workloads when amortized over many requests.

1.3 Secure accounts

Account security is critical for users and organizations, but often remains a weak link: roughly 68% of data breaches involve a human element (e.g., stolen credentials) [501]. In our systems, we explored how to protect users from themselves: if a user's device is lost or compromised, users should still have access to their accounts and strong security. Moreover, providing these guarantees should not require trusting some server or hardware security module (HSM).

In this thesis, we present an encrypted backup system SafetyPin that uses HSMs without fully trusting them (Chapter 5) and an authentication logging system larch that protects user secrets from the logging service (Chapter 6). In both systems, we split user secrets across different entities to ensure that there is no central point of attack while respecting system constraints (e.g., hardware or protocol limitations).

1.3.1 SafetyPin: an encrypted backup system that resists hardware attacks

Users need to back up sensitive data (e.g., documents, photos, and messages) to application servers without the servers learning the contents of the data. Users should be able to recover this data even if they lose all of their devices and can only remember some short PIN. One strawman approach is to encrypt the user's backup using a cryptographic key derived from the user's PIN. However, short PINs are susceptible to dictionary attacks that allow a compromised application provider to recover user data. Industry has addressed this problem using secure hardware [296, 507, 515]. The secure hardware essentially rate-limits recovery attempts, which ensures that if an attacker can compromise the application servers but not the secure hardware, the attacker cannot run a dictionary attack. While this approach defends against a compromised application provider, it requires trusting the secure hardware with sensitive user secrets. If an attacker compromises just one secure hardware device, the attacker can recover many users' backups.

In SafetyPin, we showed that it is possible to have the benefits of secure hardware without completely trusting it [137]. SafetyPin defends against an attacker that can adaptively compromise some percent of the hardware security modules (HSMs) in the data centers.

We built a prototype with 100 $20 SoloKeys [463] to show that the cryptographic tools that we employ work well on hardware with limited compute, communication, and storage. A recovery in SafetyPin takes 1.01s, and we estimate that a SafetyPin deployment would need 3,100 SoloKeys to process 1B recoveries a year. We describe SafetyPin's design and the performance of our prototype in Chapter 5.

Technical challenges and contributions. The core challenge in the design of SafetyPin was to protect against an attacker that can adaptively compromise some percent of the HSMs while also allowing the system to scale to many recovery requests. In particular, we wanted to provide the security of splitting the client's secret across many HSMs with the scalability of splitting the secret across a few HSMs. The key idea to solve this problem is to use the user's PIN to hide the set of HSMs storing the user's secret. This PIN does not have much entropy, but if an attacker does not know the user's PIN, then it does not know which HSMs to break into, and each additional PIN guess essentially requires breaking into more HSMs. We introduce and formalize this primitive as location-hiding encryption.

Location-hiding encryption solves some problems, but also creates new challenges. In particular, different PIN attempts map to different HSMs, and so we needed a coordinated mechanism for tracking the number of recovery attempts remaining across all users and across all HSMs. For security, we could send every recovery request to every HSM, but this would not be scalable. Instead, we show how to build a distributed append-only log tailored to our setting where each HSM does a small amount of work, but with high probability, any misbehavior will be caught.

The other challenge comes at recovery time. An attacker that controls the network and watches a user recover can see which HSMs store part of a user's secret. After the user finishes recovering, an attacker can simply break into these HSMs and recover a user's secret, violating privacy. To address this problem, we allow HSMs to revoke their ability to decrypt. Puncturable encryption [215] provides the properties that we need, but is ill-suited to the resource constraints of HSMs. Therefore, we adapt it to the setting where HSMs have limited storage.

1.3.2 Larch: an authentication logging system without a single point of security failure

Single sign-on systems help simplify the problem of authentication: not only do they permit a user to remember just one password for the single sign-on service, but they also make it possible to keep a comprehensive record of every authentication. This record is useful for auditing—an honest user can see all authentications made by both the honest user and attackers. However, to keep this record, users need to trust the single sign-on service with their credentials and private information. If an attacker compromises the single sign-on service, they can access user accounts and learn information like where a user has accounts, how often they authenticate, and other sensitive data. For example, when attackers breached Okta servers at the end of 2023, they stole customer data and authentication tokens that allowed attackers to access linked accounts [193, 395].

In larch, we show that it is possible to have the benefits of authentication logging without creating a single point of security or privacy failure [141]. Larch ensures that a log service correctly records every authentication, whether it is made by an honest user or an attacker that has compromised a client device. At the same time, the log service does not have access to user secrets (i.e., credentials or the contents of the authentication logs). To ease the path the deployment, larch is compatible with web servers that support protocols like FIDO2 [179] (popularized by Yubikeys and Passkeys), TOTP [369] (popularized by apps like Google Authenticator), and password-based login.

We implemented and evaluated larch. For a client with four cores and a log server with eight cores, our split-secret authentication protocols take 150ms for FIDO2, 91ms for TOTP, and 74ms for passwords (excluding preprocessing, which takes 1.23s for TOTP). We describe the design of larch and evaluate its performance in Chapter 6.

Technical challenges and contributions. The technical challenge was to ensure that a client cannot authenticate without the log service receiving a valid, encrypted log record, but at the same time, the log service should learn no information about where the client is authenticating. Essentially, the log service should correctly record every authentication without actually seeing each authentication. To make it clear why this is hard, consider a simple strawman where the client authenticates with a web server and then sends an encrypted log record to the log service. While this approach ensures that the log service is not a single point of security or privacy failure and is backwards-compatible, it does not protect against a malicious client; a malicious client can send an incorrect log record, or no log record at all, and the log service cannot detect misbehavior.

We resolve this tension by constructing split-secret authentication protocols. When we authenticate today without larch, the client has some secret (this could be a cryptographic key or a password) that it uses to generate a credential and authenticate. In larch, we split the authentication secret between the client and log service, ensuring that neither becomes a single point of security or privacy failure. To authenticate, they must use their authentication secret shares to jointly run a new split-secret authentication protocol, which is essentially a special-purpose two-party computation [525]. As a result of this protocol, the client gets a credential that it can use to authenticate, and the log service gets a valid, encrypted log record. The protocol ensures that if the client misbehaves, causing the log service to receive an incorrect log record, then the misbehaving client cannot authenticate.

We designed, implemented, and evaluated split-secret authentication protocols for FIDO2, TOTP, and password-based login. Our FIDO2 protocol uses zero-knowledge proofs [208] along with a new, lightweight two-party ECDSA protocol tailored to our setting. Our TOTP protocol uses an existing garbled circuit protocol with malicious security [511]. Our password-based protocol allows the client to privately exchange a password for an encrypted log record using a discrete-log-based proof [218].

1.4 Impact and adoption

The Signal end-to-end encrypted messaging application uses some of our techniques from Snoopy to scale their private contact discovery system, which matches user contacts to Signal users [118]. In their private contact discovery system, the Signal servers should not learn a user's contacts, but also the user should not learn the entire set of Signal users. We helped Signal adapt some of our techniques for horizontally scaling oblivious object stores. The resulting system, which draws on techniques from several academic papers [361, 473] in addition to Snoopy, is deployed to Signal users. The release of the new system allowed Signal to reduce the number of dedicated servers from nearly 600 to around 10 [516].

For all of the systems in this thesis, we released open-source code. At conferences where artifact evaluation was available, our systems achieved the "Artifact Available", "Artifact Functional", and "Artifact Reproducible" badges.

Co-authored material

This dissertation is based, in part, on co-authored publications. In particular:

- Chapter 2 is based on “DORY: An Encrypted Search System with Distributed Trust” at OSDI’20 with Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica [140].
- Chapter 3 is based on “Waldo: A Private Time-Series Database from Function Secret Sharing” at IEEE S&P’22 with Mayank Rathee, Raluca Ada Popa, and Ion Stoica [142].
- Chapter 4 is based on “Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage” at SOSP’21 with Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa [139].
- Chapter 5 is based on “SafetyPin: Encrypted Backups with Human-Memorable Secrets” at OSDI’20 with Henry Corrigan-Gibbs and David Mazières [137].
- Chapter 6 is based on “Accountable Authentication with Privacy Protection: The Larch System for Universal Login” at OSDI’23 with Danny Lin, Henry Corrigan-Gibbs, and David Mazières [141].

Part I

Private queries

Chapter 2

DORY: A private search system for end-to-end encrypted filesystems

2.1 Introduction

Users have grown increasingly reliant on filesharing systems such as Box, Dropbox, and iCloud. However, attacks on storage servers [335, 374, 391, 431] have exfiltrated large amounts of sensitive data belonging to many users, jeopardizing user privacy as well as the reputation and business of the victim organizations. End-to-end encrypted storage systems [286, 417, 465, 476, 492] provide a strong defense against this type of attack: the client stores all cryptographic keys and the server receives only encrypted data, and so an attacker that compromises the server can only exfiltrate encrypted data.

At the same time, end-to-end encrypted filesharing services struggle to provide the same functionality as plaintext storage providers like Dropbox because the server cannot decrypt the data to process it. Server-side search is a critical tool that users expect for convenience and companies require for compliance.

Despite a large body of work on searchable encryption [95, 104, 129, 148–151, 192, 202, 274, 275, 376, 436, 464, 469], *practical and leakage-free* search on encrypted data has remained an unsolved problem for two decades. Existing work can largely be divided in two categories: (1) practical but leaking search access patterns, or (2) not leaking search access patterns but expensive.

In the first category, an attacker can learn sensitive data by observing search access patterns. We now explain what search access patterns are intuitively by contrasting them to the leakage already existing in deployed end-to-end encrypted filesystems [286, 417, 465, 476, 492]. In these filesystems, when a user accesses a file, the server learns that this specific user accessed that specific file, but it does not see the content due to end-to-end encryption. The concern with leaking search access patterns on top of this filesystem leakage is that search access patterns can leak information at the word level, allowing an attacker to potentially reconstruct search queries and document plaintext [94, 269, 284, 329, 410, 415, 534].

Consider a simple example of how an attacker can exploit search access patterns [534]. The

server stores an inverted search index for Alice's emails mapping an encrypted keyword to an encrypted list of files. The attacker sends a one-word email to Alice containing "flu". If Alice's client updates entry 924 of index on the server, the attacker learns that index[924] is for "flu". By repeating this process for every word in the dictionary, the attacker can discover the word corresponding to every index entry. Later when Alice receives a confidential email, the attacker can derive all the words in that email based on which index entries are updated. More sophisticated attacks can reconstruct both entire documents and search queries from even more advanced search schemes [94, 269, 284, 329, 410, 415, 534]. In this chapter, we informally define search access pattern leakage as the set of documents matching a search keyword, the size of that set, and any information about the search query. In contrast, if a scheme does not leak search access patterns, then during a search on a folder, the search server learns only that a search is now happening in that folder.

The second category of existing work typically relies on Oblivious RAM (ORAM) [206, 392, 473], a cryptographic tool that allows a client to read and write data from a server without revealing access patterns. Many academic works point to an inverted index inside ORAM as a straightforward way to eliminate leakage [254, 375, 469]. Unfortunately, even though the asymptotic complexity of ORAM is polylogarithmic in the index size, the cost of even the most practical ORAM schemes remains prohibitively expensive for our setting. For example, inserting a file requires an expensive ORAM operation for *every* keyword in that file (and there can be hundreds).

Given that practical, leakage-free search remains a difficult problem, we revisit the system model: What do real end-to-end encrypted filesharing systems actually require from a search system? Would the problem become more tractable in their system model?

Choosing a system model. We met with five companies that provide end-to-end encrypted filesharing, email, and/or chat services: Keybase [286], PreVeil [417], SpiderOak [465], Sync [476], and Tresorit [492]. To the best of our knowledge, this is the first study of requirements for encrypted search in real filesharing systems. We discuss our findings in Section 2.2 and summarize the ones most relevant to DORY here:

Efficiency requirements. These companies care about two primary metrics: latency and monetary cost. They are not concerned about the asymptotic complexity of the search algorithm and would accept an algorithm with runtime linear in the number of documents as long as their concrete performance and cost requirements are met (see Table 2).

Trust model requirements. Some of these companies were already splitting trust to back up secret keys or distribute public keys, and we wanted to know if we could leverage a similar distributed trust assumption to make the problem of encrypted search more tractable. While these companies were willing to split trust across multiple domains, some had two requirements aimed at strengthening the distributed trust assumptions. First, if at least one trust domain is honest, then an attacker that controls all the remaining trust domains and observes user queries should not learn search access patterns. In particular, we need to protect against a *malicious* attacker rather than an honest-but-curious one and should not assume that the attacker follows the protocol. The second requirement, stated intuitively, is that only search access patterns should be protected by distributed trust, and an attacker that compromises *all* trust domains should not immediately learn the contents of the search index.

While prior work explores some forms of distributing trust for encrypted search [54, 73, 174,

255, 266, 427], we are not aware of any work that meets both the efficiency and distributed trust requirements outlined above without leaking any search access patterns, as explained in Section 2.8.

Our system: DORY. We design and implement DORY (Decentralized Oblivious Retrieval sYstem), an encrypted-search system that splits trust to meet the real-world efficiency and trust requirements summarized above (and detailed in Section 2.2). DORY ensures that an attacker who cannot compromise every trust domain does not learn search access patterns.

We implemented and evaluated DORY to show that it performs better (for some metrics, orders of magnitude better) than an ORAM baseline (Section 2.7). DORY also meets the companies' efficiency requirements; parallelized across 8 servers, searching over 1M documents takes 862ms, and, using workload estimates from the companies, we estimate that DORY costs roughly $0.0509 per user per month.

DORY combines cryptographic and systems techniques to overcome the security and efficiency challenges of previous solutions.

2.1.1 Summary of techniques

Choosing an oblivious primitive. Given the inefficiencies of ORAM, a key challenge was choosing a cryptographic primitive for hiding search access patterns. We identified a relatively recent cryptographic tool, distributed point functions (DPFs) [201] (a specific type of function secret sharing [77, 78]), as particularly promising for our setting. DPFs allow us to leverage ℓ servers (for practical constructions, $\ell = 2$) to retrieve part of the search index without any group of $< \ell$ servers learning which part of the index we're retrieving (the problem of private information retrieval, or PIR [112, 113]). A DPF-based solution requires a linear scan over the index, but the overhead per index entry is small because it relies on AES evaluations, which are implemented efficiently in hardware.

Designing the search index. An important challenge is how to structure the search index to support efficient search and update operations. To minimize the overhead of updating the search index when a file is uploaded, the client should only need to upload a small amount of data per file, and ideally avoid performing an expensive cryptographic operation for every keyword in that file. To minimize search overhead, we need to limit the number of DPF queries. To achieve both of these goals, we keep a table where each row corresponds to a bitmap of words for a document. An update simply requires the client to insert a *row* by uploading a new bitmap, and a search only requires a single DPF request to retrieve the *column* corresponding to a keyword (Section 2.4.1). However, this bitmap can become quite large to accommodate every word in the dictionary. To reduce the size of this bitmap (and thus the time for the linear scan), we use a Bloom filter, which provides compression while preserving column alignment. Bandwidth from the servers to the client is linear in the number of files searched over, but the bandwidth from a server to the client is less than 1 byte per file (Section 2.7) and, more importantly, this fixed bandwidth enables DORY to hide the number of search results, which can be exploited in volume-based attacks [94, 284, 410].

Encrypting the search index. To prevent an attacker that compromises all the servers from immediately reconstructing the plaintext search index, we need to encrypt each bit in the Bloom

filter before inserting it into the search index. Unfortunately, the expansion of encryption would increase the size of the search index (and thus the time for the linear scan) by the security parameter. To ensure that the encrypted index is the same size as the plaintext index, we instead mask the bits using a random one-time pad that we ensure is unique for each version of the file (Section 2.4.1).

Defending against a malicious attacker. DPFs do not protect against malicious attackers. To protect against a malicious attacker that compromises all but one of the trust domains, we leverage MACs to allow the client to check the integrity of search results in a way that makes blackbox use of DPFs. Applied naively, adding MACs would increase the search bandwidth and storage at the server by a factor of the security parameter. To address this problem, we employ aggregate MACs [281] to change the dependence on the security parameter from multiplicative to additive (Section 2.4.3).

Providing fault tolerance. Splitting trust across different trust domains naturally requires additional servers. With secret-sharing, one tool for distributing trust, servers store different data that they may not share. Then, to provide fault tolerance, each of these servers would need to be replicated. We observe that in DORY, servers can use each other for fault-tolerance even though they are in *different* trust domains due to two properties (Section 2.5): (1) each server has an identical copy of the state, and (2) the client can perform integrity checks.

Reducing the cost of replication. To execute a search query correctly, all the servers must operate on the same version of the state. This is challenging because clients can issue update and search requests concurrently. One possibility is to use standard Byzantine fault-tolerant consensus techniques to solve this problem, but this would require $3f+1$ trust domains to handle f failures. Instead, we observe (1) the ways in which our system setting is less demanding than that of BFT, and (2) that our cryptographic protocol enables clients to check integrity even if all servers are compromised; using these, DORY only needs $f+1$ trust domains (Section 2.5).

2.2 Identifying a system model

To understand real-world use cases, we met with five companies providing end-to-end encrypted file storage, email, and/or chat solutions: Keybase [286], PreVeil[1] [417], SpiderOak [465], Sync [476], and Tresorit [492]. For each company, we asked a set of questions over the course of discussion(s) and email exchanges while we were in the process of designing our system. We summarize our findings in Tables 1 and 2 and in the following sections, and we outline our prepared questions in Section 2.9.3. We report statistics in aggregate to preserve the confidentiality of individual companies, as they requested. These statistics and requirements motivate DORY's system model.

About the companies. We now give a brief background about each company. Keybase [286], founded in 2014 in the US and recently acquired by the video-conferencing company Zoom [529], keeps a publicly auditable key directory and offers open-source, end-to-end encrypted chat and storage systems. PreVeil [417], founded in 2015 in the US, focuses on both encrypted chat and storage solutions and open-sources some of its tools. SpiderOak [465], founded in 2007 in the US, offers encrypted storage, backup, and messaging solutions leveraging a private blockchain

[1] One of the authors was employed at PreVeil during this project.

	Keybase	PreVeil	SpiderOak	Sync	Tresorit
Need server search?	✓	✓	✓	✓	✓
Have server search?	✗	✗	✗	✗	✗
File sharing?	✓	✓	✓	✓	✓
Email?	✗	✓	✗	✗	✗
Chat?	✓	✗	✓	✗	✗
Mobile client?	✓	✓	✓	✗	✓

Table 1: The search use-cases for each of the five companies.

System cost and scale	
Average number of docs/user	100 - 45K
Maximum number of docs/user	100K - 1.3M
Price/month/user	$0-20
Search requirements	
Maximum added $/month/user	$0.70-5.54
Maximum search latencies (s)	[0.5, 1, 1, 4]
Estimated update/search ratio	50/50

Table 2: Survey statistics. In accordance with the companies' confidentiality wishes, we report most fields in aggregate although we report individual responses for maximum permissible search latency (only 4 of the companies responded).

and open-sources many of its tools. Sync [476], founded in 2011 in Canada, and Tresorit [492], founded in 2011 in Switzerland, both provide encrypted storage. With the exception of Keybase, these companies generally target enterprise customers and support compliance with regulations such as GDPR or CMMC. Some of these companies report over 750K users in over 180 countries.

The need for server-side search. Every company expressed a need for server-side search on encrypted data either for their desktop client in cases where users do not have all the files downloaded, or for the mobile or web clients. However, none currently support server-side search; they all told us that they tried at some point to develop a solution (most had researched the academic literature), but their efforts were eventually thwarted by concerns about performance or search access patterns. Several of the companies had built or used a client index as a temporary solution, but they did not see this as a long-term solution because of its inability to index many files locally (e.g. enterprise data) or its resource consumption (especially on mobile). In Section 2.7.5, we discuss how synchronization between clients makes this solution infeasible in cases where documents are constantly updated.

They all stated interest in deploying a server-side solution that met their functionality, security, and performance requirements, if such a solution were to exist.

2.2.1 System requirements

Search must be responsive. The companies reported maximum search latencies between 500ms and 4s (Table 2). The company that reported a maximum search latency of 500ms reported tens of thousands to hundreds of thousands documents per user, while some of the companies that reported larger maximum search latencies had users with approximately a million documents.

Monetary cost for search must be small. These companies prioritize keeping the cost of search below $0.70 per user per month in order to make it feasible to deploy search to all users without increasing prices (Table 2). While some companies were willing to consider charging more for the ability to search, other companies believed that users would be unwilling to pay extra because they are used to free search on other platforms.

Multiple users must be able to update and search the same documents. Each company allows multiple users to access the same file. Therefore, a search solution should be designed with multiple clients in mind and minimize the amount of state clients need to synchronize between operations.

Revoking a user's access must be cheap. All these companies implement revocation lazily [39, 186, 203, 216, 273, 435], meaning that when a user's access to a folder is revoked, the remaining users generate a new key and, rather than re-encrypting every document in the folder under the new key, simply use the new key for subsequent updates. In this way, the revoked user can still access documents that haven't been updated since the time of revocation. These companies want to adopt a similar approach for search. When a user is revoked, rather than re-computing the entire search index (as in ORAM-based solutions), subsequent updates should not allow the revoked user to search over the updated documents.

Relaxations. In addition to learning requirements, we also learned several system relaxations these companies accepted. The companies did not require search results to be fresh (they could be stale for up to a few minutes), and they were also willing to accept a small number of false positives (several other search schemes have also leveraged this allowance [54, 202]).

2.2.2 Distributed trust requirements

The majority of prior encrypted search work considers a single-server model where the attacker can take control of the entire system. As some of these companies were already leveraging distributed trust (e.g. Keybase to distribute public keys via social media servers, PreVeil to backup secret keys secret-shared among multiple clients), we wanted to know if they were willing to accept a distributed trust model for encrypted search as well, as this could be an opportunity for providing a more efficient search. We found that all the companies were open to a distributed trust model, although several companies had more specific requirements for how to distribute trust:

Hide search access patterns even with only one honest trust domain. These companies wanted the guarantee that if at least one trust domain is honest, then an attacker cannot learn search access patterns. They did not want to assume that other trust domains behaved correctly, so they wanted a malicious threat model rather than an honest-but-curious one.

Distributed trust only for search access patterns. These companies wanted to limit the damage caused by an attacker who compromises *all* the ℓ trust domains by ensuring that putting the ℓ search indices together does not readily provide the attacker with the plaintext search index. For example, if a company is subpoenaed and every trust domain must hand over its search index and search access patterns from then on, the company could potentially choose to suspend search services to protect users' privacy by reducing search access pattern leakage, similar to the case where Lavabit chose to suspend operation rather than reveal Snowden's emails [19]. In such a case, reconstructing the index from the ℓ servers' index shares should result in end-to-end encrypted data. This requirement rules out solutions based on secret-sharing a plaintext search index across multiple servers because an attacker compromising all trust domains can recover the plaintext index.

2.2.3 Opportunities

From the information we learned, we summarize what we considered opportunities to make the problem of encrypted search easier:

- Performing a linear scan to search is feasible if the response time and the cost on expected workloads are acceptable.
- Distributing trust across multiple trust domains is acceptable if certain security requirements are met.

These opportunities serve as the basis for our system design.

2.2.4 Building a distributed trust system

We now discuss how to build a system where an attacker who compromises part of the infrastructure cannot easily gain access to the entire infrastructure. Such a model has already been deployed in several real systems, including cryptocurrencies relying on consensus such as Ripple [339] or Stellar [333], Certificate Transparency [309], and academic work [121].

Split across clouds. By treating different clouds as distinct trust domains, a malicious cloud provider (or an attacker that can exploit a vulnerability in one cloud infrastructure), cannot gain access to both trust domains.

Split across institutions. By using trust domains in competing organizations or nonprofits generally trusted by the public (e.g., the Electronic Frontier Foundation), users can have a stronger assurance that the organizations are unlikely to collude.

Split across jurisdictions. By separating trust domains by jurisdiction (i.e. different countries), a single legal authority cannot gain access to the entire system.

If the trust domains are deployed in the cloud, we can take advantage of the fact that cloud providers are monetarily incentivized to provide availability. Fail stops can still occur naturally, but cloud providers make it easy to detect failures and launch new servers. Clients can report statistics on the lack of availability of a trust domain, and the organization deploying the system can take its business elsewhere.

2.2.5 Future directions

Some companies mentioned additional features that, while not necessary for initial deployment, are desirable. Although we do not support these in DORY, we note them here as potential directions for future work.

Concentrate resources in a single trust domain. The trust domain already used for the filesystem should do most of the work for search as well. Each additional trust domain should do little work, so that adding a new trust domain should be cheap. DORY concentrates resources to some extent, (Section 2.5), but, as discussed in Section 2.4, still requires a server in each trust domain to perform a linear scan.

Richer search functionality. Several companies mentioned that they would appreciate richer search functionality beyond simple keyword search (e.g. ranked search based on term frequency.) DORY only returns the set of documents containing a keyword, leaving ranked search for future work.

2.3 System design overview

In DORY, we focus only on the search system for end-to-end encrypted filesharing systems and not on the design of these filesharing systems. These systems [286, 417, 465, 476, 492] already exist and are in use. We design DORY to build on top of and interface with these systems as described in Section 2.3.2. For this purpose, we abstract out the underlying filesystem.

2.3.1 The underlying filesystem

End-to-end encrypted filesystems (including the five companies we surveyed in Section 2.2) tend to follow a common design pattern, which we now describe. To hide the contents (including the name) of documents, these filesystems assign a document ID to each document and associate the ID with an encryption of the document contents. Documents accessible by the same users are grouped into *folders*, each of which has a corresponding ID. Users who have access to the same folder share a (logical) secret key used to encrypt the documents in that folder. In this way, while the server learns the IDs of documents being accessed, the number of documents in each folder, and which users have access to which folders, it does not see the contents of the documents.

When a user is added to a folder, the other users share the existing folder key with the new user, and when a user's access to a folder is revoked, the remaining clients choose a new folder key. To prevent the remaining clients from having to re-encrypt every document in the folder after a user is revoked, these systems employ lazy revocation (as described in Section 2.2.1).

Users may choose to keep some documents synchronized with the server (i.e., store the most recent version of the document locally) and others not synchronized (i.e., do not store locally and retrieve them from the server only as needed). In either case, the user has already downloaded the most recent version of the document before she sends an update. In the case where two clients try to update the same file simultaneously, these systems often create two versions of a file.

DORY integrates with the filesystem (FS) using the following FS API (depicted in Figure 3):

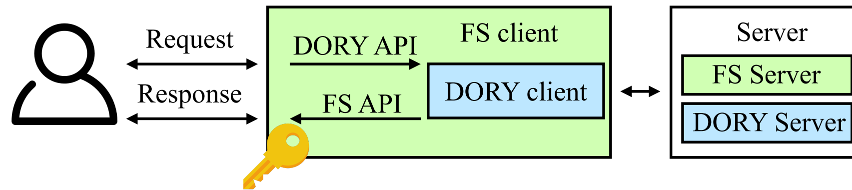


Figure 3: System software architecture. The figure shows the structure of the software rather than the physical system itself, where the server is instantiated across multiple machines.

- $\mathsf{getCurrKey}(\mathsf{folderID}) \to k$: Get the current key associated with the group of files in folderID.
- $\mathsf{getDocKey}(\mathsf{docID}) \to k$: Get the key used in the most recent update for docID.
- $\mathsf{getDocIDs}(\mathsf{folderID}) \to \mathsf{docIDs}$: Get all the document IDs used for the documents in folderID.
- $\mathsf{getVersion}(\mathsf{folderID}, \mathsf{docID}) \to \mathsf{version}$: Get the current version number associated with a file.

2.3.2 The DORY API

When a user searches or updates a file, the filesystem client calls the DORY client via DORY's API so that DORY performs the search or incorporates new updates into the search index. We now describe DORY's client API, depicted in Figure 3.

When the user updates a document in the underlying filesystem, the user's client also sends an update to the DORY client to maintain the search index, allowing DORY servers to respond to subsequent search queries correctly.

The underlying filesystem already handles key management by giving permitted users access to the folder key(s). DORY leverages this key management mechanism so the permissions of the filesystem naturally extend to DORY: when a user is added to or removed from a folder in the underlying filesystem, she also gains or loses the ability to search in DORY.

We also utilize the fact that to update a document in the underlying filesystem, the user has already downloaded that document (if it is not being added for the first time). We employ the conflict-resolution mechanisms in the underlying filesystem to resolve conflicts in search index updates.

DORY exposes the following API to filesystem clients:

- $\mathsf{Update}(\mathsf{folderID}, \mathsf{docID}, \mathsf{prevWords}, \mathsf{currWords})$: Given the folder ID, the document ID of a document in that folder, the previous set of keywords in that document prevWords, and the current set of keywords in that document currWords, update the state at the DORY servers.
- $\mathsf{Search}(\mathsf{folderID}, \mathsf{keyword}) \to \mathsf{docIDs}$: Given the folder ID to search over and a keyword, find all the documents containing that keyword. DORY has a small (configurable) false positive rate, but DORY has no false negatives.

Updates require the client to upload a small, constant-sized amount of data per file, and searches require the server to perform a linear scan over the search index for a given folder (the cost of search for a user only depends on the number of files that user has access to).

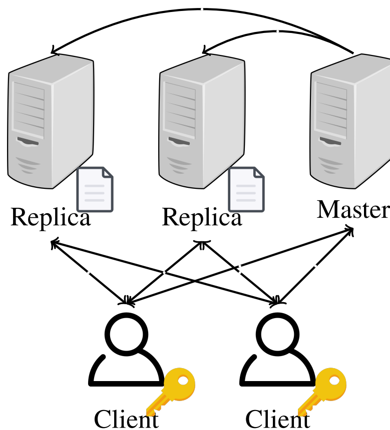


Figure 4: DORY's physical system architecture for a single partition (filesystem server not pictured). Replicas should be deployed in different trust domains, and each holds a copy of the search index.

2.3.3 System architecture

Folders in DORY are divided into *partitions*, each of which is managed by a different group of servers. A deployed system may contain many such partitions, and execution across partitions occurs in parallel. The following entities comprise DORY's system architecture for a single partition (Figure 4):

- **Filesystem server:** The underlying filesystem provides the functionality described in Section 2.3.1.
- **Replicas:** The ℓ DORY replicas maintain identical copies of the search index and execute search queries. Each replica is deployed in a separate trust domain. In our implementation, we use $\ell = 2$.
- **Master:** The DORY master ensures that the ℓ replicas have the same view of the state and that the clients know the version of this state and which servers to contact. The master can be deployed in any existing trust domain.
- **Clients:** Multiple clients send requests to the filesystem server and the DORY master and replicas. Each client only needs to store three 128-bit keys (and can optionally cache version numbers received from the master).

To search, the client must interact with ℓ replicas for each partition. The master can be co-located with the filesystem server to ensure that updates to the search system and underlying filesystem occur atomically, although this is not necessary.

2.3.4 Threat model and security properties

We now describe DORY's security properties at a high level and delegate DORY's formalism (detailing the guarantees) and proof to Section 2.9. In short, we achieve the security goals in Section 2.2.2. We discuss security at the level of trust domains, each of which may deploy one or more servers.

Below, we assume that the underlying filesystem is maliciously secure. In particular, we assume that DORY's client can always retrieve the correct version number from the underlying filesystem. Providing such a guarantee (e.g., by detecting rollback and fork attacks in filesystems) is a well-studied line of work [42, 258, 278, 289, 320]. If the underlying filesystem only defends against an honest-but-curious attacker, though, DORY also only protects against such an attacker.

Security with one honest trust domain. A malicious attacker that compromises $\ell-1$ of the ℓ trust domains does not learn any search access patterns. More precisely, such an attacker learns nothing except what is leaked by the underlying filesystem, as well as the timing of individual search requests and the folders they take place over. This security property implies both forward privacy, the privacy of newly added files in the presence of previous queries, and backward privacy, the privacy of deleted files after deletion, as defined by Stefanov et al. [469]. Notably, we do not leak the number of search results; if leaked, this information could open the door to volume-based attacks [410] (parameters that determine result sizes are public).

Security with no honest trust domains. DORY's goal is to hide search access patterns when at least one trust domain is honest. When all trust domains are compromised, we have the modest goal of defaulting to the security of prior schemes leaking search access patterns, instead of readily losing all security by immediately exposing the search index. In this case, the only additional leakage (on top of what the attacker learns if at least one trust domain is honest) is a deterministic identifier for the keyword queried. In the security definition for our cryptographic protocol, we model the attacker as seeing queries only after the point of compromise; in reality, systems retain leakage (e.g. cache state) that increases the amount of information the attacker can access [226].

We formally model the end-to-end security guarantees of DORY for the case where at least one trust domain is honest and the case where no trust domains are honest by defining an ideal functionality $\mathcal{F}$ that specifies the behavior of an ideal system, capturing the properties discussed above. $\mathcal{F}$ further captures the fact that the client can verify the integrity of the result. In Section 2.9, we present a formal definition using $\mathcal{F}$ and prove the following theorem, which captures DORY's security:

Theorem 1. *Using the definitions in Section 2.9.1, DORY securely evaluates (with abort) the ideal functionality $\mathcal{F}$ when instantiated with a secure PRF, a secure aggregate MAC, a secure distributed point function, and a secure filesystem that implements the ideal filesystem functionality.*

DORY does not provide availability if any one trust domain refuses to provide service (see Section 2.2.4 for how cloud providers are monetarily incentivized to provide availability).

Relationship with underlying filesystem. DORY interfaces with deployed end-to-end encrypted filesystems (Section 2.3.1). These, as mentioned, allow the server to learn the ID of the file being accessed (but not its contents). While search itself is protected in DORY, some side effects of the search results are not: If, after seeing the search results, a user decides to open (and retrieve from the filesystem) a file in the results, an attacker could infer that the file matched the search. DORY does not address these side effects, but simply aims to not add any leakage to the overall system during search.

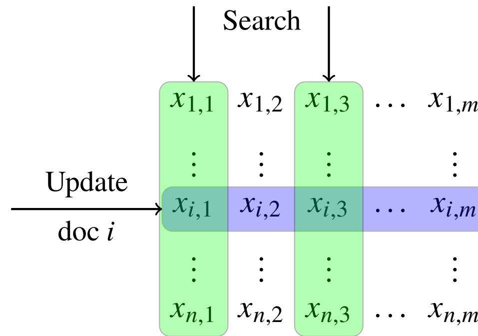


Figure 5: Search index layout for n documents with Bloom filters of length m. Updates write *rows* and searches retrieve *columns*.

2.4 Search design

We start by describing a basic encrypted search scheme that leaks search access patterns and is only secure against an honest-but-curious attacker in Section 2.4.1. We will show how to modify our basic scheme to eliminate search access patterns in Section 2.4.2, move from an honest-but-curious to malicious threat model in Section 2.4.3, and support dynamic membership in Section 2.4.4. We show the pseudocode for the complete search protocol in Section 2.4.5. For simplicity, we only discuss search servers, which we assume are deployed in different trust domains, and ignore the master and filesystem servers in this section.

2.4.1 A strawman search index

In our initial version, clients have access to a single server. For every document, the server stores an encrypted Bloom filter corresponding to the set of keywords in the document. To update the search index for a particular document, the client computes the Bloom filter for the contents of the document and encrypts it using a one time pad unique to that update. We generate the mask for a document using a pseudorandom function (PRF) keyed with a per-folder key and the current document version number as input. The key management functionality built into the underlying filesystem ensures that every client has a copy of this PRF key.

If there are n documents in the search index and Bloom filters are m bits, then we can think of the server as storing an $n \times m$ table where each element is a single bit (Figure 5). Each row in the table is a Bloom filter for a document, and the ith row corresponds to the document with ID i. For an update, the client sends a new *row* that the server inserts into its table. This allows the client to easily modify existing documents and add new ones: the server either replaces an existing row with the new row or appends the new row to the table.

To search for a keyword, the client must find all the documents where the Bloom filter indexes corresponding to that keyword are set to "1". The client can check this by retrieving from the server the *columns* corresponding to the Bloom filter indexes for that keyword. The client can decrypt bit b_i in a column by computing the mask for row i, extracting the mask bit corresponding to that

column r_i, and then evaluating $b_i \oplus r_i$. If the ith entry in each of the decrypted columns is set to "1", then the client marks document i as containing the keyword. In order to prevent the attacker from learning the queried keyword from the requested indexes, we compute the Bloom filter indexes using a PRF keyed with a per-folder key and the keyword as input. This key is managed by the underlying filesystem in the same way that the other PRF key is.

We note that in order for the contents of the client's update to remain hidden from the server, the client must be able to retrieve the correct version number from the underlying filesystem. Without this guarantee, the client could use the same mask twice, leaking information about the update contents. For this reason, we only provide security against a malicious attacker if the underlying filesystem also provides the correct version numbers (discussed in Section 2.3.4). This strawman proposal is similar to the one described in prior work [294].

2.4.2 Eliminating search access patterns

To eliminate search access patterns, we need to hide from the server which columns the client is retrieving during a search. To do this, we use a private information retrieval (PIR) protocol [112, 113], which allows a client to retrieve an entry in a database from a server (1) without the server learning which entry is being retrieved, and (2) using total communication sublinear in the database size.

Tool: Distributed Point Functions (DPFs). One concretely efficient way to implement PIR is using a distributed point function (DPF) [201] (later generalized as function secret sharing [77, 78]), which we identify as particularly well-suited for our setting. DPFs allow a client to split a point function f into *function shares* such that any strict subset of the shares reveal nothing about f, but when the evaluations at a given point x are combined, the result is $f(x)$.

A DPF is defined by the following algorithms, which implicitly take the security parameter as input:

- $\mathsf{DPF.Gen}(a,b) \rightarrow (K_1,\ldots,K_\ell)$: Generates keys $K_1,\ldots,K_\ell$ that allow the ℓ servers to jointly evaluate the point function that evaluates to b at input a.
- $\mathsf{DPF.Eval}(K_i,x) \rightarrow y$: Evaluates the function share corresponding to key K_i at server i on input x to produce output y.

To evaluate the point function f where $f(a) = b$ on some input x, the client generates keys for all ℓ servers by running $\mathsf{DPF.Gen}(a,b)$ and sending K_i and x to server i for all ℓ servers. Server i then runs $\mathsf{DPF.Eval}(K_i,x)$ and returns the result y_i to the client. The client can then compute $y_1 \oplus y_2 \cdots \oplus y_\ell$ to reconstruct $f(x) = y$. We make black-box use of the construction from Boyle et al. where $\ell = 2$ [78].

Leveraging DPFs to search. To hide search access patterns, we switch from having the client interact with a single server to having the client interact with ℓ servers in different trust domains that hold identical copies of the search index. To retrieve column j, the client generates shares of the point function that evaluate to all 1's at column j and all 0's for all other columns. The client then sends a function share to each server. Each server evaluates its function share for each column, ANDing the DPF evaluation with the contents of the column, and sends the XOR of the results back to the client. The client then assembles the responses to recover column j.

Using DPFs to retrieve columns requires a linear scan over the search index for a folder. While this is expensive asymptotically, we only aim to show efficiency for realistic workloads, motivating our decision to compress the search index using Bloom filters. Note that depending on the Bloom filter size m, it may be more concretely efficient to send a secret-shared "one-hot" vector that is zero everywhere and 1 at location j in order to retrieve column j.

2.4.3 Protecting against malicious attackers

So far, we have assumed that all servers are honest-but-curious. We now show how to defend against a malicious attacker (namely, an attacker that can deviate from the protocol) that can compromise up to $\ell - 1$ of the ℓ servers. To achieve this, we need to ensure that for a search, the server evaluates the DPF on columns corresponding to the most recent updates sent by the client (not corrupted or old updates).

Strawman: MAC for every bit. We start by showing a strawman that employs MACs, but increases the bandwidth and search latency by roughly a factor of the MAC tag size. For each update, the client additionally sends a MAC tag for every bit in the encrypted Bloom filter. The client cannot send a single tag for the row because to search, the client must retrieve individual columns rather than entire rows. We can think of the server as now storing a second table of MAC tags where each entry of this table is the tag for the corresponding entry in the original table (as in Figure 5).

We need to ensure that (1) a tag is only valid for a particular document update (to prevent replay attacks) and that (2) it cannot correspond to a different Bloom filter index. To do this, we compute the MAC over not only the single Bloom filter bit, but also the document ID, Bloom filter index, and document version number. As with the PRF key, we use the key management functionality in the underlying filesystem to ensure that every client has a copy of the MAC key.

The client now runs the DPF over the columns in both the original table and the MAC tag table. After assembling the responses from all ℓ servers, the client can check that the tag for every bit is correct. However, this increases both the bandwidth and the time to perform the linear scan over the index (i.e., the search latency) by a factor of the tag size. We identify aggregate MACs as a tool to transform this factor from a multiplicative to an additive one.

Tool: Aggregate MACs. We leverage aggregate MACs [281] to allow the servers to combine individual MAC tags into a single aggregate MAC tag. Aggregate MACs, analogous to aggregate signatures [69], allow multiple MAC tags computed with possibly different keys on multiple, possibly different messages to be aggregated into a shorter tag that can still be verified using all the keys. Notably, aggregating MAC tags does not require access to the keys.

The Katz-Lindell aggregate MAC construction [281] works as follows. To generate a MAC tag for some message m using a key k, we simply use a pseudorandom function MAC and compute $t \leftarrow \mathsf{MAC}(k, m)$. To aggregate MAC tags $t_1, \ldots, t_n$, the aggregator computes $T \leftarrow \oplus_{i=1}^{n} t_i$. To verify an aggregate MAC tag T using messages $m_1, \ldots, m_n$ and keys $k_1, \ldots, k_n$, the verifier checks $T \stackrel{?}{=} \oplus_{i=1}^{n} \mathsf{MAC}(k_i, m_i)$.

Aggregating MAC tags to improve performance. To improve performance by a factor of the tag size, we allow the servers to combine individual tags into a single aggregate tag. To search, the

server evaluates the DPF on the contents of the column and a single aggregate tag for the entire column.

Aggregating MAC tags also allows us to reduces storage space at the servers. Rather than storing an entire separate MAC table, the servers instead keep an array of aggregate tags, one for each column. On each update, the client XORs the old tag with the new tag (which is why Update takes both prevWords and currWords). By then XORing this value with the aggregate tag, the server can remove the old tag and add the new tag. To ensure that this aggregate MAC tag is maintained correctly, the server must check that the client has the latest version of the document; otherwise it rejects the update.

2.4.4 Supporting dynamic membership

Users might be added to or removed from a folder, requiring the new group to generate a new key. This new key might be in use at the same time that some parts of the search index were generated using an old key in order to support lazy revocation. We let the underlying filesystem handle key management, but we need to ensure that our search protocol supports multiple keys that may be active at the same time.

Decrypting search results is straightforward; to decrypt the results for an individual document, the client uses the same key from the last update to that document. Aggregating MAC tags is also simple because we can aggregate tags computed with different keys. We can remove old tags and add new tags with different keys using XOR in the same way as before.

2.4.5 Final DORY protocol

We show the pseudocode for the search protocol in Figures 6, 7, and 8.

2.5 Replication across trust domains

DORY requires that the servers processing search requests operate on the same version of the index in order for the client to receive a valid response; otherwise, the cryptographic shares from the DPF cannot be combined correctly. Because our system processes a mix of update and search requests, the servers need to agree on the index state. The client also needs to know the document version numbers corresponding to the index that the servers used to execute the search; otherwise, the client will be unable to decrypt and verify the result.

Because we are in an adversarial environment, a natural solution is to use a Byzantine fault-tolerant (BFT) consensus algorithm [15, 60, 96, 125, 295, 307] to agree on the ordering of update and search requests. Standard BFT provides the properties we need, but requires $3f+1$ servers, each in its own trust domain, to handle f failures. A large number of trust domains is expensive to maintain and difficult to deploy, increasing the overall system cost. We make several observations about our setting that allow us to use only $f+1$ trust domains.

```
def update (folder_id, doc_id,
            old_keywords, new_keywords):
    k1_bf, k1_prf, k1_mac = fsclient.get_curr_key(folder_id)
    k2_bf, k2_prf, k2_mac = fsclient.get_key_for_doc(doc_id)

    ## Build encrypted Bloom filters.
    a = bf.build(k1_bf, new_keywords)
    b = bf.build(k2_bf, old_keywords)
    v = fsclient.get_version(doc_id)
    a = a ^ PRF(k1_prf, (doc_id, version + 1))
    b = b ^ PRF(k2_prf, (doc_id, version))

    ## Compute tag for each Bloom filter entry.
    for i = 1 to m:
        x[i] = MAC(k_mac, (a[i], doc_id, i, version + 1))
        y[i] = MAC(k_mac, (b[i], doc_id, i, version))
        z[i] = x[i] ^ y[i]

    ## Send update to servers.
    for i = 1 to num_servers:
        dory_server[i].apply(folder_id, doc_id, a, z)
```

Figure 6: Pseudocode for client update protocol.

Observations we leverage. We make three observations that allow us to tailor the problem of consensus to DORY:

DORY deterministically detects server misbehavior. Our cryptographic protocol already defends against malicious servers; if a server executes the client's query incorrectly or over an incorrect version of the index, the client will detect this (triggering a manual investigation). This is a significant departure from the Byzantine fault model where failure information is imperfect. By handling server misbehavior at the cryptographic protocol layer, we can use a fail-stop rather than Byzantine failure model at the consensus layer. This and the next observations allow us to use just $f+1$ trust domains to tolerate f failures.

Trust domains provide availability. To support search, DORY needs all $f+1$ replicas to be available. We need to ensure that servers across multiple trust domains remain online to allow clients to search. Here we leverage the observation that for trust domains deployed in the cloud, the cloud provider is monetarily incentivized to provide availability (Section 2.2.4). This means that if a server in a trust domain fails, either it will eventually come back online or another server will take its place; even if failures occur, $f+1$ servers will be available again at some point in the future.

DPFs give us replication for free. The challenge now is to reinitialize the state of these failed servers. The use of DPFs in our cryptographic protocol requires all replicas to have identical copies of the

```
def search (folder_id, keyword):
    doc_ids = fsclient.get_doc_ids(folder_id)
    k_bf, _, _ = fsclient.get_curr_key(folder_id)
    I = bf.get_indexes(k_bf, keyword)
    for i in I:

        ## Evaluate DPF for index i at servers.
        K = dpf.gen(i, 1)
        for j = 1 to num_servers:
            v[j], t[j] = dory_server[j].eval(K[j])
        y = 0

        ## Decrypt and verify result.
        for doc_id in fsclient.get_doc_ids(folder_id):
            w = xor_all(v[:][doc_id])
            k_prf, k_mac = fsclient.get_doc_key(doc_id)
            version = fsclient.get_version(doc_id)
            y = y ^ MAC(k_mac, (w, doc_id, i, version))
            x = PRF(k_prf, (doc_id, version))
            w = w ^ x[i]

            ## Remove doc_id if no match.
            if w = 0:
                doc_ids.remove(doc_id)

        ## Abort if verification fails.
        if y != xor_all(t):
            return Error("Verification failed.")

    return doc_ids
```

Figure 7: Pseudocode for client search protocol.

```
def eval(folder_id, K):
    r = [0 for i in range(sizeof(agg_mac[0])]
    s = [0 for i in range(m)]
    ## Evaluate the DPF for each Bloom filter index.
    for i = 0 to m:
        b = dpf.eval(K,i)
        for j = 0 to sizeof(agg_mac[0]):
            r[j] = r[j] ^ (b & (agg_mac[i] & (1 << j)))
        for doc_id in fsserver.get_doc_ids(folder_id):
            s[doc_id] = s[doc_id] ^ (b & enc_contents[doc_id][i])
    return (r,s)

def apply(folder_id, doc_id, contents_update, mac_update):
    enc_contents[doc_id] = contents_update
    agg_mac = agg_mac ^ mac_update
```

Figure 8: Pseudocode for server protocols.

search index. Normally it is unsafe to transfer state between trust domains, as the recipient has no way to verify correctness. However, because the client can check the integrity of the state used to execute a search query, we can safely copy state across trust domains. Because we have $f+1$ servers, at least one server will always remain online to preserve the state of the index.

2.5.1 Algorithm

A DORY cluster contains the following entities (Figure 9):

Master: The master receives updates and manages replica state. The master stores the most recent updates and version numbers (both the overall system version number and individual document version numbers), but not the entire search index. The master can be deployed in any trust domain, as clients can detect misbehavior when verifying search results.

Replicas: The replicas receive updates from the master and perform searches from the user. The replicas store the most recent versions of the index as well as the version numbers (both the overall system version number and individual document version numbers). We must deploy ℓ replicas in ℓ different trust domains to ensure that the client can split its search request across different trust domains. However, the total number of replicas n may be greater than ℓ in order to improve fault-tolerance.

We additionally use a watchdog service (commonly available in the cloud) that periodically checks that all servers are still online and triggers recovery when it detects a crash.

Properties. Our replication algorithm should provide the following properties:

- **Correctness:** If *all* of the replicas and the master fail, a client with the correct set of document version numbers can detect this.

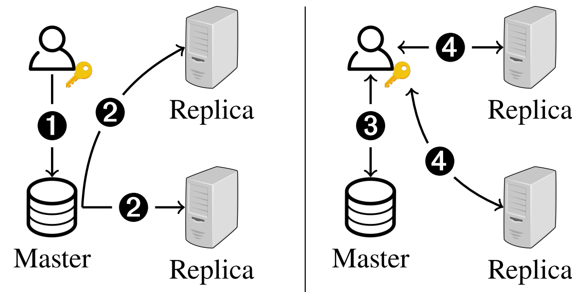


Figure 9: System architecture and protocol flow for updates (left) and searches (right). ❶ Client sends update to master. ❷ Master propagates updates to replicas. ❸ Client requests version number(s) from master. ❹ Client splits search request across replicas.

- **Fault-tolerance:** If at most $n-1$ of the n replicas fail, then the search index is preserved. If the master fails, then the most recent set of updates can be recovered with help from the client.

We do not guarantee availability if individual trust domains do not provide availability.

Algorithm. We now explain how we handle updates and searches and recover from failure (see Figure 9).

Updating a document. To update a document, the client sends the update along with the new document version number to the master. The master needs to send the update to the replicas and increment the version number. Because the master might fail while sending the update to the replicas, the master runs two-phase commit [308] with the replicas to ensure that all the replicas receive the update and associated version number. We do not need to worry about replica failures during two-phase commit (and so do not need multiple replicas in each trust domain); if a replica fails, the watchdog service will detect this and coordinate recovery as described below.

Searching for a keyword. To search for a keyword, the client first needs to learn the current version numbers from the master (both the overall system version number and the corresponding individual document version numbers). If the client has a relatively recent set of document version numbers, the master can simply send updates for a few of the document version numbers, making the overall bandwidth much smaller than the number of documents. The client then generates a search query for ℓ of the replicas. The replicas execute the search on the version of the index corresponding to the system version number sent by the client.

Coordinating recovery. We rely on the watchdog service to detect failures. If at least ℓ of the replicas across ℓ different trust domains remain online, clients can continue searching. Otherwise, we can start new replicas and transfer the state from a remaining replica to the new replica, even if the replicas are in different trust domains. This will cause a slight delay for clients waiting to search, but is safe due to the underlying cryptographic protocol (as discussed above). We do not need to worry if the master fails, because the master does not respond to the client until it has propagated the update to the replicas. If a replica fails during two-phase commit, the master can roll back the two-phase commit and then start another replica in the same trust domain and copy the state across trust domains.

2.5.2 Batching

Rather than running two-phase commit between the master and replicas for every update, we can apply batching to amortize the cost. Instead of immediately sending an update to the replicas, the master aggregates a batch of updates, and, when this batch reaches a certain size or a certain amount of time has elapsed, it runs two-phase commit with the replicas to transfer the current batch of data.

However, now that the master is responding to clients before sending the updates to the replicas, we need to ensure that the master does not lose state when it fails. In particular, the master needs to be able to recover the updates that were waiting to be committed to the replicas. The master does this by comparing the individual document version numbers at the replicas with those at the filesystem server. For each document where the version numbers differ, the master can request an update from the next client to come online with access to that document.

2.6 Implementation

We implemented DORY in ~5,000 lines of C (for the distributed point function and other low-level cryptographic operations) and Go (for the networking and consensus). We used the OpenSSL library, and our DPF implementation closely follows the one in Express [167]. We instantiate the PRF using AES. We also implemented the DORY client on an Android Google Pixel 4. In addition to the C code, which we ported to the mobile platform, we wrote ~1,200 lines of Java. We used the tiny AES library [488] to minimize memory usage in our mobile implementation. Our implementation supports a single folder and does not include the watchdog service and coordinated recovery described as part of Section 2.5. We always generate DPF keys, regardless of the Bloom filter size. The source code is available at `https://github.com/ucbrise/dory`.

2.6.1 Parallelism

The linear scan over the search index can be easily parallelized across both cores and servers because it carries no state from document to document.

Thread-level parallelism. Since we evaluate the DPF on each column of the search index, we parallelize the scan operation by simply assigning each thread a number of columns and then combining the results computed by each thread.

Server-level parallelism. We can partition the search index by having different pairs of replicas maintain different parts of the search index. The client then sends a search query to all pairs of replicas and simply computes the union of the results. Replica partitioning improves latency since each replica now only needs to search over a part of the index instead of the full index. Each pair of replicas can store part of the search index for many folders, making it possible to keep search latency low, but the overall throughput high.

Docs	BF size
$\leq 2^{10}$	140 B
$\leq 2^{11}$	160 B
$\leq 2^{12}$	180 B
$\leq 2^{13}$	200 B
$\leq 2^{14}$	225 B
$\leq 2^{15}$	250 B
$\leq 2^{16}$	280 B
$\leq 2^{17}$	315 B
$\leq 2^{18}$	350 B
$\leq 2^{19}$	390 B
$\leq 2^{20}$	435 B

(a)

Docs	Time breakdown, p=1 (ms)				End-to-end latency (ms)		
	Consensus	*Client*	*Network*	*Server*	*p=1*	*p=2*	*p=4*
2^{10}	0.73	0.54	58.67	2.68	62.62	61.81	61.51
2^{11}	0.73	0.87	58.41	4.11	64.12	62.39	61.89
2^{12}	0.73	1.52	57.99	7.09	67.33	64.46	62.92
2^{13}	0.73	2.80	58.74	12.03	74.30	68.08	64.78
2^{14}	0.75	5.30	77.88	26.24	110.17	75.76	68.59
2^{15}	0.76	10.18	80.59	50.97	142.50	112.71	76.76
2^{16}	0.81	19.83	100.67	108.78	230.09	147.39	115.50
2^{17}	0.86	38.99	119.38	240.45	399.48	243.43	153.56
2^{18}	1.19	76.92	142.28	527.67	748.06	428.40	256.15
2^{19}	1.78	154.37	151.98	1172.46	1480.59	800.98	454.52
2^{20}	2.81	306.34	148.96	2602.83	3060.94	1636.80	862.42

(b)

Table 10: On the left, Bloom filter sizes (in bytes) necessary for < 1 expected false positive assuming an average of 73.18 keywords per document where each keyword hashes to 7 Bloom filter indexes (Table 10a). On the right, breakdown of search latency without parallelism and end-to-end search latency with parallelism where p is the degree of server parallelism (Table 10b).

2.6.2 Fast PRF evaluation

In order to decrypt the search result received from the server, the client must compute a mask for each individual document. To reduce the number of PRF evaluations to decrypt, we group Bloom filter indexes for the same keyword in the same 128-bit block. This grouping allows the client to decrypt the search results for one document using a single PRF evaluation. This does not significantly impact the false positive rate of the Bloom filter because we can now model a m-bit Bloom filter storing w words as $m/128$ independent Bloom filters each storing $128w/m$ words.

2.7 Evaluation

We evaluated DORY to determine (1) how it performs in comparison to existing techniques and (2) whether it meets the requirements outlined by the companies we surveyed. We consider the following metrics: latency (Section 2.7.2), throughput (Section 2.7.3), storage (Section 2.7.4), bandwidth (Section 2.7.5), and cost (Section 2.7.6). We compare DORY's performance to two different variations of DORY as well as plaintext search and a baseline built on ORAM (Section 2.7.1) that provides similar guarantees to those of DORY. We show that DORY meets the requirements outlined by the companies we surveyed and outperforms (in some cases, by orders of magnitude) our ORAM baseline (Section 2.7.1).

Experimental setup. We evaluate DORY on AWS using r5n.4xlarge instances with 128GB of memory and 16 vCPUs for the replicas and the master. We use a c5.large client with 4GB of memory and 2 vCPUs to model a user's desktop machine. We use an Android Pixel 4 to measure the time

to search on a mobile client. We place the two trust domains in different regions to ensure that machines are in different clusters to model different organizations, although in practice these clusters would likely be geographically close to maximize performance. We use `us-east-1` and `us-east-2` for our DORY experiments and `us-west-1` and `us-west-2` for our ORAM baseline experiments. All DORY communication occurs over TLS. We run experiments for a single folder; a real system would maintain many such folders in parallel. Our experiments assume queries with just one keyword. To support queries with up to k keywords, the client could make k requests in parallel.

System parameters from Enron email dataset. We use the Enron email dataset, which is commonly used to evaluate encrypted search schemes [94, 269, 329, 361, 375, 376, 534] to set Bloom filter sizes for DORY. We leverage the same standard keyword extraction techniques used in Oblix [361]: we stemmed the words and removed stopwords and words that were > 20 or < 4 characters long or contained non-alphabetic characters. In the over 500K emails, each email has an average of 73.18 keywords with a standard deviation of 114.89.

Regarding the configuration of the Bloom filters, each keyword hashes to 7 locations in the Bloom filter, as we found that it provided a reasonable tradeoff between the time to perform the linear scan at the server and bandwidth. We choose the Bloom filter size based on the number of documents in a folder so that, for every search in that folder, the search results have less than one false positive document in expectation. The sizes of the Bloom filters are specified in Table 10a.

2.7.1 Baselines

We evaluate DORY in comparison to four baselines:

- **ORAM baseline**: Eliminates search access patterns using ORAM (expected to incur a significant overhead). With this baseline, we show how DORY compares to a solution that provides comparable security guarantees.
- **Plaintext search**: Searches over a plaintext inverted index and does not provide any security guarantees (expected to have much lower overhead than DORY, even though not optimized).
- **Semihonest DORY**: Modifies the DORY protocol to only provide security against semihonest adversaries (expected to have lower overhead than DORY).
- **Leaky DORY**: Modifies the DORY protocol to allow search access pattern leakage by using only one trust domain and querying the replica directly for the indexes corresponding to a keyword rather than using a DPF (expected to have lower overhead than DORY).

Semihonest DORY illustrates the overhead of the MAC checks necessary to defend against malicious adversaries, and leaky DORY illustrates the overhead of the DPF queries. In all of the baselines except the ORAM baseline, we use the same consensus system as in DORY, although for the baselines where there is only one trust domain (leaky DORY and plaintext search), the master only needs to send update batches to a single trust domain (we model this by placing all servers in the same AWS region). Only the ORAM baseline has security guarantees comparable to those of DORY.

ORAM baseline. Many academic works [254, 269, 375, 469] point to an inverted index in ORAM [206, 392] as a way to achieve searchable encryption without search access pattern leakage, making it a natural baseline for searching within a folder. Traditional ORAM is designed for a

single client and requires the client to maintain ORAM client state hidden from the server [473]. A separate line of work explores extending single-user constructions to multi-user settings [40, 110, 241, 341, 342, 352]. Mayberry et al.'s system [352] is particularly fit for our setting as it protects mutually trusting clients (clients with access to a given folder) from a malicious server. For a semi-honest server or for a malicious server for which we have a mechanism to verify the data returned (discussed in Section 2.3.4), their protocol uses a single-user ORAM and requires clients to store the encrypted ORAM client at the server. To perform an operation, the client acquires a lock at the server, downloads and decrypts the ORAM client state, performs the operation, encrypts and sends back the state, releasing the lock.

Client failures. We observed that the above proposal did not consider client failures. If a client fails after issuing operations at the server but before uploading the updated client ORAM state, the next client's access may leak search access patterns (e.g. if it searched for the same word as the previous client). To handle client failures, we require each client to record a client "prepare" operation at the server, and if it fails before completing, the next client can finish the operation.

Eliminating frequency leakage. Popular keywords require multiple ORAM blocks to store all the document identifiers containing that keyword. We need to ensure that the number of blocks accessed doesn't leak the frequency of a keyword due to known attacks [410], as DORY does not leak this frequency. For each search, we fetch the maximum number of blocks a keyword maps to. Similarly for each keyword we update in a document, we fetch the maximum number of blocks a keyword maps to and write back a single block.

Implementation. We implemented our baseline on top of an existing open-source PathORAM implementation in Go [402].

Evaluation on Enron email dataset. While DORY's performance relies only on the system parameters and not the contents of the documents themselves, the performance of both our ORAM and plaintext search baselines depends on document contents. We evaluate these baselines using subsets of the Enron email dataset with the same keyword extraction techniques described above. To evaluate different numbers of documents, we take different-sized subsets of the Enron email dataset. We treat updates as adding an entire email to the index. Because the Enron email dataset only has $\sim$ 528K emails, we do not measure the ORAM and plaintext search baseline beyond that number of documents.

2.7.2 Latency

Update latency. Figure 11 shows that the update latency of DORY is orders of magnitude faster than that of the ORAM baseline. This holds for both the desktop and mobile clients (Figure 12). The baseline requires a number of ORAM accesses (each of which necessitates round trips) linear in the number of document keywords. In contrast, DORY simply uploads a single encrypted Bloom filter. Update latency determines (1) how long it takes for updates to be reflected in search results and (2) how long the client must remain online. Neither is a concern in DORY where updates are processed in less than 1ms, but the ORAM baseline requires clients to remain online for potentially hours.

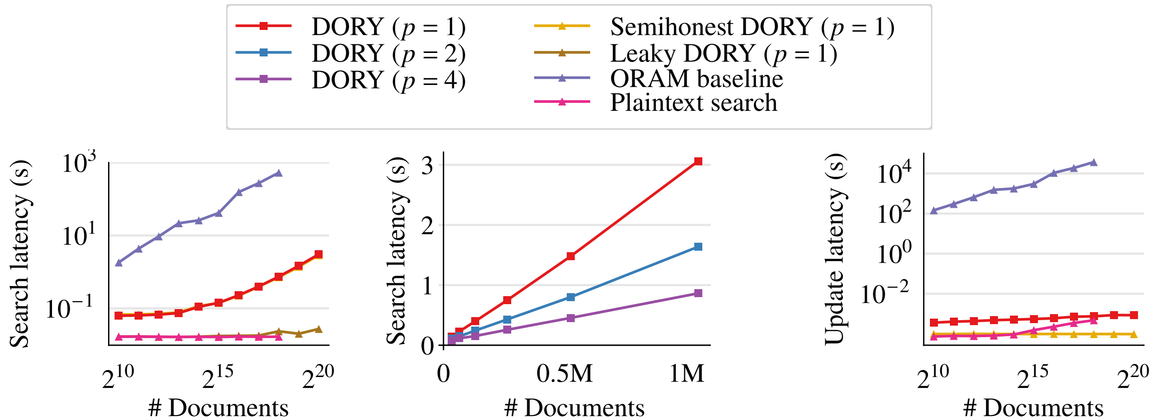


Figure 11: Search latency and update latency. The left and center figures use a logarithmic scale on both axes, and the right figure uses a linear scale on both axes (p denotes server parallelism). The update latency of leaky DORY exactly matches that of DORY, and the search latency of semihonest DORY is slightly less than that of DORY.

Note that semihonest DORY has a faster update time than DORY because the client does not have to generate a MAC for every bit in the Bloom filter.

Search latency. Table 10b shows the breakdown in search latency. As the number of documents increases, the majority of time is spent performing the linear scan at the server. This is apparent in Figure 11, where leaky DORY's search latency is significantly lower than that of DORY and stays relatively constant as the number of documents increases due to the fact that leaky DORY does not need to perform a linear scan.

Despite overheads incurred due to the linear scan, DORY is orders of magnitude faster than the ORAM baseline. The MAC overhead to protect against malicious adversaries is barely noticeable, as semihonest DORY and DORY have almost identical search latencies. Mobile clients incur additional overhead in comparison to desktop clients (the mobile client spends 5 seconds on client-side processing for 1M documents). This overhead is below 1 second for 2^{17} documents (Figure 12).

By increasing the degree of parallelism p and partitioning the search index across replica groups, we can reduce the server time by roughly a factor of p, as this time is linear in the number of documents (Figure 11). Parallelism allows us to reach the target latency set by the companies (Table 2).

2.7.3 Throughput

DORY achieves significantly higher throughput than the ORAM baseline (Figure 13). Parallelism improves DORY's throughput by roughly a factor of p for larger numbers of documents (Figure 14). Relative to other workloads, DORY performs best under update-heavy workloads (updates require an insertion while searches require a linear scan), and the ORAM baseline performs best under search-heavy workloads (searches require fewer ORAM accesses than updates).

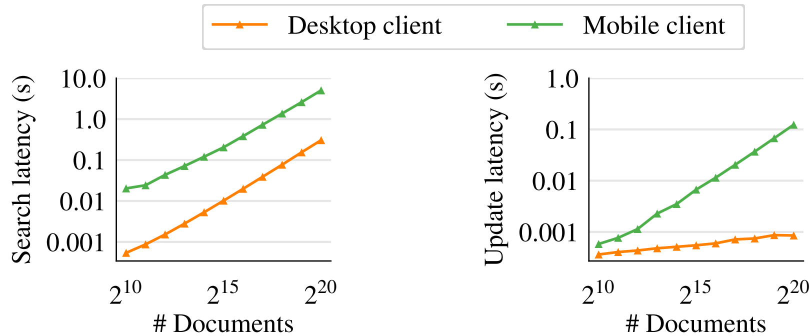


Figure 12: Latency for mobile client and desktop client. Both plots use a logarithmic scale on both axes.

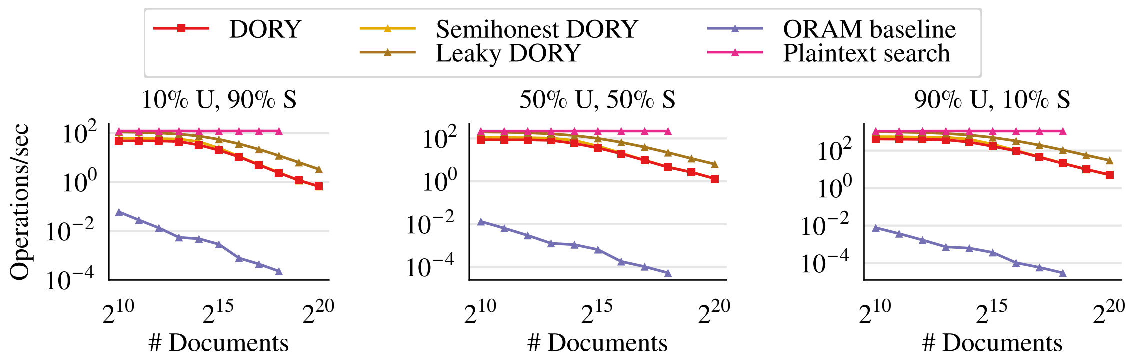


Figure 13: Throughput under a variety of workloads (U indicates updates, S indicates searches). The performance of semihonest DORY closely matches that of DORY. All plots use a logarithmic scale on both axes.

2.7.4 Storage

Server state. Figure 15 shows how DORY uses substantially less storage space at the server than the ORAM baseline and storage space comparable to that of a plaintext inverted index. The plaintext search index grows more slowly than DORY's search index, making the plaintext search index smaller than the DORY search index for larger numbers of documents.

Client state. DORY only requires that the client store three 128-bit keys. To generate an update or decrypt a search result, the client also needs to know the version number for each document. To minimize bandwidth, the client can optionally cache the latest version numbers so that it only needs to retrieve the version numbers that changed. For 45K documents (the highest average number of documents per user among the companies we surveyed), storing these version numbers would require 175.8KB. For 1M documents, storing these would require 3.84MB. Our ORAM baseline only requires the client to store a single 128-bit AES key to encrypt and decrypt the ORAM client, and plaintext search requires no client storage.

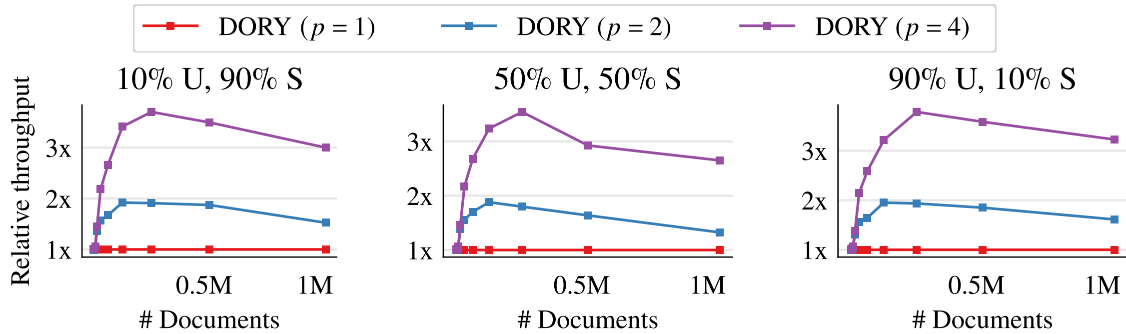


Figure 14: Effect of parallelism (p denotes the degree of parallelism) on throughput for different workloads (U indicates updates, S indicates searches).

2.7.5 Bandwidth

Search and update bandwidth is also much smaller in DORY than in the ORAM baseline (Figure 15). The ORAM baseline incurs a significant overhead by sending the encrypted client state, but ORAM accesses are responsible for the majority of the communication. In contrast, the search bandwidth in DORY is linear in the number of documents, and the update bandwidth depends on the size of the Bloom filter. MACs are responsible for a significant part of the update bandwidth in DORY, which is why semihonest DORY has much lower update bandwidth. The difference in search bandwidth between leaky DORY and DORY is due to the size of the DPF keys; however, unlike plaintext search, the search bandwidth for both is still linear in the number of documents. We do not include the bandwidth to retrieve version numbers for individual document numbers in DORY, as these version numbers can for the most part be cached at the client as described above.

Comparison to client index. To evaluate the practicality of a client-side index instead of DORY, we built an inverted index over the Enron email dataset using a B+ tree. We found that the index is 159.9MB, and while it is feasible to store this amount of data, even on a mobile device, synchronization requires significant bandwidth. One way to keep this data structure updated would be to require each client to download the contents of every update. However, this solution requires roughly the same amount of bandwidth as syncing all the files locally, which we were trying to avoid in the first place. Instead, we could keep an encrypted copy of the client index at the server. Which part of the index is updated leaks information about the document contents, and so whenever a client performs an update, it must encrypt the entire index and send it to the server. Before a client updates or searches, it must download the most recent copy of the search index. This results in roughly a $365\times$ increase in search bandwidth and a $3,334\times$ increase in update bandwidth in comparison to DORY.

2.7.6 Cost

The companies we met with estimated a workload with 50% updates and 50% searches, and the highest average number of documents per user reported was 45K. The throughput of two replicas and a master operating on a folder of 45K documents under this workload is 19.5 operations/second.

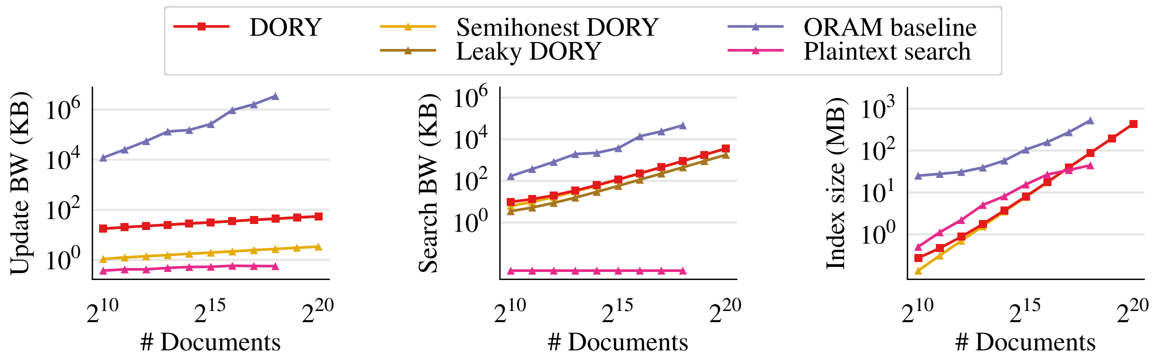


Figure 15: Storage space and bandwidth for DORY in comparison to other baselines. The update bandwidth of leaky DORY exactly matches that of DORY, and the search bandwidth of semihonest DORY is slightly less than that of DORY.

One of the companies reported that active users make roughly 50 updates per day, and so based on 100 operations per day and the cost to run a single r5n.4xlarge instance ($1.192/hour), each user costs roughly $0.0509 per month, well under the maximum permissible cost per user per month of $0.70-$5.54 reported by the companies. Depending on the way in which trust is distributed (see Section 2.2.4), trust domains may incur additional setup and maintenance costs not captured by our calculation.

2.8 Related Work

We first describe related work before the time of publication of the original paper [140] (Section 2.8.1), and then we describe related work after publication (Section 2.8.2).

2.8.1 Related work before publication

Symmetric Searchable Encryption (SSE). A long line of work has examined the problem of Symmetric Searchable Encryption (SSE) [95,104,129,148–151,192,202,274,275,376,436,464,469], summarized in the following surveys [72,242,411]. Many of these schemes assume a single user and do not support efficient revocation, but more importantly, many permit some search access pattern leakage, opening the door to attacks [94,269,284,329,410,415,534]. SEAL [150] explicitly allows developers to tradeoff between leakage and performance.

Multi-server SSE and ORAM. Some SSE schemes use multiple servers to improve efficiency but still permit leakage, with some providing richer functionality than simple keyword search [54, 174, 266, 303, 399, 427]. Bösch et al. [73] and Hoang et al. [255] use multiple servers to hide search access patterns and improve efficiency. Hoang et al. [255] use a similar table layout where updates and searches correspond to different dimensions in the table. However, both schemes do not support multiple users, assume honest-but-curious servers, and require expensive updates to hide

the document being updated. Our scheme also has similarities to distributed ORAM schemes that leverage multiple servers to hide access patterns with improved efficiency [17, 161, 212, 336, 470]. Implementing search with one of these schemes would still require clients to perform an ORAM access for every document keyword during an update.

Multi-user SSE and ORAM. Many existing multi-user searchable encryption schemes that support fast revocation use a different key for each user and leverage proxy encryption [34, 45] or pairings [45, 288, 412, 481]. This class of schemes use deterministic query encryption algorithms that leak search access patterns. The most efficient ORAM constructions assume a single user, with multi-user ORAMs incurring a much larger overhead by leveraging expensive tools such as multi-party computation [40, 110, 241, 341, 342].

SSE and ORAM with trusted hardware. One way to improve performance and, in the case of search, potentially reduce leakage is by leveraging trusted hardware. ZeroTrace [445], Obliviate [21], ObliDB [168], GhostRider [328], Tiny ORAM [181], and Shroud [334] combine oblivious techniques with trusted hardware. HardIDX [187], Oblix [361], POSUP [253], and Amjad et al. [25] use trusted hardware specifically for the problem of searching on encrypted data. Unlike DORY, such solutions only require a single server, but they necessitate both additional trust assumptions (due to known side-channel attacks) and additional deployment costs.

Prior use of DPFs in systems. Splinter [508] uses function secret sharing (both DPFs and range queries) to allow users to efficiently make private queries on a public database. DURASIFT [174] uses DPFs with multi-party computation across multiple servers to support boolean expressions of keyword searches for multiple users without leaking search access patterns. However, its techniques incur significant overhead in comparison to ours, and the authors consider thousands rather than millions of documents. Floram [161] uses DPFs to implement a distributed-trust ORAM that has linear costs but fast concrete performance. Metadata-hiding communication also benefits from DPFs (e.g. Riposte [122] and Express [167]).

BFT consensus and fault-tolerance. BFT consensus [15,60,96,125,295] is a classical problem. Prior work has explored reducing the number of participants in BFT consensus by separating agreement from execution [528], only activating some nodes when failures are detected [159, 276, 518], relaxing synchrony assumptions [16, 332, 414], adopting a hybrid fault model [414], and using an attested, append-only log [114]. A separate line of theoretical work considers Byzantine fault-tolerance specifically for the case of private information retrieval [44, 50, 185, 475] using information-theoretic tools.

Oblivious systems. ObliviStore [471], Obladi [128], Opaque [536], Cipherbase [32], and Taostore [442] are practical systems for obliviously storing and querying data (not necessarily for the problem of searchable encryption).

2.8.2 Subsequent related work

We now include some related work from after the time of the original paper publication [140].

Searchable encryption and leakage-abuse attacks. Researchers have proposed a number of new searchable encryption schemes, including schemes that take into account corrupted participants [100],

reduce costs when deletions are permitted [99], optimize for I/O efficiency [366], and resist volume leakage [330] (see this 2023 survey [319]). Researchers have also developed new leakage-abuse attacks [134, 381, 393, 394, 519, 533] taking advantage of leakage from memory accesses, search patterns, and the number of results. Another line of work has pointed out the importance of considering leakage from the larger system that a searchable encryption scheme is a part of [231]—in particular, the leakage introduced by retrieving documents. DORY is a search engine that integrates with an end-to-end encrypted filesystem that can leak information, and so this context is important to consider for a DORY deployment. The system SWiSSSE aims to address this problem [232]. Gui, Paterson, and Tang describe attacks that arise from integrating MongoDB's queryable encryption with a larger database system [233].

Private search systems. Another line of work builds search systems with strong privacy guarantees via access-pattern-hiding primitives. Coeus allows a user to privately retrieve a list of ranked query results for Wikipedia documents and then privately fetch the document [22]. Tiptoe uses linearly homomorphic encryption with techniques from machine learning in order to privately search over roughly 360M webpages [249]. DeSearch uses trusted hardware in order to build a decentralized search engine for decentralized services (e.g., blockchain applications) [321]. Like DORY, these systems provide strong privacy for user queries, but unlike DORY, they search over public data.

Servan-Schreiber et al. show how to support private approximate nearest-neighbor search using two non-colluding servers and distributed point functions [449]. Their scheme aims to protect client queries from the servers and server data (other than the query response) from the clients. We also built Waldo, which provides guarantees similar in spirit to those of DORY, but supports more complex queries (Chapter 3).

Setting up distributed-trust systems. We explored how challenging it is to bootstrap distributed-trust systems and showed how to take advantage of existing cloud infrastructure and secure hardware to make this process easier [138]. Flock uses serverless functions to make it possible to deploy distributed-trust applications from the client [283]. These techniques could make it easier to set up a DORY deployment.

Private information retrieval. DORY is based on private information retrieval (PIR). Researchers have recently proposed a number of PIR schemes with good concrete efficiency, some of which rely on preprocessing and some of which batch queries together [250, 317, 331, 358, 359, 370, 371, 538]. A line of work around authenticated PIR provides integrity guarantees, much as DORY provides integrity guarantees for queries [117, 146, 158, 513].

2.9 Security analysis

We use the simulation paradigm of multi-party computation. to define DORY's security guarantees against an adversary who can compromise any number of servers. In particular, we allow our adversary to be *malicious* and so deviate arbitrarily from the protocol. We define security using an ideal world where rather than running the DORY protocol, the clients interact with an ideal functionality $\mathcal{F}$. We compare the ideal world to the real world, where the clients, honest server(s),

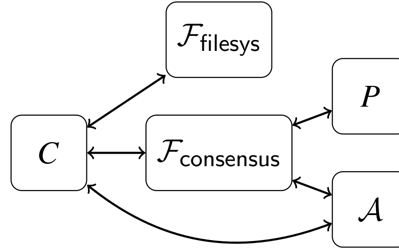


Figure 16: Real world, including the clients C with access to a given folder (all other clients are controlled by $\mathcal{A}$), the honest servers P, and the adversary $\mathcal{A}$, which includes the compromised servers.

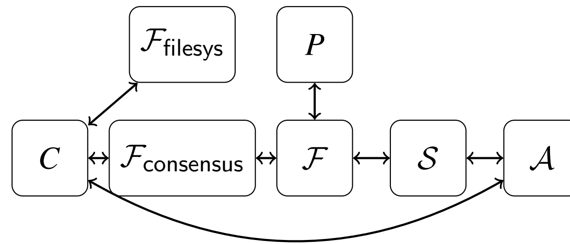


Figure 17: Ideal world, including the client C with access to a given folder (all other clients are controlled by $\mathcal{A}$), the filesystem ideal functionality $\mathcal{F}_{\text{filesys}}$, the consensus ideal functionality $\mathcal{F}_{\text{consensus}}$, the DORY ideal functionality $\mathcal{F}$, the honest servers P, the simulator $\mathcal{S}$, and the adversary $\mathcal{A}$, which includes the compromised servers.

and adversary interact directly using the DORY protocol. For simplicity, we frame our analysis in terms of servers rather than trust domains and assume that each server is deployed in a different trust domain. Our modeling of a system using the simulation paradigm draws inspiration from Ghostor [258].

In the ideal world, the ideal functionality $\mathcal{F}$ allows us to exactly define the leakage for DORY using a simulator $\mathcal{S}$. The clients interact with $\mathcal{F}$ and $\mathcal{F}$ gives to $\mathcal{S}$ exactly what DORY leaks. $\mathcal{S}$ then executes the operation via interaction with the adversary $\mathcal{A}$. The challenge is for the adversary $\mathcal{A}$ to tell whether it is being invoked in the real world (Figure 16) or the ideal world (Figure 17).

We allow $\mathcal{A}$ to adaptively choose the operations issued by clients and see the results of those operations in both the real and ideal worlds. This choice strengthens our security definition, as it allows us to model what happens if an adversary is able to influence the operations performed by the client (e.g. in the real world, an adversary might be able to make the client update the index by sending an email) or observe the outputs of operations. Note that the adversary can only observe the final outputs of the operations; the adversary *cannot* observe the messages sent from the honest server to the client.

To model systems that provide folders for different groups of clients that do not trust each other, we will consider clients with access to a single folder and the adversary $\mathcal{A}$ as having control of every client without access to that folder. The set of clients not controlled by $\mathcal{A}$ is static, and the folder they have access to does not change.

We additionally augment the protocol such that a client verifies that it has access to a folder before making an update or search request to the folder. We assume that the clients under the control of $\mathcal{A}$ only send valid requests.

We will show (informally) that for some ideal functionality $\mathcal{F}$ modeling the functionality that DORY provides, there exists a simulator $\mathcal{S}$ in the ideal world such that for every adversary $\mathcal{A}$, $\mathcal{A}$ cannot tell whether it is in the real world or interacting with the simulator $\mathcal{S}$.

What we don't model. We model our consensus protocol in a blackbox way, treating it simply as the ideal functionality $\mathcal{F}_{\mathsf{consensus}}$. We make this choice to focus on the cryptographic components of the DORY protocol and because two-phase commit is already well-studied. As our protocol interacts with the underlying filesystem in a blackbox way without specifying an implementation, we also model the filesystem as the ideal functionality $\mathcal{F}_{\mathsf{filesys}}$. We also do not model dynamic membership (in particular, user revocation) for simplicity; the set of compromised clients and the set of compromised folders are static.

2.9.1 Definitions

Ideal functionality for filesystem. To simplify our analysis, we model the underlying filesystem as the ideal functionality $\mathcal{F}_{\mathsf{filesys}}$. We now define the ideal functionality for $\mathcal{F}_{\mathsf{filesys}}$ following the API defined in Section 2.3.1.

For each folder, $\mathcal{F}_{\mathsf{filesys}}$ keeps:

- the current key for that folder, and
- a list of the document IDs docIDs in that folder. Each document ID has a corresponding:
  - most recently used key, and
  - version number.

Each time an update for a document is received, the most recently used key for that document is set to the current key for that folder and the version number is incremented. The current key for a folder only changes when a client's access to a folder is revoked (our analysis does not include user revocation).

The ideal functionality $\mathcal{F}_{\mathsf{filesys}}$ uses this data to respond to the following API queries (see Section 2.3.1):

- $\mathsf{getCurrKey}(\mathsf{folderID}) \to k$: Get the current key associated with the group of files in $\mathsf{folderID}$.
- $\mathsf{getDocKey}(\mathsf{docID}) \to k$: Get the key used in the most recent update for docID.
- $\mathsf{getDocIDs}(\mathsf{folderID}) \to \mathsf{docIDs}$: Get all the document IDs used for the documents in $\mathsf{folderID}$.
- $\mathsf{getVersion}(\mathsf{folderID}, \mathsf{docID}) \to \mathsf{version}$: Get the current version number associated with a file.

For each function, $\mathcal{F}_{\mathsf{filesys}}$ only responds to the query if the client making the request has access to the folder.

Leakage of underlying filesystem. In order to allow our analysis to apply to many types of filesystems that implement this ideal functionality but may permit different levels of leakage, we define the leakage of the underlying filesystem in terms of the leakage of the filesystem API defined above. We use the leakage function $\mathcal{L}$ to denote the leakage of each filesystem function.

Ideal functionality for consensus. We also simplify our analysis by modeling our consensus algorithm as the ideal functionality $\mathcal{F}_{\mathsf{consensus}}$. We do this to focus on DORY's cryptographic guarantees and because two-phase commit is already well studied. $\mathcal{F}_{\mathsf{consensus}}$ simply serializes the requests from the client C before forwarding the requests to either the ideal search functionality $\mathcal{F}$ (in the ideal world) or the DORY protocol (in the real world).

Definition structure for search ideal functionality.

Ideal world model. We define the ideal world functionality as a function $\mathcal{F}$. The simulator $\mathcal{S}$ interacts with $\mathcal{F}$, and the adversary can interact with $\mathcal{S}$. The client C represents all the clients with access to the folder in question and can issue requests in parallel based on the API defined in Section 2.4. We only trust the clients with access to the folder in question; all other clients are controlled by $\mathcal{A}$. The client can also interact with the filesystem, modeled as $\mathcal{F}_{\mathsf{filesys}}$. At startup, the ideal world functionality takes as input the number of servers in the system and the identities of the servers that are or will be compromised. The ideal functionality in turn passes this information to the simulator $\mathcal{S}$, which can use this information to determine execution. This information dictates which of two cases $\mathcal{F}$ is in: either at least one server is honest, or no servers are honest. This means that the adversary $\mathcal{A}$ must statically declare its corruptions at the beginning of execution. As a reminder, the goal of DORY is to minimize leakage when at least one server is honest; we model both cases here to show that DORY still provides some level of protection in the event that all servers are compromised.

Real world model. The real world models DORY's execution. We describe the differences from the ideal world for each of the API calls with the client C and the adversary $\mathcal{A}$:

- $\mathsf{Update}(k, \mathsf{folderID}, \mathsf{docID}, \mathsf{prevWords}, \mathsf{currWords})$: In addition to what $\mathcal{S}$ receives in the ideal world, $\mathcal{A}$ additionally receives the update request sent by C, as specified in Section 2.4 (namely, the update requests received by the servers controlled by $\mathcal{A}$).
- $\mathsf{Search}(k, \mathsf{folderID}, \mathsf{keyword}) \rightarrow (\mathsf{docIDs})$: In addition to what $\mathcal{S}$ receives in the ideal world, $\mathcal{A}$ additionally receives some number of search requests sent by C, as specified in Section 2.4, depending on the number of servers compromised (namely, the search requests received by the servers controlled by $\mathcal{A}$).

Definition structure. We break our definition into two cases: (1) at least one server is honest, and (2) no servers are honest. These cases correspond to different attacker models in the real world, and different numbers of compromised servers given as input to $\mathcal{F}$ in the ideal world. We describe the ideal functionality and simulator for each case separately.

At least one honest server. We begin by describing the ideal functionality for the case where at least one server is honest.

Defining ideal functionality. We start by defining the ideal functionality $\mathcal{F}$ for the case where the adversary has statically compromised at most $\ell - 1$ of the ℓ servers. We denote the remaining honest server(s) as P. P also interacts with $\mathcal{F}$ to process requests. We define $\mathcal{F}$ for this case as follows:

- $\mathsf{Update}(\mathsf{folderID}, \mathsf{docID}, \mathsf{prevWords}, \mathsf{currWords})$:
  - $\mathcal{F}$ replaces the old keywords $\mathsf{prevWords}$ with the new keywords $\mathsf{currWords}$ for docID.
  - $\mathcal{F}$ sends $\mathcal{S}$:

    * the request type (Update),
    * folderID,
    * docID,
    * $\mathcal{L}(\mathcal{F}_{\mathsf{filesys}}.\mathsf{getCurrKey}(\mathsf{folderID}))$,
    * $\mathcal{L}(\mathcal{F}_{\mathsf{filesys}}.\mathsf{getDocKey}(\mathsf{docID}))$,
    * $\mathcal{L}(\mathcal{F}_{\mathsf{filesys}}.\mathsf{getVersion}(\mathsf{docID}))$, and
    * the bit β where $\beta = 1$ if the return value of $\mathcal{F}_{\mathsf{filesys}}.\mathsf{getCurrKey}(\mathsf{folderID})$ changed since the last update request; $\beta = 0$ otherwise.

- $\mathsf{Search}(\mathsf{folderID}, \mathsf{keyword}) \rightarrow (\mathsf{docIDs})$:
  - $\mathcal{F}$ builds Bloom filters for the documents in folderID and returns the documents docIDs that match the Bloom filter indexes for keyword.
  - $\mathcal{F}$ sends $\mathcal{S}$ the request type (Search) and, for each docID in folderID:
    * the request type (Search),
    * folderID,
    * $\mathcal{L}(\mathcal{F}_{\mathsf{filesys}}.\mathsf{getDocIDs}(\mathsf{folderID}))$, and
    * for each docID in folderID,
      · $\mathcal{L}(\mathcal{F}_{\mathsf{filesys}}.\mathsf{getVersion}(\mathsf{docID}))$, and
      · $\mathcal{L}(\mathcal{F}_{\mathsf{filesys}}.\mathsf{getDocKey}(\mathsf{docID}))$.

No honest servers. We now move to the case where no servers are honest and describe the ideal functionality for this case.

Defining ideal functionality. We define the ideal functionality $\mathcal{F}$ for the case where every server is compromised as follows. We represent leakage that allows the attacker to correlate queries by attaching an ID to each query. As the ideal functionality is only different from the case where at least one server is honest for search and is identical for update, we only describe search:

- $\mathsf{Search}(\mathsf{folderID}, \mathsf{keyword}) \rightarrow (\mathsf{docIDs})$:
  - $\mathcal{F}$ builds Bloom filters for the documents in folderID and returns the documents docIDs that match the Bloom filter indexes for keyword.
  - $\mathcal{F}$ sends $\mathcal{S}$ the request type (Search) and, for each docID in folderID:
    * the request type (Search),
    * folderID,
    * $\mathcal{L}(\mathcal{F}_{\mathsf{filesys}}.\mathsf{getDocIDs}(\mathsf{folderID}))$,
    * for each docID in folderID,
      · $\mathcal{L}(\mathcal{F}_{\mathsf{filesys}}.\mathsf{getVersion}(\mathsf{docID}))$ and
      · $\mathcal{L}(\mathcal{F}_{\mathsf{filesys}}.\mathsf{getDocKey}(\mathsf{docID}))$,
    * the current queryID, and
    * the queryIDs of all prior identical queries.

Defining security. We denote the security parameter as λ in our definition and proof.

Definition 1. Let Π be the protocol for an encrypted search system. Let $\mathcal{A}$ be an adversary that outputs a single bit. For any set of compromised servers ω statically chosen by $\mathcal{A}$, let ${}_{\Pi,\mathcal{A},\omega}(1^\lambda)$ be the random variable denoting $\mathcal{A}$'s output when interacting with the real world where clients only make queries to folders they have access to. For a simulator $\mathcal{S}$, an adversary $\mathcal{A}$ that outputs a single bit, and the set of compromised servers ω, let $\mathsf{Ideal}_{\mathcal{S},\mathcal{A},\omega}(1^\lambda)$ be the random variable denoting $\mathcal{A}$'s output when interacting with the ideal world where clients only make queries to folders they have access to.

We say that a protocol Π securely evaluates (with abort) the ideal functionality $\mathcal{F}$ defined above if there exists a non-uniform algorithm $\mathcal{S}$ probabilistic polynomial-time in λ such that for every non-uniform adversary $\mathcal{A}$ probabilistic polynomial-time in λ that outputs a single bit, for every set of compromised servers ω statically chosen by $\mathcal{A}$, the probability ensemble of ${}_{\Pi,\mathcal{A},\omega}(1^\lambda)$ over λ is computationally indistinguishable from the probability ensemble of $\mathsf{Ideal}_{\mathcal{S},\mathcal{A},\omega}(1^\lambda)$ over λ:

$$\exists \mathcal{S}\ \forall \mathcal{A}\ \forall \omega\ \{{}_{\Pi,\mathcal{A},\omega}(1^\lambda)\}_\lambda \approx_c \{\mathsf{Ideal}_{\mathcal{S},\mathcal{A},\omega}(1^\lambda)\}_\lambda\ .$$

2.9.2 Proof

Theorem 1. *Using the definitions in Section 2.9.1, DORY securely evaluates (with abort) the ideal functionality $\mathcal{F}$ when instantiated with a secure PRF, a secure aggregate MAC, a secure distributed point function, and a secure filesystem that implements the ideal filesystem functionality.*

Proof. We begin by defining the simulator $\mathcal{S}$. The simulator $\mathcal{S}$ takes in as input the total number of servers and the number of compromised servers. We initialize the system so that a Bloom filter is of length m and each keyword hashes to κ indexes. Based on the number of compromised servers, we define $\mathcal{S}$ in one of the two following ways.

Constructing the simulator with at least one honest server. We start by describing how to construct the simulator for the case where at least one server is honest; namely, adversary $\mathcal{A}$ has statically compromised at most $\ell - 1$ out of ℓ servers. We construct a simulator $\mathcal{S}$ for a single folder in the ideal world for this case as follows.

- We initialize $\mathcal{S}$ as follows:
  - $\mathcal{S}$ samples $k \xleftarrow{R} \{0,1\}^\lambda$.
  - For each folder, $\mathcal{S}$ keeps a Bloom filter table mapping document IDs to strings in $\{0,1\}^m$. Every entry in this table is initialized with 0^m.
  - $\mathcal{S}$ keeps a key table mapping document IDs to active keys. Every entry in this table is initialized with k.
  - $\mathcal{S}$ keeps a version number table mapping document IDs to version numbers. Every entry in this table is initialized with 0.
- When $\mathcal{S}$ receives an update request from $\mathcal{F}$ for docID:
  - If the bit $\beta = 1$, $k \xleftarrow{R} \{0,1\}^\lambda$.

  - $\mathcal{S}$ samples the string $b \xleftarrow{R} \{0,1\}^m$.
  - $\mathcal{S}$ retrieves the entry in the table for docID, called b'.
  - $\mathcal{S}$ looks up the version number for docID in the version number table, called version.
  - $\mathcal{S}$ looks up the current key for docID in the key table, called k'.
  - For $i \in [m]$:
    * $y_i \leftarrow \mathsf{MAC}(k, (b_i, \mathsf{docID}, i, \mathsf{version}+1))$
    * $y'_i \leftarrow \mathsf{MAC}(k', (b'_i, \mathsf{docID}, i, \mathsf{version}))$
    * $z_i \leftarrow y_i \oplus y'_i$
  - $\mathcal{S}$ increments the entry in the version number table for docID.
  - $\mathcal{S}$ sets the entry in the key table for docID to k.
  - $\mathcal{S}$ sets the entry in the Bloom filter table for docID to b.
  - $\mathcal{S}$ sends $(b, z_1, \ldots, z_m)$ to $\mathcal{A}$ along with the rest of the data that $\mathcal{S}$ received from $\mathcal{F}$.
  - $\mathcal{S}$ returns the response from $\mathcal{A}$.
- When $\mathcal{S}$ receives a search request from $\mathcal{F}$:
  - For $j \in [\kappa]$, $\mathcal{S}$ computes:
    * $i_j \xleftarrow{R} [m]$
    * $K_{j,1}, \ldots, K_{j,\ell-1} \leftarrow \mathsf{DPF.Gen}(i_j)$
  - $\mathcal{S}$ sends $K_{1,1}, \ldots, K_{\kappa,\ell-1}$ to $\mathcal{A}$ along with the rest of the data that $\mathcal{S}$ received from $\mathcal{F}$.
  - $\mathcal{S}$ returns the response from $\mathcal{A}$.

Constructing the simulator with no honest servers. We now construct a simulator $\mathcal{S}$ for a single folder in the ideal world for the case where the attacker has compromised every honest server. We have $\mathcal{S}$ respond to updates requests in the same way as the case where at least one server is honest, and so we only describe how $\mathcal{S}$ responds to searches in this case.

- We initialize $\mathcal{S}$ in the same way as in the case where at least one server is honest, but with one addition:
  - For each folder, $\mathcal{S}$ keeps a query ID table mapping sets of query IDs to κ indexes in $[m]$. This table is initially empty.
- When $\mathcal{S}$ receives a search request from $\mathcal{F}$:
  - $\mathcal{S}$ begins by looking up the queryIDs of all prior identical queries in the query ID table.
    * If there is an entry for the queryIDs, add the new queryID to the list of query IDs in that entry, and set $x_1, \ldots, x_\kappa \in [m]$ to be the indexes at that entry.
    * Otherwise, for $i \in [\kappa]$, sample $x_i \xleftarrow{R} [m]$ and add a new entry to the table: $\{\{\mathsf{queryID}\}, \{x_1, \ldots, x_\kappa\}\}$.
  - For $i \in [\kappa]$, $\mathcal{S}$ computes $K_{i,1}, \ldots, K_{i,\ell-1} \leftarrow \mathsf{DPF.Gen}(x_i)$.
  - $\mathcal{S}$ sends $K_{1,1}, \ldots, K_{\kappa,\ell}$ to $\mathcal{A}$ along with the rest of the data that $\mathcal{S}$ received from $\mathcal{F}$.
  - $\mathcal{S}$ returns the response from $\mathcal{A}$.

Putting it together. Lemma 2 proves that security holds against an adversary that can statically corrupt at most $\ell - 1$ of the ℓ servers, and Lemma 3 proves that security holds against an adversary that can corrupt all ℓ servers. Together these complete the proof. □

Lemma 2. *DORY securely evaluates (with abort) the ideal functionality $\mathcal{F}$ as defined in Definition 1 for the case when the adversary can statically corrupt at most $\ell - 1$ of the ℓ servers when instantiated with a secure PRF, a secure aggregate MAC scheme, a secure distributed point function, and a secure filesystem that implements the filesystem ideal functionality defined in Section 2.9.1.*

Proof. To prove the above lemma, we construct a series of hybrid protocols $\mathcal{H}_0, \dots, \mathcal{H}_3$. We show that for $i \in \{0, 1, 2\}$, the adversary $\mathcal{A}$ cannot distinguish between $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$.

Hybrid 0. We start off running the DORY protocol without any modifications (see pseudocode in Section 2.4.5).

Hybrid 1. We now modify the protocol from $\mathcal{H}_0$ such that for an update, rather than masking the m-bit Bloom filter $b \in \{0, 1\}^m$ with the output of a PRF evaluated at the document ID and its version number (obtained via $\mathcal{F}_{\mathsf{filesys}}.\mathsf{getVersion}$) keyed with $\mathcal{F}_{\mathsf{filesys}}.\mathsf{getCurrKey}$ evaluated at the folder ID, we mask b with a random $r \xleftarrow{R} \{0, 1\}^m$.

Updates in $\mathcal{H}_0$ and $\mathcal{H}_1$ are computationally indistinguishable. This follows from the security of the PRF (which ensures that the output of the PRF is computationally indistinguishable from random because the key can only be accessed by legitimate clients) and the underlying filesystem (which ensures that the PRF is evaluated on unique inputs so that a mask is never repeated and that the PRF key is only known to legitimate users). Because the updates in $\mathcal{H}_0$ and $\mathcal{H}_1$ are computationally indistinguishable, $\mathcal{A}$ cannot distinguish between $\mathcal{H}_0$ and $\mathcal{H}_1$.

Hybrid 2. We change the protocol from $\mathcal{H}_1$ to hide the contents of updates. For an update, $\mathcal{H}_1$ computes the Bloom filter $b \in \{0, 1\}^m$ and samples $r \xleftarrow{R} \{0, 1\}^m$ and sends $b \oplus r$. In $\mathcal{H}_2$, we instead just send r.

The value r is acting as a one-time pad, making the updates statistically indistinguishable, and so $\mathcal{A}$ cannot distinguish between $\mathcal{H}_1$ and $\mathcal{H}_2$.

Hybrid 3. To construct $\mathcal{H}_3$, we modify the search operation in $\mathcal{H}_2$ to argue that we don't leak information about the keyword being searched for. Rather than running $\mathsf{DPF.Gen}$ on the indexes corresponding to the keyword that the client is searching for, we instead sample $i_1, \dots, i_\kappa \xleftarrow{R} [m]$.

Because the adversary only gets access to at most $\ell - 1$ of the ℓ function shares, from the security of distributed point functions, the adversary cannot tell the difference between function shares chosen from random $i_1, \dots, i_\kappa$ and the indexes corresponding to the query, and so $\mathcal{A}$ cannot distinguish between $\mathcal{H}_2$ and $\mathcal{H}_3$. Notice that $\mathcal{H}_3$ is identical to $\mathcal{S}$ for the case where at most $\ell - 1$ out of ℓ servers are honest.

We now show that $\mathcal{H}_3$ cannot provide incorrect results without detection by the user. This follows directly from the security of the underlying MAC scheme; a user with access to the correct version numbers can verify that every bit in the search result was derived from the most recent update. Furthermore, as the MAC is generated using a key retrieved via $\mathcal{F}_{\mathsf{filesys}}.\mathsf{getCurrKey}$, only legitimate clients can generate MACs, and $\mathcal{A}$ cannot forge MACs. This ensures that the adversary cannot distinguish between $\mathcal{H}_3$ and the real world, thus completing the proof. □

Lemma 3. *DORY securely evaluates (with abort) the ideal functionality $\mathcal{F}$ as defined in Definition 1 for the case when the adversary can corrupt all ℓ servers when instantiated with a secure PRF, a secure aggregate MAC scheme, and a secure filesystem that implements the filesystem ideal functionality defined in Section 2.9.1.*

Proof. To prove the above lemma, we construct a series of hybrid protocols $\mathcal{H}_0, \ldots, \mathcal{H}_3$. We show that for $i \in \{0, 1, 2\}$, the adversary $\mathcal{A}$ cannot distinguish between $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$. Hybrids $\mathcal{H}_0, \mathcal{H}_1$, and $\mathcal{H}_2$ match that of Lemma 2, as they concern hiding the contents of updates, and the ideal functionality $\mathcal{F}$ and simulator $\mathcal{S}$ behavior for updates match in both cases.

Hybrid 0. (Same as $\mathcal{H}_0$ in Lemma 2.) We start off running the DORY protocol without any modifications (see pseudocode in Section 2.4.5).

Hybrid 1. (Same as $\mathcal{H}_1$ in Lemma 2.) We now modify the protocol from $\mathcal{H}_0$ such that for an update, rather than masking the m-bit Bloom filter $b \in \{0, 1\}^m$ with the output of a PRF evaluated at the document ID and its version number (obtained via $\mathcal{F}_{\mathsf{filesys}}.\mathsf{getVersion}$) keyed with $\mathcal{F}_{\mathsf{filesys}}.\mathsf{getCurrKey}$ evaluated at the folder ID, we mask b with a random $r \xleftarrow{R} \{0, 1\}^m$.

Updates in $\mathcal{H}_0$ and $\mathcal{H}_1$ are computationally indistinguishable. This follows from the security of the PRF (which ensures that the output of the PRF is computationally indistinguishable from random because the key can only be accessed by legitimate clients) and the underlying filesystem (which ensures that the PRF is evaluated on unique inputs so that a mask is never repeated and that the PRF key is only known to legitimate users). Because the updates in $\mathcal{H}_0$ and $\mathcal{H}_1$ are computationally indistinguishable, $\mathcal{A}$ cannot distinguish between $\mathcal{H}_0$ and $\mathcal{H}_1$.

Hybrid 2. (Same as $\mathcal{H}_2$ in Lemma 2.) We change the protocol from $\mathcal{H}_1$ to hide the contents of updates. For an update, $\mathcal{H}_1$ computes the Bloom filter $b \in \{0, 1\}^m$ and samples $r \xleftarrow{R} \{0, 1\}^m$ and sends $b \oplus r$. In $\mathcal{H}_2$, we instead just send r.

The value r is acting as a one-time pad, making the updates statistically indistinguishable, and so $\mathcal{A}$ cannot distinguish between $\mathcal{H}_1$ and $\mathcal{H}_2$.

Hybrid 3. To construct $\mathcal{H}_3$, we modify the search operation in $\mathcal{H}_2$ to argue that we only leak the set of previous queries corresponding to the same keyword. Rather than running $\mathsf{DPF.Gen}$ on the indexes corresponding to the keyword that the client is searching for, the client instead keeps a mapping of keywords to indexes. If a keyword has already been searched for, the client uses those indexes. Otherwise, the client samples a new set of indexes $i_1, \ldots, i_\kappa \xleftarrow{R} [m]$ and updates the mapping accordingly.

Because the adversary can see all ℓ function shares, it can tell which indexes the client is retrieving. However, because the client chooses the indexes as $\mathsf{PRF}(k, \mathsf{keyword})$ where k is obtained via $\mathcal{F}_{\mathsf{filesys}}.\mathsf{getCurrKey}$, the set of indexes queried is computationally indistinguishable from random (only legitimate clients have access to the key), and so $\mathcal{A}$ cannot distinguish between $\mathcal{H}_2$ and $\mathcal{H}_3$. Notice that $\mathcal{H}_3$ is identical to $\mathcal{S}$ for the case where all ℓ servers are compromised.

As in the proof for Lemma 2, $\mathcal{H}_3$ cannot provide incorrect results without detection by the client. This follows directly from the security of the underlying MAC scheme; a client with access to the correct version numbers can verify that every bit in the search result was derived from the most recent update. Furthermore, as the MAC is generated using a key retrieved via $\mathcal{F}_{\mathsf{filesys}}.\mathsf{getCurrKey}$, only

legitimate clients can generate MACs, and $\mathcal{A}$ cannot forge MACs. This ensures that the adversary cannot distinguish between $\mathcal{H}_3$ and the real world, thus completing the proof. □

2.9.3 Our questions for companies

For each company, we asked a set of questions over the course of discussion(s) and email exchanges. We asked these questions as we were in the process of designing our system. Some of the answers were already available in publicly available material; in these cases, we used this information and did not repeat the question during the course of our discussion. As many of these questions were asked over the course of a discussion, we did not use the same wording every time, but summarize our questions below for reference:

Encrypted search use-case.

1. Do you have a need for encrypted search?
2. What settings do you need encrypted search for? Mobile? Desktop? Offline files?
3. Have you explored implementing encrypted search? If so, what progress did you make and what, if any, challenges did you encounter?

Performance and cost.

1. What are the requirements for the overhead of search?
   a) If cost mentioned: How much would you be willing to pay per user per month to support search?
   b) If speed mentioned: What maximum end-to-end search latency would you consider acceptable to deploy? Is a linear scan over the documents acceptable for searching, provided the overall latency was low?
2. What are the requirements for the overhead of updates? Is it feasible to perform an expensive cryptographic operation such as an ORAM write for each keyword in a document?
3. How do you handle membership changes and, in particular, revocation? Would you accept a solution that required you to recompute the entire search index when a user's access is revoked?

Workload.

1. What is the average and maximum number of files each user has access to?
2. What do you anticipate will be the ratio of updates to searches?
3. How many updates on average does a user currently perform each day?

Splitting trust.

1. Are you already splitting trust in your system? If so, how?
2. Would you consider deploying a solution that split trust between multiple servers? What trust guarantees would you require to consider a multi-server solution?
3. If yes to the above question: Is it acceptable to split trust to hide the contents of the search index?

Other relaxations.

1. Is it acceptable if there is some delay between when updates are performed and when search results are returned?
2. Is it acceptable to allow a small (configurable) number of false positives in the search results?

2.10 Conclusion

DORY is an encrypted search system that distributes trust to meet real-world efficiency and security requirements. By reexamining the system model, we are able to build a system that is performant without leaking search access patterns.

Chapter 3

Waldo: A private time-series database

3.1 Introduction

Organizations today rely on the ability to continuously collect and analyze time-series data. To cheaply store and query this data, organizations turn to cloud databases [24, 260, 484]. However, many systems produce time-series data that is not only useful, but also *sensitive*. For example, remote patient monitoring systems and smart homes both generate time-series data that users might not want to store in the cloud due to the danger of data breaches [335, 374, 391, 431].

One solution to this problem is to perform queries over encrypted time-series data, as Timecrypt [83] and Zeph [84] do. These systems have two serious limitations. The first is that they only support aggregation by time over a data stream (e.g. average heart rate over a week). Many modern time-series databases [30, 260, 388, 484] support *multidimensional* data and allow users to filter based on different predicates that are not predefined. Multi-predicate queries are critical for some applications. For example, a doctor might want to run the following query to assess congestive heart failure risk without revealing the filter values or query result to the server [474]:

```
SELECT COUNT(*) FROM MedicalHistory
WHERE (systolic < 90 OR diastolic < 50 OR
    weight_gain > 2 OR heart_rate < 40
    OR heart_rate > 90) AND (time BETWEEN
    2021:07:01:00:00 AND 2021:08:01:00:00)
```

The second limitation is that Timecrypt and Zeph reveal the query time interval to the server, which could be problematic for some applications. For example, if a doctor is querying for a patient's heart rate, the queried time period could reveal when the patient had a heart attack or started a new medication. To address the first limitation (functionality), we could leverage techniques from encrypted databases [188, 247, 396, 398, 407, 413, 452, 493, 537]. While many of these systems can support multi-predicate queries, they generally achieve good performance by permitting some leakage, which an attacker can exploit to learn information about the query and the database contents

(e.g. the attacker could learn the patient's blood pressure) [94, 223, 224, 230, 269, 284, 292, 293, 305, 393, 410, 534].

Both oblivious RAM (ORAM) [206, 392] and general-purpose secure multiparty computation (MPC) [55, 205, 525] are natural tools for this problem. ORAM is suited to the trusted proxy setting (common in encrypted databases [396, 407, 413]), and MPC works in the distributed-trust setting where servers are deployed in different trust domains (the same setting used in Zeph [84]). Unfortunately, both are prohibitively expensive for the time-series setting. Storing a multidimensional tree in ORAM makes it possible to execute queries in polylogarithmic time, but appends are just as or more expensive than executing queries and require many round-trips, which results in poor throughput due to the append-intensive nature of time-series workloads (Section 3.7). General-purpose MPC is also a poor fit, as existing tools require massive amounts of communication (Section 3.7). In a distributed-trust setting in which servers are likely deployed in different clouds to minimize the chance that multiple servers are compromised, many round-trips and large bandwidth imply high latency and high monetary cost.

We present Waldo, an oblivious, maliciously secure time-series database that leverages distributed trust. Waldo provides:

- **Multi-predicate functionality:** Waldo provides two types of indices: one that supports additive aggregates (e.g. sum, count, mean, variance, standard deviation) based on multiple predicates (Section 3.4) and another that supports arbitrary aggregates (e.g. max, min, top-k) over a time interval (Section 3.5). Prior work [83, 84] only supports aggregation over time.
- **Obliviousness with malicious security:** Waldo distributes trust to protect not only the data contents, but also the query filter values and search access patterns (Section 3.3). Our design uses three servers and provides malicious security when at least two servers are honest.
- **Efficiency:** We implement and evaluate Waldo (Section 3.6, Section 3.7) on a set of 32-core machines. With features modulo 2^8, Waldo (specifically our index with multi-predicate support in Section 3.4) runs a query with 8 range predicates for 2^{10} records in 0.22s, compared with 1.75s for an MPC baseline and 9.60s for an ORAM baseline, and 2^{20} records in 11.82s, compared with 45.72s for MPC and 16.70s for ORAM. The MPC baseline uses $9-82\times$ more bandwidth between servers than Waldo for 2^{10} to 2^{20} records, and the ORAM baseline uses $20-152\times$ more bandwidth between the client and server(s) than Waldo for 1-10 predicates. Waldo is also highly parallelizable.

3.1.1 Summary of techniques

As we show in Section 3.7, ORAM and general-purpose MPC are poorly suited to the time-series database setting due to the many rounds of interaction (ORAM) and substantial communication overhead (MPC) they require. We design Waldo to overcome these shortcomings and be efficient when servers are in different trust domains: we need to rely less on communication, which is limited and expensive [1], and instead take advantage of compute resources, which are significantly cheaper and easy to increase. With this goal in mind, we turn to function secret sharing (FSS) [77, 78], a recent cryptographic tool that allows the client to generate compact shares of a function that the

servers can then use to evaluate the corresponding equality or range predicate without learning what the predicate is (critical for security). Crucially, the servers can evaluate their shares of the predicate without interaction (in contrast, other state-of-the-art MPC techniques require interaction proportional to the number of comparisons).

At its core, FSS is a simple primitive designed for semihonest servers with public inputs where efficient implementations exist for a limited class of functions [77, 78]. The high-level challenge in Waldo is to adapt this fairly simple primitive to the much more complex encrypted time-series database setting where there are *malicious servers, secret inputs, and chained predicates*. Prior work has explored applying FSS in different settings that require some combination of private data, malicious security, and complex queries [75, 79, 82, 140, 161, 508], but as we discuss below (and in Section 3.9), these techniques do not easily translate to our setting. These shortcomings motivate the techniques we develop in Waldo, which we summarize below.

FSS for private predicates (Section 3.4.1). Using FSS to evaluate predicates on private data is not straightforward because, for correctness, servers evaluating function shares must provide the same input. Providing the input in plaintext is clearly a problem for implementing equality or range predicates where the server should not know the values being compared. To circumvent this problem, prior works on FSS for secure computation [75, 79] use additive masks to hide secret values, but as we discuss in Section 3.4.1, this technique is highly inefficient in our setting, requiring client communication linear in the database size. To solve this problem, we develop a *shared one-hot index*, which hides the contents of the data while supporting high-throughput appends and private queries with FSS. The way in which we split our index across the servers is inspired by Bunn et al.'s distributed ORAM [82] that combines FSS with replicated secret sharing [31]. However, distributed ORAM only requires a block storage abstraction, whereas we need to evaluate different range and equality predicates on the database contents. We introduce new techniques that build on top of FSS and replicated secret sharing to obliviously evaluate predicates (Section 3.4.1).

Combining multiple predicates (Section 3.4.2). To support multi-predicate queries, we need a mechanism for efficiently combining the outputs of equality and range queries. There are two critical challenges here: (1) how to structure the outputs of the FSS evaluations so that they can be efficiently merged, and (2) how to perform the actual merging. To solve (1), we design our shared one-hot index such that the FSS evaluation output is a vector of zeroes and ones that can easily be combined with a vector for another predicate. Then, to address (2), we leverage the fact that our vectors are shared using replicated secret sharing to take advantage of existing communication-efficient techniques for semihonest 3-party honest-majority multiplication [31, 479]. Our techniques for combining predicates are effective for computing count, sum, and, by extension, mean, variance, and standard deviation.

Supporting complex aggregates (Section 3.5). The above protocol supports complex filtering, but a limited set of aggregates. To support more complex aggregates (e.g. max, min, top-k), we show how to build a *shared aggregate tree* that supports any user-defined aggregation function where the server does not have to know how values are aggregated. Like our shared one-hot index, our shared aggregate tree uses FSS and replicated secret sharing to hide the query and data. Our shared aggregate tree only supports aggregation over time, but notably, queries do not require any server

interaction. Furthermore, server execution time is independent of the aggregation function.

Providing malicious security (Section 3.4.3, Section 3.5). For both types of queries, we need to defend against a malicious adversary that might try to tamper with the query results. Some 3-party honest-majority MPC protocols can rely on replication coupled with cut-and-choose [189, 365] or triple sacrifice techniques [164], but these solutions don't work for us because we use 2-party FSS. We need to authenticate the results of the FSS evaluations in a way that is compatible with our techniques for combining outputs from multiple predicates. Our solution is inspired by Boyle et al.'s use of information-theoretic MACs [56, 126, 133] for authenticating FSS evaluation outputs [75]. The challenge is to make this approach compatible with multiplications: the multiplication protocol that we use for combining multiple predicate outputs is very efficient, but designed for the semihonest setting [31, 479]. We show how to use authenticated outputs from FSS evaluations to securely chain multiplications together such that the client only needs to check the integrity of the final result and a random linear combination of intermediate results.

3.2 System overview

3.2.1 Time-series workloads

Waldo accounts for these elements of time-series workloads:

- **Append-only:** Time-series databases tend to be append-only, as records represent data captured at some timestamp [485].
- **Write-intensive:** Time-series workloads have a high ratio of appends to queries, and so the database must be able to process a large volume of appends quickly [261, 367].
- **Multiple features, multiple predicates:** The data recorded for a timestamp often has multiple features. Therefore, queries with predicates corresponding to more than one feature are very common [260, 421].
- **Recent data is more valuable:** Even though the number of records grows rapidly over time (time-series workloads are append-only and append-intensive), the most recent data is the most relevant, and the value of data decays over time [261].
- **Aggregation:** Aggregation queries are very common in time-series workloads. Plaintext time-series databases build specialized indices to quickly aggregate data [27, 484]

3.2.2 Running example: Remote Patient Monitoring

Time-series data is critical to many applications, including smart homes [387], smart cars [458], energy conservation [460], and industrial internet-of-things (IoT) [486]. We now discuss remote patient monitoring as a running example (although Waldo can support a wide variety of applications). Remote patient monitoring systems allow doctors to use sensors to monitor at-risk patients while they are home. The COVID-19 pandemic has made these tools even more critical, with more patients

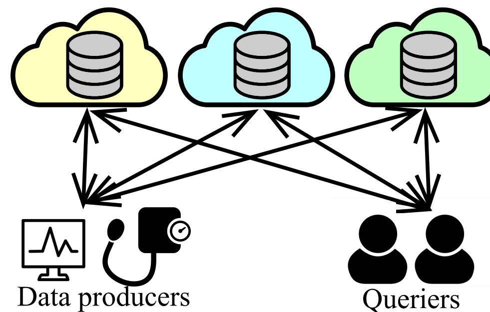


Figure 1: System architecture. Here the ECG sensor and blood pressure monitor are data producers, and the doctors are queriers. The servers are deployed in different trust domains.

opting for telehealth visits and the federal government expanding Medicare coverage to remote patient monitoring [383].

RPM can be particularly valuable for at-risk patients to manage conditions such as hypertension, chronic obstructive pulmonary disease, diabetes, and asthma [418]. In some cases, doctors only need to monitor a single vital sign (e.g. glucose levels in a diabetes patient), but in a growing number of cases, doctors find it valuable to make decisions based on more biometric data (e.g. blood pressure, heart rate, and weight [474]).

One challenge for remote patient monitoring is that this data is extremely sensitive, and the query itself can reveal information about a patient's condition. For example, the threshold vital signs that a doctor checks for may reveal if a patient is diabetic. Waldo ensures that the attacker only learns the database schema and the structure, origin, and timing of queries (Section 3.3).

3.2.3 System architecture

The Waldo system is composed of the following entities (Figure 1):

- **Clients:** There are two types of clients: data producers and queriers. Some clients may be both.
  - **Data producers:** Sensors or other devices collect real-time data and update the servers' state.
  - **Queriers:** Queriers query the data collected by the data producers and stored at the servers.
- **Servers:** Three servers in different trust domains store data collected by data producers and execute queries made by queriers. If a majority of the servers are honest, the single malicious server cannot not learn the data contents, query filter values, or any search access patterns. These "logical" servers might be distributed across multiple machines.

Because Waldo leverages distributed trust (Section 3.3), each server should be deployed in a different trust domain. This could mean that the servers are hosted in different clouds, managed by different, potentially competing organizations, and/or deployed in different jurisdictions. Clients send messages directly to each of the three servers. This is in contrast to prior works that use a trusted proxy [396, 407, 413]: with a trusted proxy, users route their queries through a computationally powerful machine that interacts with the server on behalf of the clients. The proxy model has the

disadvantage that the clients must set up a powerful, trusted-by-all machine rather than outsourcing computation to the cloud. In Waldo, both storage and computation are outsourced, and clients interact directly with the untrusted servers.

3.2.4 Waldo API

We now describe the API that clients use to interact with Waldo. Waldo exposes two types of indices to clients: WaldoTable and WaldoTree. WaldoTable stores multiple features for a single timestamp and supports multi-predicate queries. WaldoTree, on the other hand, stores a single feature for a timestamp and only supports queries over a time range. While WaldoTable supports more complex multi-predicate filtering, WaldoTree supports a larger class of aggregation functions and is more performant. WaldoTree is also useful for queries with predefined filters (as it is faster and uses less storage than WaldoTable), whereas WaldoTable is useful when queries are unpredictable. Both types of indices support Init, Append, and Query operations where Init and Append are invoked by data producers, and Query is invoked by the queriers. All routines trigger execution at the servers. We describe the API for both below.

WaldoTable**:**

- $\mathsf{Init}(1^\lambda, 1^{\tilde{s}}, \mathsf{schema})$: Initialize a table index given computational security parameter λ, statistical security parameter $\tilde{s}$, and a schema layout parameter $\mathsf{schema} = (N, F, 2^{\ell_1}, \dots, 2^{\ell_F})$ where N is the number of records in the window, F is the number of features associated with a timestamp, and 2^{ℓ_i} is the feature size for feature i.
- $\mathsf{Append}(t, v_1, \dots, v_F)$: Update the table index to store record with timestamp t and values $v_1 \in \mathbb{Z}_{2^{\ell_1}}, \dots, v_F \in \mathbb{Z}_{2^{\ell_F}}$.
- $\mathsf{Query}(P_1 \wedge \dots \wedge P_n, \mathsf{feature}, \mathsf{type}) \to x$: Aggregate by type for feature over the boolean formula composed of predicates $P_1, \dots, P_n$, where each P_i implicitly conveys the feature it applies to. Here type $\in$ {`count, sum, mean, variance, stdev`}. Output the aggregate of values in feature filtered by the query predicates (or abort if integrity checks fail).

WaldoTree**:**

- $\mathsf{Init}(1^\lambda, 1^{\tilde{s}}, \mathsf{schema})$. Initialize a tree index given computational security parameter λ, statistical security parameter $\tilde{s}$, and a schema layout parameter $\mathsf{schema} = (2^\ell, \mathsf{type})$, where 2^ℓ is the feature size of the values in the tree index and type is any user-defined aggregation function (Section 3.5).
- $\mathsf{Append}(t, v)$: Update the tree index to store record with timestamp t and value $v \in \mathbb{Z}_{2^\ell}$.
- $\mathsf{Query}(t_1, t_2) \to x$: Aggregate over time interval (t_1, t_2) by type (set during initialization) and output the result (or abort if integrity checks fail).

In WaldoTable, the schema parameter takes a number of records in the window, N. Because records are constantly appended, N is not the total number of records in the table, but rather the number of most recent records that the client can query (recall that the most recent data is typically the most valuable). The parameter N represents a tradeoff between performance (smaller N values result in better performance) and query expressiveness (clients might want access to data further in

the past). WaldoTable can easily be extended to support multiple window sizes N if the client is willing to reveal which window size is being used for the current query.

WaldoTable supports both equality ($x = a$) and range ($a < x < b$) predicates. Clients can chain predicates together using AND operations. The servers can easily compute NOTs (Section 3.4.2), and so we can express any combination of ANDs, ORs, and NOTs via De Morgan's laws in a WaldoTable query.

Access control. Waldo enforces different permissions for clients across different tables. Because the servers know the identity of the client, access control is straightforward: all three servers must participate to compute a query, and we allow at most one server to be malicious, so we can restrict the types of queries that different users are allowed to make using a standard database access control list [145]. Each server checks a client's permissions, and if a client doesn't have permission for that operation, the honest servers will simply refuse to participate. Note that the servers can only enforce permissions for the parts of the query that are public (e.g. the data that the query is executing on and the query structure), but not the parts that remain private. Also, permission to make some types of queries (e.g. mean) implicitly gives permission to view some intermediate values (e.g. sum and count to compute mean). Access control can be at the record level in WaldoTable and at the database-table level in WaldoTree. For simplicity, when describing the design of Waldo, we focus on the case of a single table, as it is straightforward to extend this design to multiple tables with access control.

3.2.5 Notation

In Waldo, we consider all database values $\in \mathbb{Z}_{2^\ell}$ where ℓ depends on the feature size defined in schema. Waldo uses secret-sharing to split a value $x \in \mathbb{Z}_{2^\ell}$ into parts $[x]_1, \ldots, [x]_p \in \mathbb{Z}_{2^\ell}$ where $p \in \{2, 3\}$ such that $x = (\sum_{i=1}^{p} [x]_i) \bmod 2^\ell$. Note that we can sample shares in $\mathbb{Z}_{2^m}$ for $m \geq \ell$ to represent values in $\mathbb{Z}_{2^\ell}$. We sometimes use $x_1, \ldots, x_p$ and $x^{(1)}, \ldots, x^{(p)}$ to also refer to the secret shares of x, and within a server's context, we sometimes drop the share subscript altogether. All arithmetic operations such as $(+, -, \cdot)$ correspond to ring operations. Arithmetic operations on vectors refers to their component-wise application in the underlying ring. $[N]$ denotes the set $\{0, \ldots, N-1\}$. We use $a \leftarrow b$ to denote assignment of b's value to a, and $a \xleftarrow{R} \mathbb{R}$ denotes randomly sampling a from ring $\mathbb{R}$. We denote the computational security parameter as λ and the statistical security parameter as $\tilde{s}$.

3.3 Threat model and security guarantees

We describe Waldo's security guarantees and, due to space constraints, delegate Waldo's formalism (detailing its guarantees) to Section 3.10. Waldo operates in the malicious three-party honest-majority setting, meaning that it provides security with abort if at most one server is malicious. In the malicious threat model, the attacker can influence the server's behavior arbitrarily. If a server is malicious, the client does not receive output and only learns that an error occurred.

If at most one server is corrupted, then Waldo guarantees that the attacker does not learn the record contents, query filter values, or any search access patterns and only learns public information. The public information available to the attacker is: (1) the database schema (i.e. for each table, the number of records, number of features, and the size of each feature); (2) the structure of the query (i.e. the number of predicates, the type of each predicate, the structure of conjunctions and negations, the feature being aggregated, and, in the case of WaldoTable, the aggregation function); and (3) when a query is performed and which client performed the query. The predicate type includes whether the predicate is an equality or range (single-sided or interval) predicate and the feature corresponding to the predicate. To make the query structure leakage more concrete, we consider the congestive heart failure query in Section 3.1: the attacker learns that the query is computing `COUNT(*)` for `(RANGE(f1) OR RANGE(f2) OR RANGE(f3) OR RANGE(f4) OR RANGE(f5)) AND (RANGE(f6) AND RANGE(f6))` where `RANGE` implies a single-sided range predicate and the mapping of features to feature ID is consistent across queries. Expressing queries in terms of some query "normal form" with dummy predicates could eliminate leakage due to query structure, although this would negatively impact performance and query expressiveness. As discussed in Section 3.2.4, for some types of queries, clients are able to learn intermediate values (e.g. the client learns sum and count when running a mean query).

Notably, Waldo does not reveal any information about the filter values, which records are selected in a query, or how many records are selected, among other potential sources of leakage; protecting access patterns and volume leakage defends against a large class of leakage-abuse attacks [94, 223, 225, 230, 269, 284, 292, 293, 305, 393, 410, 534]. Because Waldo is maliciously secure, the client can check the integrity of the query result. If at most one server is corrupted, Waldo ensures that only clients granted permission to make queries or updates to a given table are able to perform those operations. Waldo does not provide availability if any one server refuses to provide service.

We formally model the end-to-end security guarantees of Waldo by defining an ideal functionality $\mathcal{F}$ that specifies the behavior of an ideal system, capturing the properties discussed above. $\mathcal{F}$ additionally captures the fact that the client can verify the integrity of the result. In Section 3.10, we present a formal definition of security using $\mathcal{F}$, which we use in the following theorem:

Theorem 2. Using Definition 4 (Section 3.10), Waldo securely evaluates (with abort) the ideal functionality $\mathcal{F}$ (Section 3.10) when instantiated with secure distributed point and comparison functions and a pseudo-random function, all with a computational security parameter of λ.

We include the full proof in Section 3.10.

3.4 Multi-predicate queries

In this section, we describe how to implement the WaldoTable API to filter on multiple predicates. We will start with a strawman that provides limited functionality and incomplete security and show how to modify our scheme to support the full query functionality and security guarantees we want.

3.4.1 Single predicate with semihonest security

Our first step is to choose a building block to help us obliviously filter by predicates. As we discussed previously, ORAM and generic MPC are natural candidates, but these solutions perform poorly in the time-series setting (Section 3.7). We instead identified two-party function secret-sharing to be an excellent fit for equality and range predicates.

Tool: Function Secret Sharing (FSS). Two-party function secret sharing (FSS) makes it possible to split a function f into succinct function shares such that any strict subset of the shares doesn't reveal anything about the function f, but when the evaluations at a given point x are combined, the result is $f(x)$. A two-party FSS scheme is defined by the following algorithms:

- $\mathsf{Gen}(1^\lambda, f) \rightarrow K_1, K_2$: Given the security parameter 1^λ and a function description f, output keys K_1 and K_2.
- $\mathsf{Eval}(K_i, x) \rightarrow y_i$: Given the key K_i and input x, output value y_i, corresponding to this party's share of $f(x)$. We assume that key K_i implicitly contains the party index i.

Adding together the two outputs of Eval y_1, y_2 yields $f(x)$.

We identify two FSS constructions as a natural fit for Waldo: distributed point functions and distributed comparison functions [75, 77, 78]. Distributed point functions (DPFs) are FSS schemes for the point function $f_{\alpha,\beta}$ where $f_{\alpha,\beta}(\alpha) = \beta$ and $f_{\alpha,\beta}(\alpha') = 0$ for all $\alpha' \neq \alpha$ [77, 78]. Similarly distributed comparison functions (DCFs) are for functions $g_{\alpha,\beta}$ where $g_{\alpha,\beta}(x) = \beta$ if $x < \alpha$ and $g_{\alpha,\beta}(x) = 0$ otherwise [75]. Analogously, DCFs can also describe predicates $x > \alpha$. Constructions for interval containment (IC) build on DCFs to express functions of the form $a < x < b$ [79]. Throughout the chapter, we will use $\mathsf{Gen}^{=}(1^\lambda, \alpha, \beta)$ and $\mathsf{Gen}^{<}(1^\lambda, \alpha, \beta)$ to refer to FSS generator algorithms for DPFs and DCFs respectively. For $a < x < b$ predicates, we use the IC construction from Boyle et al. [79] that requires 2 DCF keys per IC, and we refer to its generator as $\mathsf{Gen}^{\mathsf{IC}}(1^\lambda, a, b, \beta)$. For all these cases, we refer to the evaluation algorithms as Eval, and we assume the keys implicitly convey the type of algorithm being invoked.

DPFs are a natural fit for equality queries, and DCFs are a natural fit for range queries.

FSS for private data. Applying FSS to filter *public* data based on an equality or range predicate is fairly straightforward [75, 78, 508]. Two servers store identical copies of a public database (here the database is just a list of values). To privately query the database for the number of records matching a predicate, the client generates FSS keys with $\beta = 1$ for the equality or range predicate using a DPF or a DCF and sends a key to each server. Each server evaluates its key on each value in the database, sums the evaluations together, and sends the results back to the client, computing $\sum_{i=1}^{N} \mathsf{Eval}(K, d_i)$ for a database composed of values $d_i, \ldots, d_N$ with FSS key K on server s. When the client sums the results from each server, it obtains the number of records matching the predicate.

Leveraging FSS to search over *private* data, however, introduces a new challenge: the server cannot simply evaluate its FSS key on the database contents because the server should not be able to view the database contents. At the same time, the servers need to evaluate their keys on identical copies of the database to produce correct outputs. Prior works on using FSS for secure computation [75, 79] keep values secret by ensuring that the servers hold additively masked versions of the secret. To output shares of $f(x)$ instead of $f(x+r)$ (where f is the shared function, x is the

secret input, and r is the mask), they rely on sharing the matching function $\tilde{f}_r = f(x - r)$. In the database setting, each entry x_i must be masked with an independently sampled r_i. Thus we would need a different $\tilde{f}_{r_i}$ for each database entry x_i even though the servers only need to evaluate a single function f. This practically means that the size of the function shares would match the size of the database, defeating the purpose of using FSS to minimize communication. Therefore, we need different techniques for the encrypted time-series database setting.

Our solution to this problem is inspired by that of DORY [140]. For each feature, we build a table of size $N \times 2^\ell$ where N is the number of records that can be queried (the window size from Section 3.2.4) and 2^ℓ is the feature size (i.e. the number of possible values for that feature). For each record, the corresponding row in the table is set to "1" at the location corresponding to record value and "0" elsewhere. We call this structure a *one-hot index*, as it is a table of "one-hot" vectors, and we use this tool as a building block to construct a *shared one-hot index*. While the construction of the core one-hot index is very similar to the data structure in Dory, the shared one-hot index we construct from it provides more powerful functionality and guarantees confidentiality and integrity using different techniques, as we discuss later.

Now we can leverage FSS using the *structure* rather than *content* of the search index. We want to compute the number of records matching the predicate. For every entry $d_{i,j}$ in the table D for record $i \in [N]$ and feature value $j \in [2^\ell]$, the server s evaluates its FSS key K on the current value j, multiplies the evaluation by the table entry $d_{i,j}$, and then computes

$$\mathsf{EvalPred}(s, K, D) \leftarrow \sum_{j=0}^{2^\ell - 1} \left(\mathsf{Eval}(K, j) \cdot \sum_{i=0}^{N} d_{i,j} \right) \tag{3.1}$$

There are two remaining challenges here. First, we need to understand how to encode different types of record values using a small feature size 2^ℓ, as the computation required is $O(N \cdot 2^\ell)$. Second, while this clearly works if $d_{i,j} \in \{0, 1\}$, if the values $d_{i,j}$ are encrypted, then the summation will not produce the correct result. We address both below.

Encoding values with a small feature size. Choosing a small feature size 2^ℓ is critical for good performance in WaldoTable. For the remote patient monitoring applications we examine, we find that all sensitive fields with a predicate computed over them are already from a small domain (size 2^8) or can easily be mapped to one (Section 3.7.2). Notably only the values being compared in predicates need to use small feature sizes; the values being aggregated are not subject to these restrictions. We summarize three techniques for encoding values in a large domain using a small feature size below.

One way to represent a large set of values using a small feature size is by bucketing intervals in $\mathbb{Z}_{2^\ell}$, improving performance at the expense of precision. Bucketing preserves ordering for range predicates and is used in prior work [20, 84, 121] to efficiently compute aggregate statistics. Candidates for bucketing include attributes such as weight, blood glucose level, salary, GPA, or percentages.

Hash maps or Bloom filters can compress large values for point queries where high precision is required (for Bloom filters, the client needs to check that each bit at each hash location is 1). One

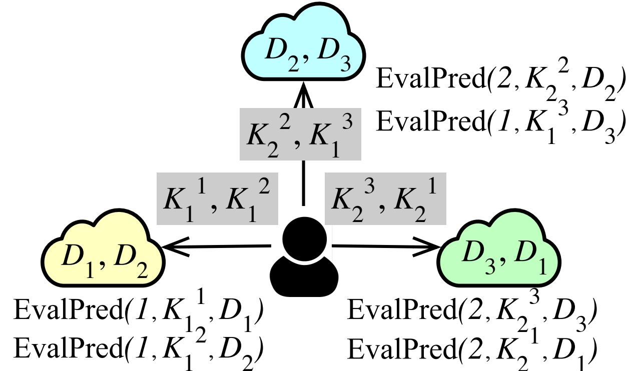


Figure 2: Query predicate evaluation with FSS and RSS.

example of a field that can be represented in this way and is only used in point queries is an identifier (e.g. a client ID, company ID, social security number, or phone number).

Large domains can also be represented via a conjunction of predicates. For example, a time can be represented as a timestamp or as a conjunction of the year, month, day, hour, and minute. The number of predicates can leak information about the resulting filter, although this leakage can be eliminated by always using the maximum number of predicates.

Identifying replicated secret sharing (RSS). To solve the second problem (protecting the database contents while computing the correct sum), we turn to secret sharing. In standard 2-out-of-2 secret sharing, to secret share a value $x \in \mathbb{Z}_{2^k}$, we sample shares $[x]_1, [x]_2 \xleftarrow{R} \mathbb{Z}_{2^k}$ such that $x = [x]_1 + [x]_2$. Since the output of Eval is 2-out-of-2 shares of $f(x)$, if the table D is also shared in the same way, then each of the 2^ℓ multiplications in Equation (3.1) will require expensive MPC tools [48]. While 2^ℓ secure multiplications might seem feasible, to chain together predicates, we will show in Section 3.4.2 that we need to perform $N \cdot 2^\ell$ multiplications to evaluate a single predicate, which is impractical if these multiplications must use Beaver triples [48]. To overcome this challenge, we leverage replicated secret sharing, which hides the data contents while only requiring *local multiplications*. Replicated secret sharing requires switching from two servers to three servers, but this third server allows us to significantly improve performance. This combination of RSS with FSS makes single-depth multiplications essentially free (no communication required).

We use the 2-out-of-3 replicated secret sharing from Araki et al. [31]. To secret share a value $x \in \mathbb{Z}_{2^k}$, sample shares $[x]_1, [x]_2, [x]_3 \xleftarrow{R} \mathbb{Z}_{2^k}$ such that $x = [x]_1 + [x]_2 + [x]_3$. Each server gets a pair of shares: S_1 has $([x]_1, [x]_2)$, S_2 has $([x]_2, [x]_3)$, and S_3 has $([x]_3, [x]_1)$.

Layering RSS with FSS. RSS provides the replication necessary for FSS without sacrificing confidentiality. If the database is split into shares $[D]_1, [D]_2, [D]_3$, then the client can generate three pairs of FSS keys (K_1^1, K_2^1), (K_1^2, K_2^2), and (K_1^3, K_2^3). Each server's share of the database is a pair $(D.\mathsf{first}, D.\mathsf{second})$, where S_1 has $([D]_1, [D]_2)$, S_2 has $([D]_2, [D]_3)$, and S_3 has $([D]_3, [D]_1)$. The client sends S_1 (K_1^1, K_1^2), S_2 (K_2^2, K_1^3), and S_3 (K_2^3, K_2^1)). Then

- S_1 computes $x_1^{(1)} \leftarrow \mathsf{EvalPred}(1, K_1^1, [D]_1)$ and
  $x_1^{(2)} \leftarrow \mathsf{EvalPred}(1, K_1^2, [D]_2)$,
- S_2 computes $x_2^{(2)} \leftarrow \mathsf{EvalPred}(2, K_2^2, [D]_2)$ and

$x_1^{(3)} \leftarrow \mathsf{EvalPred}(2, K_1^3, [D]_3)$, and

- S_3 computes $x_2^{(3)} \leftarrow \mathsf{EvalPred}(3, K_2^3, [D]_3)$ and $x_2^{(1)} \leftarrow \mathsf{EvalPred}(3, K_2^1, [D]_1)$.

Client can fetch these and compute $x^{(1)} \leftarrow x_1^{(1)} + x_2^{(1)}$, $x^{(2)} \leftarrow x_1^{(2)} + x_2^{(2)}$, $x^{(3)} \leftarrow x_1^{(3)} + x_2^{(3)}$, and $x \leftarrow x^{(1)} + x^{(2)} + x^{(3)}$. This way, RSS allows us to hide contents of the data from servers and evaluate a single predicate without communication between servers (Figure 2). We call this data structure (and the corresponding API for appending to and querying it) a *shared one-hot index*.

Appends to this shared one-hot index are straightforward: to append value v_i corresponding to feature i with feature size 2^{ℓ_i}, the data producer generates a one-hot vector $V \in \mathbb{Z}_{2^k}^{2^{\ell_i}}$ where $V_j = 1$ if $j = v_i$ and $V_j = 0$ if $j \neq v_i$. Then, the data producer splits V component-wise into RSS shares and sends a pair of shares to each server. Each server appends each share to its corresponding table. Note that we do *not* support private updates to existing records, and so we only consider append-only workloads like time-series. Bunn et al. [82] also explore how FSS and RSS compliment each other in the distributed ORAM setting. They show how to provide a private key-value store interface, whereas we build a data structure that can handle more complex queries while only requiring a small number of FSS keys.

3.4.2 Multiple predicates with semihonest security

So far, we have focused on evaluating a single predicate, but our goal is to filter records based on a combination of multiple predicates. We have also focused only on counting the number of records matching a predicate, but in practice we additionally want to compute sums (below we describe how to use sum and count as a foundation for other aggregates).

To support both of these, we transition from each server computing a single value (the number of records matching the predicate) to each server computing a filter (a vector of size N where the value at index $i = 1$ if record $i \in [N]$ matches the predicate). Server s with FSS key K and table D with table entry $d_{i,j} \in \mathbb{Z}_{2^\ell}$ can compute

$$\begin{aligned}\mathsf{FilterPred}(s, K, D) \leftarrow \Big(&\sum_{i=0}^{2^\ell - 1} \big(\mathsf{Eval}(K, i) \cdot d_{0,i}\big), \ldots, \\ &\sum_{i=0}^{2^\ell - 1} \big(\mathsf{Eval}(K, i) \cdot d_{N-1,i}\big)\Big)\end{aligned} \tag{3.2}$$

From this filter, it is easy to count the number of matching records as before: simply sum the elements in the filter. Computing the sum of values for records matching the filter requires more work, as we need to compute the dot product of the filter F and the values W, where W is a vector of database values corresponding to the feature being added (not in one-hot form), and both are secret-shared.

Combining filters using logical ANDs also requires multiplication. Given two filters F_1 and F_2, we can compute $F_1 \wedge F_2$ by multiplying F_1 and F_2 together because all the elements are in $\{0, 1\}$. The NOT operator can be easily computed locally (one pair of shares is set to $[x]_i \leftarrow 1 - [x]_i$ and the

others are set to $[x]_i \leftarrow -[x]_i$), and the OR operator can be written as a combination of ANDs and NOTs. Thus the problems of aggregating by sum and combining filters both reduce to the problem of multiplying secret-shared values.

Tool: Multiplying RSS shares. Generic MPC tools for multiplication are particularly efficient in the three-party honest-majority setting. Given shares $x_i, x_{i+1}, y_i, y_{i+1}$, we can compute $z_i = x_i y_i + x_{i+1} y_i + x_i y_{i+1}$. Then $z_1 + z_2 + z_3 = xy$, yielding a 3-out-of-3 additive sharing. We refer to this existing technique as Mult, which takes in RSS shares of operands x, y and outputs 3-out-of-3 shares of xy. To obtain a replicated secret-sharing, S_i can send a blinded share $z_i + \alpha_i$ to S_{i+1} where $(\alpha_1, \alpha_2, \alpha_3)$ is a fresh secret-sharing of zero. To generate fresh sharings of zero, we rely on a pseudorandom function (PRF) keyed during initialization. During setup, server S_i samples PRF key k_i and sends k_i to server S_{i+1}. The jth share of zero is (z_1, z_2, z_3) where $z_i = \mathsf{PRF}(k_i, j) - \mathsf{PRF}(k_{i-1}, j)$. This general approach is used in CryptGPU [479], and the technique for generating fresh sharings of zero is from Araki et al. [31]. We refer to this technique for resharing 3-out-of-3 shares to get RSS shares as $\mathsf{Reshare}$, and Mult followed by $\mathsf{Reshare}$ as $\mathsf{MultAndReshare}$, which takes RSS shares of x, y and outputs RSS shares of xy.

Supporting multiple predicates. This multiplication tool simplifies the problem of summation and combining multiple filters. The local computation costs are small, but now each server must send N values where N is the number of records for each multiplication. Generating filters using FSS with RSS produces filters that are 3-out-of-3 shared, and so before multiplying them, we use the same share conversion trick to get replicated secret shares. To reduce the cost of share conversion, we can precompute the PRF evaluations and store them until we receive a query, making the online cost of multiplication and resharing depend almost entirely on the cost of communication.

Supporting different types of aggregates. So far, we have described how to compute count (summing elements in the filter F) and sum (computing the dot product of F and the values W). We now discuss how to use these building blocks to compute other functions using previously known techniques. A well-known technique for computing the mean is to run a sum and a count query and divide locally. By storing not just the value X but also X^2, it is also straightforward to compute the variance of X as $\mathsf{Var}(X) = E[X^2] - (E[X])^2$ using the above technique to compute the mean and then squaring and subtracting locally. For standard deviation, the client locally computes the square root of variance. When using the same filter with different aggregation functions, we can reuse the filter to save computation.

3.4.3 Multiple predicates with malicious security

Up to this point, we have only considered a semihonest adversary, but to defend against a malicious adversary, the client needs to be able to check that the servers performed the computation correctly. We leverage information-theoretic MACs from MPC [126] and show how to provide integrity when chaining together predicates evaluated using FSS.

Tool: Information-theoretic MACs. To authenticate a value x in the ring $\mathbb{Z}_{2^k}$, we use the information-theoretic MACs from SPD$\mathbb{Z}_{2^k}$ [126]. The servers hold shares of x over the ring $\mathbb{Z}_{2^{k+s}}$ where s is the statistical security parameter. For some MAC key $\alpha \in \mathbb{Z}_{2^{k+s}}$ not known to the servers, the servers

also hold shares of $\alpha x \in \mathbb{Z}_{2^{k+s}}$. These MACs are additively homomorphic: $\alpha x_1 + \alpha x_2 = \alpha(x_1 + x_2)$. All computation is now performed in $\mathbb{Z}_{2^{k+s}}$ over both the value and the MAC, and the protocol aborts if the output y and the MAC tag σ do not satisfy $\alpha y = \sigma$. The probability that the attacker can forge the MAC is the probability that the attacker can guess s lower bits of α, which is $1/2^s$. We choose to use rings rather than fields even though this means that we need a larger ring because it allows us to take advantage of native instructions for addition and multiplication in our implementation.

Encoding information-theoretic MACs. We would like to apply information-theoretic MACs to Waldo such that the client chooses a value $\alpha \xleftarrow{R} \mathbb{Z}_{2^{k+s}}$ and then receives a query result of the form $(x, \sigma = \alpha x)$. Existing RSS-based 3PC works in the honest-majority setting provide malicious security for multiplications by replication combined with either cut-and-choose techniques [189, 365] or triple sacrifice [164]. Our setting is different because we are interleaving 2-party FSS with 3-party RSS. Using 2-party FSS requires us to use dishonest-majority techniques to provide malicious security, and information-theoretic MACs are a natural candidate here [56, 75, 126, 133]. Because we use MACs to authenticate the FSS evaluations, we can take advantage of authenticated input shares to provide malicious security when we combine filters via multiplication.

To authenticate multiplications, we use linear combinations of all intermediate values x_i and their MAC tags σ_i with random coefficients χ_i, i.e., $\sum \chi_i x_i$ and $\sum \chi_i \sigma_i$ where $\sigma_i = \alpha x_i$ [126]. This technique allows us to safely compute the MAC tag for xy by multiplying $\sigma = \alpha x$ directly with y because the batch check via random linear combinations ensures that xy is bound to the resulting MAC tag. The servers can compute shares of a random value χ_i using the same technique with PRFs that we use to generate random sharings of zero [31]; we refer to this as RandCoeff, which outputs RSS shares of random values and takes as input server index $\in \{1, 2, 3\}$ and the number of shares needed. With RSS shares of χ_i values, servers invoke the Mult function with RSS shares of χ_i and x_i to get 3-out-of-3 shares of $\chi_i x_i$ and similarly for $\chi_i \sigma_i$. We refer to this function as RandComb. Prior MPC work that computes random linear combinations of MACs requires the servers to perform a two-round commit-and-open protocol [126], but we can avoid this cost by taking advantage of the client. Each server simply sends the client its share of $y = \sum \chi_i x_i$ and $z = \sum \chi_i \sigma_i$ and the client assembles the shares and checks that $\alpha y = z$. Prior work [126] shows that this check catches any introduced errors with high probability; however, we only achieve $\tilde{s} = s - \log(s+1)$ parametric statistical security (see Lemma 5).

The only remaining challenge is how to encode α. A natural choice would be to keep two versions of the database where if the first version contains x, the second version contains αx. Then, the servers execute queries on both versions. This solution is secure but requires twice the amount of storage and makes revocation challenging: every client knows α, so if access is revoked, the *entire* table must be rebuilt with a new α'.

Instead, in our approach, the client chooses an α value for every query and encodes α *in the FSS key itself*. Instead of sending one (logical) keypair per predicate, the client now sends two (logical) FSS keypairs per predicate: one that evaluates to 1 on the desired input (as in the semihonest case), and another that evaluates to α on the desired input. (Note that because of the interaction with RSS, the client actually now needs to send a total of six FSS keypairs instead of three) The servers execute the query for the keys that evaluate to 1 to produce x and then for the keys that evaluate to α

to produce σ, and the client checks that $\alpha x = \sigma$. This has two key benefits: (1) we do not need to expand the size of the index for malicious security, and (2) clients do not need to share α values (no recomputation necessary when access is revoked). Boyle et al. [75] first explored the idea of supporting malicious security by encoding α values directly in FSS keys, but for the MPC setting rather than the client-server setting in databases. A key difference between these settings is that, in the MPC setting, the parties must perform a joint verification protocol, whereas in our setting, the client can verify the MAC directly from the shares produced by each server.

3.4.4 Putting it together

We present the final protocol for the client in Figure 3 and the server in Figure 4 and provide a high-level overview below.

In Figure 3, the client begins by sampling the MAC key α. For each predicate P_i, the client samples three pairs of FSS keys, which evaluate to shares of 1 for inputs satisfying P_i. In addition, for malicious security, the client samples three extra key pairs, which evaluate to shares of α when P_i is satisfied. The client sends the keys corresponding to the servers' RSS shares of the data, along with the predicate feature IDs, the ID of the feature being aggregated, and the type of aggregation (e.g. sum, count). Once the client receives responses from the servers, it reconstructs the query result x and the MAC tag σ, as well as the value $\hat{m}$ and its MAC tag $\hat{\sigma}$ that contain traces of malicious behavior via a random linear combination of the entire transcript. The client can then verify the MAC tags and output x if the checks pass.

In Figure 4, the servers receive FSS keys corresponding to the query result and the query MAC tag for each predicate and each RSS share of the data. For each predicate in the query, the servers need to compute RSS shares of the intermediate filter and its corresponding MAC tag. To do this, they evaluate each FSS key on the corresponding table share to generate a 1-out-of-6 share of the resulting filter (FilterPred from §3.4.2). Each server has RSS shares of the table and receives 2 FSS keys to evaluate each predicate, so it generates two 1-out-of-6 shares of the predicate filter, which it can then combine into a single 1-out-of-3 share. By running the Reshare protocol, the servers can convert their 1-out-of-3 shares to RSS shares. Then the servers run MultAndReshare (§3.4.2) to combine predicates together and output an RSS sharing of the accumulated filter. They can then use shares of the final accumulated filter to compute shares of the final aggregate. By performing this process for both the FSS keys for the query result and the FSS keys for the MAC tag, the servers can compute shares of both the query result and the MAC tag. To ensure that the malicious server does not manipulate the filter shares or corresponding MAC tags during multiplication, the servers must compute a random linear combination of all the messages they received using RandCoeff (§3.4.3). The servers send back shares of the final result x and the corresponding MAC tag σ, as well as the accumulated random linear combination $\hat{m}$ and its corresponding MAC tag $\hat{\sigma}$.

$\underline{\text{client.WaldoTable.Query}^{S_1,S_2,S_3}(P_1 \wedge \cdots \wedge P_n, p, \text{type})}$

1: $\alpha \xleftarrow{R} \mathbb{Z}_{2^{k+s}}, \mathcal{K}_i, \mathcal{K}'_i \leftarrow \{\}$ for $i \in \{1,2,3\}$
2: $\forall i \in \{1,\dots,n\}$, $p_i \leftarrow \text{idx}(P_i)$
3: **for** $i = 1$ to n **do**
4: **for** $j = 1$ to 3 **do**
5: $(K_1^j)_i, (K_2^j)_i \leftarrow \text{Gen}(1^\lambda, P_i, 1)$
6: $(K_1'^j)_i, (K_2'^j)_i \leftarrow \text{Gen}(1^\lambda, P_i, \alpha)$
7: **end for**
8: $\mathcal{K}_1 \leftarrow \mathcal{K}_1 \bigcup (K_1^1)_i, (K_1^2)_i$ and $\mathcal{K}'_1 \leftarrow \mathcal{K}'_1 \bigcup (K_1'^1)_i, (K_1'^2)_i$
9: $\mathcal{K}_2 \leftarrow \mathcal{K}_2 \bigcup (K_2^2)_i, (K_1^3)_i$ and $\mathcal{K}'_2 \leftarrow \mathcal{K}'_2 \bigcup (K_2'^2)_i, (K_1'^3)_i$
10: $\mathcal{K}_3 \leftarrow \mathcal{K}_3 \bigcup (K_2^3)_i, (K_2^1)_i$ and $\mathcal{K}'_3 \leftarrow \mathcal{K}'_3 \bigcup (K_2'^3)_i, (K_2'^1)_i$
11: **end for**
12: **for** $i = 1$ to 3 **do**
13: $(x_i, \sigma_i, \hat{m}_i, \hat{\sigma}_i) \leftarrow S_i.\text{WaldoTable.Query}(\mathcal{K}_i, \mathcal{K}'_i, \{p_j\}_j, p, \text{type})$
14: **end for**
15: $x \leftarrow \sum_{i=1}^3 x_i, \sigma \leftarrow \sum_{i=1}^3 \sigma_i, \hat{m} \leftarrow \sum_{i=1}^3 \hat{m}_i, \hat{\sigma} \leftarrow \sum_{i=1}^3 \hat{\sigma}_i$
16: **if** $(\alpha \cdot x \neq \sigma) \vee (\alpha \cdot \hat{m} \neq \hat{\sigma})$ **then**
17: Output $\perp$ and broadcast $\perp$ to all servers
18: **end if**
19: Output x

Figure 3: Client WaldoTable.Query algorithm. $\text{Gen}(1^\lambda, P_i, \beta)$ refers to $\text{Gen}^=(1^\lambda, a, \beta), \text{Gen}^<(1^\lambda, a, \beta)$, or $\text{Gen}^{\text{IC}}(1^\lambda, a, b, \beta)$ depending on the predicate P_i being $x = a, x < a$, or $a < x < b$, respectively. We denote P_i's feature ID as $\text{idx}(P_i)$, and $\{p_i\}_i$ denotes $\{p_1, \dots, p_n\}$.

3.5 Complex aggregates over time ranges

While our shared one-hot index can compute a useful set of aggregates using sum and count queries, not all valuable aggregates can be expressed as a combination of dot products (e.g. min, max, top-k). In many cases, the client needs to compute a complex aggregate over a time period (e.g. a doctor might want to compute the maximum glucose level of a diabetic patient in the last week). Our WaldoTree index allows the client to compute any aggregate function over a time period without server-server interaction and without revealing the time interval being queried (as prior work does [83, 84]). Because it is more efficient than WaldoTable and doesn't require interaction between the servers, WaldoTree is also valuable in cases where the query predicates are predefined.

We call our solution to this problem a *shared aggregate tree*. Our core data structure is inspired by ideas from authenticated data structures [318], Faber et al. [170], Arx [407], and Timecrypt [83]: each leaf node contains a (private) record value, and each internal node contains the (private) aggregate of its two children. Each leaf node has a public timestamp, and the n leaf nodes are ordered by time so that each internal node has a public time interval. In this way, the client can compute an

$\underline{\text{server.WaldoTable.Query}^{S_1,S_2,S_3}(\mathcal{K},\mathcal{K}',\{p_i\}_i,p,\text{type})}$

1: Parse $\mathcal{K}$ as $(K_1)_1,(K_2)_1,\ldots,(K_1)_n,(K_2)_n$
2: Parse $\mathcal{K}'$ as $(K'_1)_1,(K'_2)_1,\ldots,(K'_1)_n,(K'_2)_n$
3: $\hat{m} \leftarrow 0$, $\hat{\sigma} \leftarrow 0$, $z_1 \leftarrow 1$ if $\mathsf{ID} = 2$ and 0 otherwise, and $z_2 \leftarrow 1$ if $\mathsf{ID} = 1$ and 0 otherwise.
4: $\tilde{F} \leftarrow (z_1^N, z_2^N), \tilde{F}' \leftarrow (z_1^N, z_2^N) \in \mathbb{Z}_{2^{k+s}}^N \times \mathbb{Z}_{2^{k+s}}^N$.
5: **for** $i = 1$ to n **do**
6: $G_1 \leftarrow \mathsf{FilterPred}(\mathsf{ID},(K_1)_i,T_{p_i}.\mathsf{first})$
7: $G_2 \leftarrow \mathsf{FilterPred}(\mathsf{ID},(K_2)_i,T_{p_i}.\mathsf{second})$
8: $G'_1 \leftarrow \mathsf{FilterPred}(\mathsf{ID},(K'_1)_i,T_{p_i}.\mathsf{first})$
9: $G'_2 \leftarrow \mathsf{FilterPred}(\mathsf{ID},(K'_2)_i,T_{p_i}.\mathsf{second})$
10: $G \leftarrow G_1 + G_2$ and $G' \leftarrow G'_1 + G'_2$
11: $\tilde{H} \leftarrow \mathsf{Reshare}^{S_1,S_2,S_3}(G)$ and $\tilde{H}' \leftarrow \mathsf{Reshare}^{S_1,S_2,S_3}(G')$
12: $\tilde{F} \leftarrow \mathsf{MultAndReshare}^{S_1,S_2,S_3}(\tilde{F},\tilde{H})$
13: $\tilde{F}' \leftarrow \mathsf{MultAndReshare}^{S_1,S_2,S_3}(\tilde{F}',\tilde{H})$
14: $\{\tilde{\chi}_1,\ldots,\tilde{\chi}_{2N}\} \leftarrow \mathsf{RandCoeff}(\mathsf{ID},2N)$
15: $\hat{m} \leftarrow \hat{m} + \mathsf{RandComb}(\tilde{F}||\tilde{H},\{\tilde{\chi}_1,\ldots,\tilde{\chi}_{2N}\})$
16: $\hat{\sigma} \leftarrow \hat{\sigma} + \mathsf{RandComb}(\tilde{F}'||\tilde{H}',\{\tilde{\chi}_1,\ldots,\tilde{\chi}_{2N}\})$
17: **end for**
18: **if** type is `sum` **then**
19: $x \leftarrow \sum_{i=1}^N \mathsf{Mult}\big(\tilde{F}_i,(D_p)_i\big)$, $\sigma \leftarrow \sum_{i=1}^N \mathsf{Mult}\big(\tilde{F}'_i,(D_p)_i\big)$
20: **end if**
21: **if** type is `count` **then**
22: $x \leftarrow \sum_{i=1}^N \tilde{F}_i.\mathsf{first}$, $\sigma \leftarrow \sum_{i=1}^N \tilde{F}'_i.\mathsf{first}$
23: **end if**
24: Output $(x,\sigma,\hat{m},\hat{\sigma})$

Figure 4: Server WaldoTable.Query algorithm. We use N for the window size, n for the number of query predicates, and $\mathsf{ID} \in \{1,2,3\}$ for the server id. Variables with "tilde" denote RSS shares. The variables D,T refer to RSS shares of database and its corresponding shared one-hot index, respectively. We use $(D_p)_i$ to denote the ith record's pth feature in D (not in one-hot form), T_{p_i} refers to the shared one-hot index for p_ith feature, and $\tilde{F}_i$ denotes ith RSS share in $\tilde{F}$. We use first, second to access the respective component from an RSS share. For simplicity we assume that predicates are chained via conjunctions; disjunctions can be expressed by adding negations.

aggregate over some time interval by retrieving at most $2\log n+1$ nodes. This set of nodes represents the *covering set*, as it covers the time range that the client is querying (see Figure 5). Once the client has retrieved the nodes in the covering set, the client can locally aggregate the intermediate aggregates to compute the query result. To hide the contents of the tree from the server, we can again use RSS to share the aggregate value of each node.

Oblivious queries. The client cannot directly request the covering set from the server, as this set

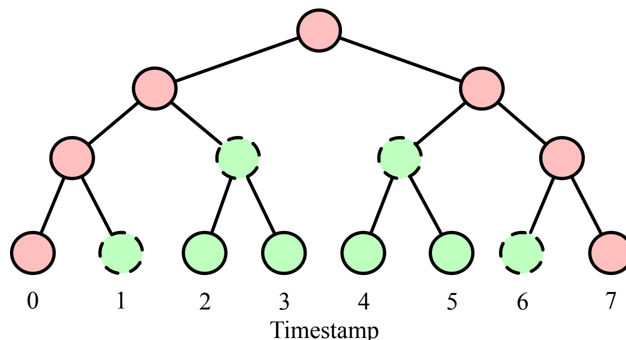


Figure 5: Node activation for the query $0 < x < 7$. Green nodes are activated, and nodes with dashed boundary are in the covering set.

reveals to the server the time period being queried. To hide this set, we can once again leverage FSS. We use the same techniques for combining FSS with RSS discussed in Section 3.4.1 and so do not describe the interplay between FSS and RSS. As a strawman, the client could send $2\log n + 1$ logical DPF keys to the servers (here we refer to each "logical" FSS key as corresponding to 3 "physical" FSS keys, one for each of the 3 pairs of servers), each of which corresponds to a node in the covering set. Note that this strawman solution requires the timestamps at leaf nodes to be at regular intervals (we fix this issue in our final solution). To prevent query leakage, clients always need to send $2\log n + 1$ keys to the servers.

We can reduce the number of FSS keys that the client needs to send to just two logical keys per server pair by instead using DCFs for the two intervals $a < x$ and $x < b$. The client generates the DCF keys for the leaf level of the tree, sends the keys to the server, and then each server evaluates its DCF keys on the timestamp for each leaf separately for both $a < x$ and $x < b$. We say that a leaf node is "activated" if its DCF evaluation is a secret-share of one, meaning that the node is within the range that the client is querying for (note that to protect the client's query, the server does not know whether or not a node is activated). Each server then *projects* the DCF evaluation for each single-sided range at the leaf nodes to the internal nodes. Projection maintains the invariant that an internal node is "activated" only if both of its children are. We can perform this projection by copying the value of the left or right child to the parent depending on the direction of the single-sided range being evaluated. For example, in Figure 5, the value of node 0 is copied to the parent of node 0 and node 1 (as this is the left side of the range), and the value of node 7 is copied to the parent of node 6 and node 7 (as this is the right side of the range). This projection operation allows us to correctly copy the DCF evaluations (secret shares of the activation status) from the leaf nodes to the internal nodes (1) without knowing which nodes are actually activated, while (2) maintaining the invariant that a node is only activated if its time interval covers part of the queried time range.

However, we only want to retrieve the nodes in the covering set (i.e. the nodes where the parent is not also activated); we can't retrieve *all* activated nodes because the number of total activated nodes is large and depends on the queried range. To filter out nodes where the parents are also activated, we compute two sums for each level ℓ: (1) the sum of all the activated nodes X_ℓ, and (2) the sum of the *children* of the activated nodes Y_ℓ. Then for each level ℓ, we return $X_\ell - Y_{\ell-1}$. This ensures that we return at most one node per level (we compute each single-sided range separately).

At the end of the protocol, the client receives $\log n$ values for each of the single-sided ranges, which the client can then use to recover the covering set for the intersection of the two ranges.

High-throughput appends. Because the leaves are ordered by time, appends are fairly straightforward. Nothing about the append is unpredictable or secret except for the value that the client is appending, and so the server simply sends the path from the root to the right-most leaf node (the tree will populate leaves moving to the right). The client uses these nodes to compute a path that incorporates the new value being appended at the internal node and aggregates and sends this path (secret-shared) to the servers. The servers use this path to update the tree to incorporate the new node. To keep the tree balanced, the servers can periodically rotate the tree (if all servers rotate the tree in the same way, the RSS sharings remain correct). Because access control is only at the table level for WaldoTree, clients can view the aggregates in the upper levels of the tree without a problem. Like WaldoTable, WaldoTree does not support privates updates to existing records (this functionality would require privately writing an arbitrary path in the tree).

One way to reduce the append overhead for resource-constrained data producers is to batch appends: the client retrieves the path along the right edge of the tree and then sends back the nodes for paths for the new values in one round trip. This greatly reduces not only round trips, but also total bandwidth because (1) only one path needs to be fetched, and (2) many of the new paths sent to the servers overlap.

Malicious security. Malicious security for our shared aggregate tree is straightforward. As in the shared one-hot index, we use information-theoretic MACs for queries by encoding them directly into the FSS key, as initially proposed by Boyle et al. [75]. For our appends, we need to ensure that the servers send the correct version of the right-most path in the tree. Here, we can rely on the fact that at most one server is malicious and each secret share is stored at two servers: if a pair of shares don't match, the client knows a server is corrupt.

Summarizing old data. We need to ensure that the tree size (and query execution time) stays bounded as the number of records increases. Our approach is inspired by Timecrypt [83]. When our tree reaches a maximum size, we rotate the tree, causing the left-most leaves to exceed the maximum depth. We then prune these leaves, leaving previously internal nodes as leaf nodes to summarize the pruned data. The client can no longer make fine-grained queries over old data, but can make coarse-grained queries that include this old data. Summarizing older data is common in modern time-series databases [262, 484].

Aggregation functions. The aggregation function does *not* need to be based on addition and can include min, max, top-k, bottom-k, histograms, and quantiles (some functions, like top-k, require storing multiple aggregate values per node). Our shared aggregate tree can also in principle support sketch algorithms [356], as well as aggregation-based encodings that allow private training of linear models [121, 279]. Notably, aggregation functions with the same output size will require the same amount of server execution time.

Final protocol. We present the final protocol for the client in Figure 6 and the server in Figure 7.

$\underline{\mathsf{client.WaldoTree.Query}^{S_1,S_2,S_3}(0,t)}$

1: $\alpha \xleftarrow{R} \mathbb{Z}_{2^{k+s}}$
2: **for** $i = 1$ to 3 **do**
3: $\quad K_1^i, K_2^i \leftarrow \mathsf{Gen}^{<}(1^\lambda, t, 1)$
4: $\quad K_1'^i, K_2'^i \leftarrow \mathsf{Gen}^{<}(1^\lambda, t, \alpha)$
5: **end for**
6: $\mathcal{K}_1 \leftarrow K_1^1, K_1^2$ and $\mathcal{K}_1' \leftarrow K_1'^1, K_1'^2$
7: $\mathcal{K}_2 \leftarrow K_2^2, K_1^3$ and $\mathcal{K}_2' \leftarrow K_2'^2, K_1'^3$
8: $\mathcal{K}_3 \leftarrow K_2^3, K_2^1$ and $\mathcal{K}_3' \leftarrow K_2'^3, K_2'^1$
9: **for** $i = 1$ to 3 **do**
10: $\quad \{x_i^{(j)}\}_{j=0}^{\log n}, \{\sigma_i^{(j)}\}_{j=0}^{\log n} \leftarrow S_i.\mathsf{WaldoTree.Query}(\mathcal{K}_i, \mathcal{K}_i')$
11: **end for**
12: **for** $j = 0$ to $\log n$ **do**
13: $\quad x^{(j)} \leftarrow \sum_{i=1}^{3} x_i^{(j)}, \sigma^{(j)} \leftarrow \sum_{i=1}^{3} \sigma_i^{(j)}$
14: $\quad$ **if** $\alpha \cdot x^{(j)} \neq \sigma^{(j)}$ **then**
15: $\quad\quad$ Output $\perp$ and broadcast $\perp$ to all servers
16: $\quad$ **end if**
17: **end for**
18: Output $x \leftarrow \mathsf{Agg}(\{x^{(0)}, \ldots, x^{(\log n)}\})$

Figure 6: Client WaldoTree.Query algorithm. n is the number of leaves in the current shared aggregate tree. Here the aggregate is computed over time range 0 to t. General case follows similarly by using double the keys and servers returning $2\log n + 1$ values. Aggregation function Agg is defined by clients during call to Init procedure; it takes in a list of values and outputs their aggregate. $\{x^i\}_{i=a}^{b}$ denotes $\{x^a, \ldots, x^b\}$.

3.6 Implementation

We implemented Waldo in ~6,200 lines of C/C++ code (excluding tests and benchmarking infrastructure). We used the libPSI [5] DPF implementation (with some minor modifications), the cryptoTools library [3] for cryptographic primitives, and gRPC for communication. We configured Waldo to aggregate values of up to size 2^{32} and set our statistical security parameter $\tilde{s} = 80$ and computational security parameter $\lambda = 128$. This allows us to use a 128-bit ring, which makes the additions and multiplications used to evaluate predicates very fast. Our implementation is available at `https://github.com/ucbrise/waldo`.

EvalAll for DCFs. We extend the state-of-the-art DCF construction from Boyle et al. [75] to perform full domain evaluations more efficiently. Waldo's protocols rely on evaluating FSS key K on each point in the domain of size 2^ℓ, referred to as $\mathsf{EvalAll}(K)$ [78]. Boyle et al. [78] proposed an EvalAll optimization for DPFs that reduces pseudorandom generator (PRG) invocations by a factor of ℓ. We apply this same insight to DCFs, providing the first EvalAll implementation for DCFs that we are aware of. This optimization improves single-threaded execution time for DCF EvalAll by 7.5×

server.WaldoTree.Query$^{S_1,S_2,S_3}(\mathcal{K},\mathcal{K}')$

1: Parse $\mathcal{K}$ as K_1, K_2 and $\mathcal{K}'$ as K'_1, K'_2
2: Initialize empty trees P, Q with n leaves each.
3: $\{x_0,\ldots,x_{\log n}\} \leftarrow \{0,\ldots,0\}$, $\{\sigma_0,\ldots,\sigma_{\log n}\} \leftarrow \{0\ldots,0\}$
4: **for** $i \in n$ **do**
5: $\quad P^{(i)}_{\log n} \leftarrow \big(\mathsf{Eval}(K_1,i), \mathsf{Eval}(K_2,i)\big)$
6: $\quad Q^{(i)}_{\log n} \leftarrow \big(\mathsf{Eval}(K'_1,i), \mathsf{Eval}(K'_2,i)\big)$
7: **end for**
8: **for** $d \in \{\log n - 1,\ldots,0\}$ **do**
9: $\quad$ **for** $i \in \{1,\ldots,2^d\}$ **do**
10: $\quad\quad P^{(i)}_d \leftarrow P^{(i)}_d.\mathsf{right}$ and $Q^{(i)}_d \leftarrow Q^{(i)}_d.\mathsf{right}$
11: $\quad$ **end for**
12: **end for**
13: **for** $d \in \{0,\ldots,\log n\}$ **do**
14: $\quad$ **for** $i \in \{1,\ldots,2^d\}$ **do**
15: $\quad\quad x_d \leftarrow x_d + P^{(i)}_d \odot \big(T.\mathsf{first}^{(i)}_d, T.\mathsf{second}^{(i)}_d\big)$
16: $\quad\quad \sigma_d \leftarrow \sigma_d + Q^{(i)}_d \odot \big(T.\mathsf{first}^{(i)}_d, T.\mathsf{second}^{(i)}_d\big)$
17: $\quad$ **end for**
18: **end for**
19: **for** $d \in \{0,\ldots,\log n - 1\}$ **do**
20: $\quad$ **for** $i \in \{1,\ldots,2^d\}$ **do**
21: $\quad\quad c \leftarrow T.\mathsf{first}^{(i)}_d.\mathsf{left} + T.\mathsf{first}^{(i)}_d.\mathsf{right}$
22: $\quad\quad \hat{c} \leftarrow T.\mathsf{second}^{(i)}_d.\mathsf{left} + T.\mathsf{second}^{(i)}_d.\mathsf{right}$
23: $\quad\quad x_{d+1} \leftarrow x_{d+1} - P^{(i)}_d \odot (c,\hat{c})$
24: $\quad\quad \sigma_{d+1} \leftarrow \sigma_{d+1} - Q^{(i)}_d \odot (c,\hat{c})$
25: $\quad$ **end for**
26: **end for**
27: Output $(\{x_0,\ldots,x_{\log n}\}, \{\sigma_0,\ldots,\sigma_{\log n}\})$

Figure 7: Server WaldoTree.Query algorithm. n is the number of leaves, $\mathsf{ID} \in \{1,2,3\}$ is the server id. T denotes the shared aggregate tree corresponding to WaldoTree object. $T^{(i)}_d$ denotes the ith node on dth level of the tree and $i.\mathsf{left}, i.\mathsf{right}$ access the value stored on the left and right child of a node i, respectively. $(a,b) \odot (c,d) = ac + bd$. Here the aggregate is computed over time range 0 to t, and so on line 10, the right child's activation status is used at parent. General case follows similarly by using twice as many keys and returning $2\log n + 1$ values.

for $\ell = 20$. Furthermore, our DCF implementation is only $60 - 70\%$ more expensive than libPSI's implementation of DPFs.

Parallelism. We parallelize most of the query computation in WaldoTable and WaldoTree across

32 threads. Waldo achieves parallelism both within and across the evaluation of predicates. We can additionally parallelize the PRF evaluations for share conversion and malicious security.

3.7 Evaluation

In evaluating Waldo, we ask the following questions:

1. How does the performance of Waldo compare to that of an ORAM and generic MPC baseline in terms of latency (Section 3.7.3, Section 3.7.4), throughput (Section 3.7.5), bandwidth (Section 3.7.6), and monetary cost (Section 3.7.7)?
2. How do the individual components of Waldo perform, and how are they affected by different parameter settings (Section 3.7.3, Section 3.7.4)?

Experimental setup. We evaluate Waldo on AWS EC2 instances. For the servers, we use r5n.16xlarge instances with 32 physical cores and 512 GB memory running on a 3.1 GHz Intel Xeon scalable processor. For the client, we use an r4.2xlarge instance with 4 physical cores and 61 GB memory running on a 2.3 GHz Intel Xeon E5-2686 v4 processor. To model the cost of transferring data between trust domains where the servers are geographically close but located in different data centers, server machines have a network bandwidth of 3 Gbps (max bandwidth to an external IP address in Google Cloud [115]) with 20ms RTT (the ping time we measured between AWS regions `us-west-1` and `us-west-2`).

3.7.1 Baselines

We now describe the two baselines that we compare Waldo to: a multidimensional tree in ORAM and our functionality executed in a generic MPC framework. We do not compare against Timecrypt [83] or Zeph [84] because they do not support multi-predicate queries and provide less security (only semihonest security, and they leak the query).

Oblivious multidimensional tree. Prior work shows how to achieve obliviousness for multidimensional queries by layering oblivious tools like private information retrieval with a multidimensional tree [199, 287]. Because we need to support both searches and updates, we store a multidimensional tree in ORAM. We use an R-tree [237] because it handles updates well (k-d trees cannot be easily rebalanced [58]). However, R-trees are poorly suited to oblivious exact query execution (prior work uses them for approximate queries). While the average-case search complexity is logarithmic in tree size, the worst-case search complexity is linear. Our baseline does not pad the number of node accesses to the worst-case (this is impractical), and so its security guarantees are weaker than Waldo's. An interesting direction for future work would be to design a multidimensional tree for time-series data compatible with ORAM.

We implement our oblivious R-tree by taking an existing R-tree implementation [237] and replacing reads and writes to nodes in local memory with ORAM accesses. We use SEAL-ORAM's implementation of PathORAM [7], which relies on MongoDB for block storage. Because Waldo uses an in-memory index, we configure MongoDB to use a memory-mapped file. A full-fledged

implementation would use techniques from oblivious data structures (ODS) [512] to encode data about the position map in the R-tree itself to minimize client local storage. For simplicity, we store the entire position map locally, as using ODS techniques would likely only add overhead (at the bare minimum, we would need to keep pointers in ORAM blocks). We use random data and random predicate values for our ORAM baseline, which adds a small amount of noise to our experiments (in contrast, Waldo and our MP-SPDZ baseline are fully oblivious, and so their performance is not affected by the data or query values). Point and range queries are executed in the same way, and we set tree dimension to the number of query predicates (for WaldoTable, the dimension is 1).

MP-SPDZ. For the MPC baseline, we implement Waldo's functionality in MP-SPDZ [6, 285], a state-of-the-art framework for general-purpose MPC. For WaldoTable queries, servers first check which records match the predicate(s), select the values for the matching records, and then aggregate. For WaldoTree queries, to give our baseline the advantage, we assume that the client encodes the query as indices of nodes in the covering set (padded to the worst-case size). Servers securely select nodes from secret shares of these indices and then aggregate by depth.

MP-SPDZ offers implementations of many different 3-party honest-majority protocols. We tested each and found that the post-processing variant of the RSS-based maliciously secure protocol from Eerikson et al. [164] was best for our setting. We used the library's mixed-mode circuit support, as well as loop decorators to parallelize computation and reduce the number of round trips. Values are aggregated modulo 2^{32} with 80-bit statistical security. All comparisons and equality checks are done only over ℓ bits for feature size of 2^{ℓ}, as in Waldo. For simplicity, we only implement the server processing and so only report server execution time (the client only has to submit a query).

3.7.2 Queries for real-world applications

We measure the performance of WaldoTable by evaluating sum queries with conjunctions of 2, 4, and 8 point and range predicates. A count query is a slightly cheaper version of the sum query (does not include a final multiplication by the values being summed), and mean, standard deviation, and variance are simply combinations of sum and count queries. For simplicity, we measure the case where all predicates are either point or range predicates, although real queries would likely contain a mix. Because Waldo is oblivious, performance does not depend on the query values or data contents. However, to make our results more concrete, we describe real-world applications where doctors need to examine relationships between measurements that correspond to two, four, and eight predicates. Throughout our evaluation, we use feature size 2^8 as it supports the applications we describe below (to the best of our knowledge as we are not medical experts).

Queries with two predicates can be used to compute the number of times an asthma patient's peak exipatory flow rate exceeded some patient-specific threshold in the last week [419]. Queries with four predicates can help identify at-risk pregnancies: doctors need to check for elevated blood pressure (`systolic < 120 AND diastolic < 80`) and sudden weight change over a time period [345]. Queries with eight predicates are useful for predicting heart failure decompensation: the success of the HeartLogic index has shown that the relationship between the first and third heart

sounds, intrathoracic impedance, respiration rate, the ratio of respiration rate to tidal volume, heart rate, and patient activity over a time period can help identify at-risk heart failure patients [444].

WaldoTree queries can be used to compute a patient's maximum or minimum heart rate (for patients with heart conditions) or maximum or minimum glucose levels (for diabetic patients) over some time period. The execution time is independent of the time interval and aggregation function (we do not measure the time for the client to aggregate nodes in the covering set, as this should be very fast).

3.7.3 Latency: WaldoTable

Understanding Waldo's performance. In Table 8, we show WaldoTable query latency for different numbers of records N and different numbers of predicates P. As expected in Waldo, after a certain point ($N = 2^{16}$), the latency grows linearly with N and P, as both computation and communication costs are $O(NP)$ (fixed ℓ). Range and point predicates perform very similarly (small performance differences are due to implementation differences). For larger values of N, the overhead of malicious security is $6-7\times$ that of semihonest security: we can use 32-bit integers rather than 128-bit integers if we only need semihonest security ($4\times$ saving), and we don't need MACs ($2\times$ saving).

Figure 9 illustrates the breakdown in query execution time for 2^{10} and 2^{20} records with different numbers of predicates (P). The majority of the overhead is due to network latency, particularly for smaller N and larger P. This is due to the fact that the number of Waldo round-trips is linear in P. For larger numbers of records, the computation and the bandwidth increase, but the number of round trips does not, and so the ratio of compute time to network time increases. Note that the PRF evaluation time (for re-sharing after multiplications and random linear combinations of MACs) could be moved into a separate preprocessing step to reduce online latency.

Comparison to baselines. In Figure 10a, we show how WaldoTable's query latency compares to that of the two baselines for different numbers of records N with 8 predicates. The latency of WaldoTable and MP-SPDZ increase at roughly the same rate, as expected, as both incur costs linear in N. Although they grow at similar rates, WaldoTable remains substantially faster than MP-SPDZ, $7.8\times$ for 2^{10} records for both point and range queries, and for 2^{20} records, $2.5\times$ for point queries and $3.9\times$ for range queries (range predicates are more expensive to evaluate in MP-SPDZ as they require two comparisons rather than one). On a slightly slower 1Gbps connection between the servers, Waldo performs $7.8\times$ better than MP-SPDZ for 2^{20} records with 8 range predicates. On the other hand, the ORAM baseline latency starts high but grows slowly, as expected due to its polylogarithmic complexity: for 2^{10} records, Waldo is approximately $37\times$ faster, and for 2^{20} records, approximately $1.4\times$ faster. Recall that our ORAM baseline does not pad to the maximum number of accesses and so has an advantage over Waldo.

Parallelism across servers. WaldoTable is parallelizable not just across cores, but also across servers without introducing new trust domains. We can split each "logical" server into n "physical" servers. The client divides its index into n equally-sized sub-indexes and delegates a sub-index to a triple of servers split across trust domains. The client can run its query on all n sub-indexes and locally aggregate the results. Because each triple processes its query chunk independently,

		WaldoTable latency (s)					
		– Point –			*– Range –*		
	$\log N$	$P=2$	$P=4$	$P=8$	$P=2$	$P=4$	$P=8$
Malicious	10	0.08	0.13	0.23	0.07	0.12	0.22
	12	0.09	0.14	0.24	0.08	0.13	0.24
	14	0.11	0.18	0.31	0.10	0.17	0.32
	16	0.22	0.40	0.79	0.21	0.39	0.77
	18	0.78	1.53	3.11	0.80	1.56	3.03
	20	3.10	6.00	11.94	3.03	6.18	11.82
Semihonest	10	0.07	0.12	0.21	0.07	0.11	0.21
	12	0.08	0.12	0.22	0.07	0.12	0.21
	14	0.09	0.14	0.23	0.08	0.13	0.23
	16	0.11	0.16	0.28	0.10	0.15	0.27
	18	0.19	0.28	0.49	0.18	0.27	0.47
	20	0.57	0.94	1.70	0.55	0.90	1.65

Table 8: WaldoTable query latency for P predicates and N records.

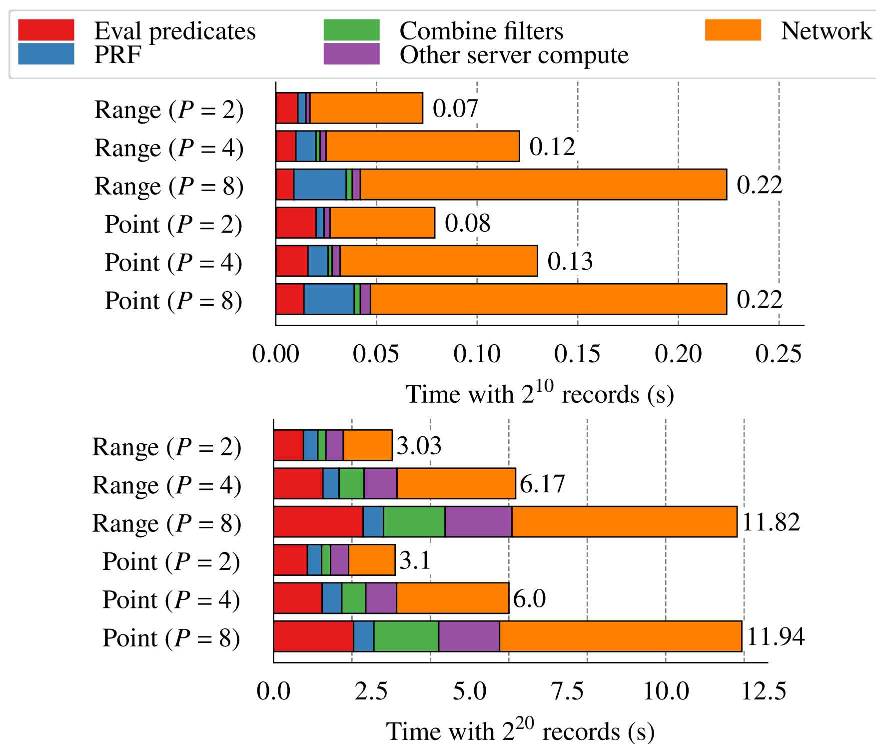


Figure 9: Breakdown of WaldoTable query latency with different numbers of predicates (P).

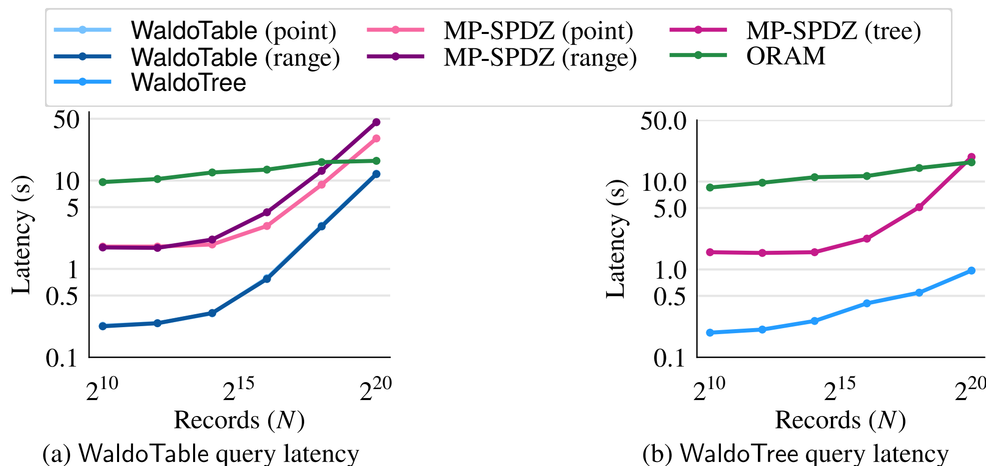


Figure 10: WaldoTable query latency is for 8 predicates, and latency for point and range predicates is almost identical.

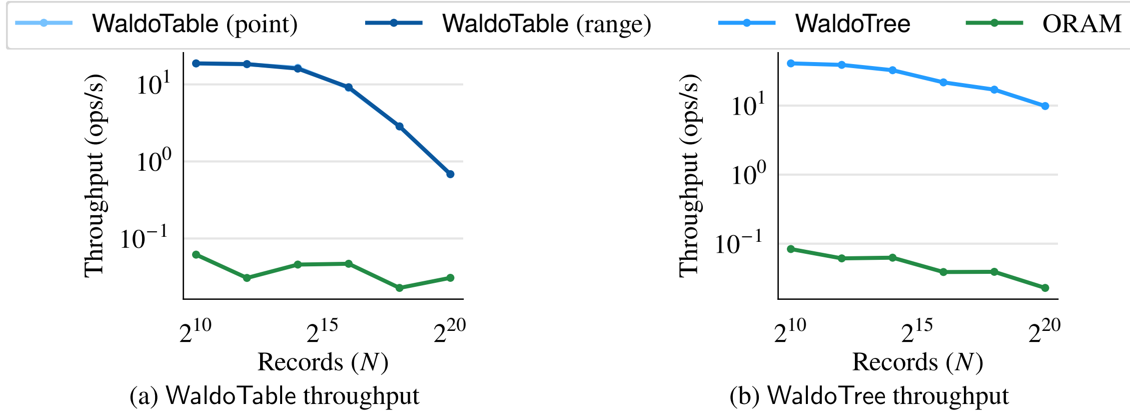
parallelism is trivial. By using 12 servers instead of 3, we estimate that 8-predicate range queries take 3.0s, whereas with 3 servers they take 11.82s.

3.7.4 Latency: WaldoTree

In Figure 10b, we show that WaldoTree queries are much faster than queries in our two baselines. WaldoTree achieves an $8-20\times$ improvement in query latency over MP-SPDZ, with the gap increasing for larger N. This gap is due to the fact that WaldoTree does not require server interaction, whereas MP-SPDZ requires a substantial amount of communication for comparisons. WaldoTree achieves a $45\times$ improvement over ORAM for 2^{10} records and a $17\times$ improvement for 2^{20} records. Again, this improvement is due to the fact that WaldoTree does not require any network overhead to execute the query whereas the client must perform many ORAM accesses to traverse the tree, resulting in many round trips.

3.7.5 Throughput

In Figure 11a and Figure 11b, we compare Waldo's throughput to that of our ORAM baseline for a 90% append, 10% query workload (time-series workloads are append-heavy, see Section 3.2.1). Waldo's throughput is orders of magnitude higher: with 2^{10} records, WaldoTable's throughput is roughly $303\times$ that of ORAM, and with 2^{20} records, roughly $22\times$ that of ORAM. For WaldoTree, this gap is even more pronounced: with 2^{10} records, the throughput difference is approximately $488\times$, and with 2^{20} records, approximately $431\times$. The throughput gap is much larger than the latency gap due to the differing cost of updates. ORAM baseline updates require multiple ORAM accesses for inserting into and potentially rebalancing the R-tree. In contrast, Waldo clients only send a secret-shared one-hot vector to the three servers.



(a) WaldoTable throughput (b) WaldoTree throughput

Figure 11: WaldoTable is configured with 8 predicates and has similar throughput for point and range predicates. ORAM throughput fluctuates due to the fact that it is not fully oblivious (doesn't make the max number of accesses) and we randomly sample the data and queries.

We don't report MP-SPDZ throughput numbers because our baseline only uses the MP-SPDZ framework for query execution and we do not implement updates. However, it is easy to compute a reasonable upper bound on MP-SPDZ's throughput. Appends in a system with MP-SPDZ would only require sending a small amount of data to each server, and so we can use 10ms as a lower bound for append latency (20ms round-trip time). From this, we can upper-bound MP-SPDZ's throughput for 90% appends and 10% searches: for WaldoTable functionality, with 2^{10} records, MP-SPDZ can achieve at most 5 ops/sec for point and range predicates (Waldo's throughput is 3.7× larger) and with 2^{20} records, 0.33 ops/sec for point predicates (Waldo's is 2× larger) and 0.22 ops/sec for range predicates (3× larger). For WaldoTree functionality, with 2^{10} records, MP-SPDZ can reach at most 5.7 ops/sec (Waldo's throughput is 7× larger), and with 2^{20} records, at most 0.5 ops/sec (Waldo's is 19× larger).

3.7.6 Communication

Server communication. In Figure 12, we compare the bandwidth between servers for Waldo and MP-SPDZ (ORAM is single-server). MP-SPDZ uses 80 – 82× more server bandwidth for 2^{10} records, and 5.8 – 8.9× more for 2^{20} records. This is due to the fact that MP-SPDZ uses communication to perform comparisons to evaluate predicates, whereas Waldo only uses compute for these evaluations. The MP-SPDZ bandwidth for 2^{10} records is inflated due to how MP-SPDZ batches comparisons; bandwidth grows linearly starting at 2^{14} records.

Client communication. In Figure 13a and Figure 13b, we show how the client bandwidth of Waldo compares to that of our ORAM baseline. Again, we do not include MP-SPDZ here because we did not implement a client, although the client bandwidth cost would likely be small. While Waldo is in the range of tens of kilobytes, our ORAM baseline is clearly in the range of megabytes (even without the ORAM baseline padding the number of accesses to match Waldo's security). In Waldo, the client only has to send 4 FSS keys to each server for every predicate for a WaldoTable query, and

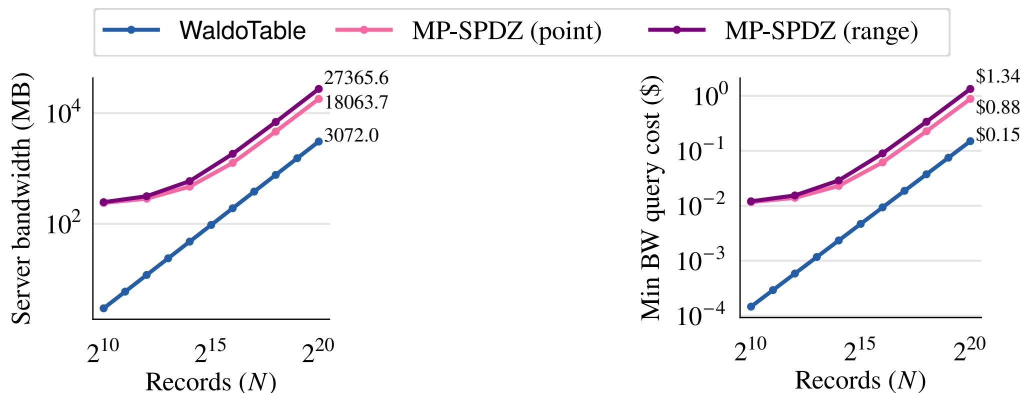


Figure 12: Overhead and cost of total bandwidth between servers for a WaldoTable query with 8 predicates. We use the minimum AWS egress bandwidth cost of \$0.05/GB [38] to compute the cost (bandwidth pricing is based on total egress bandwidth per month).

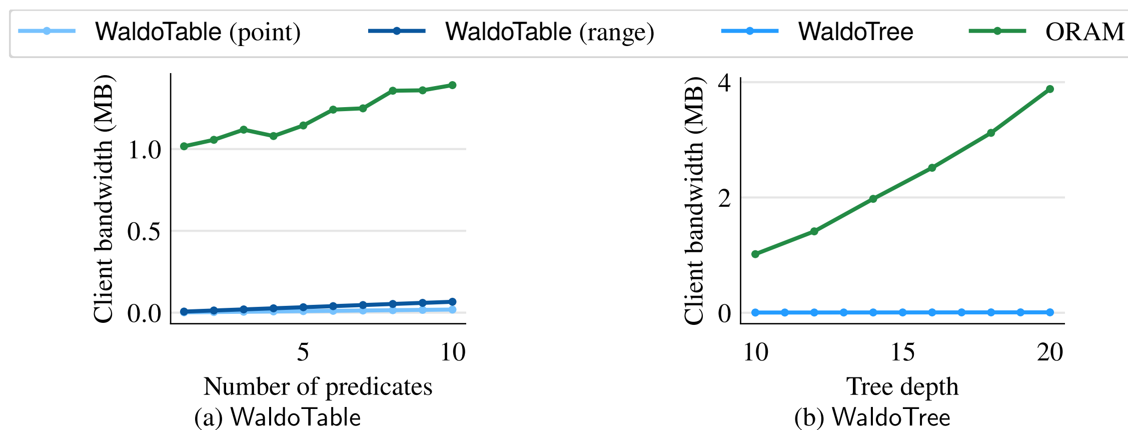


Figure 13: WaldoTable is configured with 2^{10} records and has almost identical bandwidth for point and range predicates.

8 FSS keys total for a WaldoTree query. In contrast, the ORAM baseline requires many roundtrips, performing multiple ORAM accesses for queries.

3.7.7 System cost

Server bandwidth has serious implications for system cost. Clouds typically charge steep prices for transferring data out of the cloud to incentivize customers to keep their data in the cloud. This cost is challenging for a distributed trust setting where servers routinely need to send information between clouds. For example, AWS charges \$0.09/GB if communication is less than 10TB each month and \$0.05/GB if communication is greater than 150TB per month [38]. Executing a query with 8 range predicates for 2^{20} records with MP-SPDZ costs \$1.34-\$2.41 (depending on communication cost). In contrast, a Waldo query with 8 predicates only costs \$0.15-\$0.27 (depending on communication

cost) because Waldo uses much less communication. Each server (r5n.16xlarge instance) only costs $4.77 per hour.

3.8 Limitations and future work

Small feature sizes are critical for good performance in WaldoTable. In Section 3.4.1, we discuss techniques for encoding values in a large domain using a small feature size. While Waldo supports richer functionality than prior work [83,84], it does not support all features provided by some modern plaintext time-series databases (e.g. retrieving individual records, sorts, group by, or joins [30,484]). Also, Zeph [84] supports differential privacy [163] but Waldo does not because it provides malicious security: the servers would need to add noise in a verifiably correct way and without causing MAC verification to fail. Supporting more expressive queries and providing differential privacy are valuable directions for future work. Finally, our ORAM baseline has weaker security guarantees than Waldo (recall that we do not pad to the maximum number of accesses), and so our baseline has a performance advantage. We expect that at some point for a very large number of records, the ORAM baseline will outperform Waldo because of its asymptotic complexity, but because the ORAM baseline has a performance advantage, it is not clear exactly where an ORAM solution of comparable security overtakes Waldo.

3.9 Related work

We first describe related work before the time of the original publication of this project [142] (Section 3.9.1), and then we summarize related work after publication (Section 3.9.2).

3.9.1 Related work at the time of publication

Private time series queries. Timecrypt [83] and Zeph [84] both support queries over encrypted time-series data. Unlike Waldo, they do not support filtering and leak the queried time interval. Also unlike Waldo, both systems also focus on allowing a third-party service fine-grained access to data, with Zeph considering aggregation across users. Other works explore similarity range queries over encrypted time-series data [537] and queries over encrypted and compressed time-series data [247], but these use specialized encryption schemes that have leakage that can be leveraged in statistical attacks.

FSS for private queries and secure computation. Splinter [508] uses FSS to allow users to make a variety of private queries, but unlike Waldo, the data is public and the servers are assumed to be semihonest. DORY [140] uses DPFs to enable clients to privately search for keywords in encrypted files without leaking search access patterns. DORY's queries are much simpler than Waldo's, and while DORY defends against malicious adversaries, its techniques don't easily extend to our setting because there isn't a clear way to combine its MAC tags for different predicates in the same query. Durasift [174] uses n servers to support at most $n-3$ conjunctions of arbitrary

boolean expressions of keywords using private information retrieval, implemented with a DPF. Unlike Waldo, Durasift operates in the semihonest setting, can only combine a limited number of predicates, does not support range predicates, and operates on lists of documents rather than aggregates. Floram [161] and Bunn et al. [82] use DPFs for private reading and writing in the distributed ORAM setting; these works only need to provide a block storage abstraction, whereas Waldo must evaluate predicates and combine filters. Boyle et al. first explored how FSS can be used to implement secure computation [79], with subsequent work improving on these constructions providing malicious security [75].

Encrypted databases. Encrypted databases [150, 188, 396, 398, 407, 413, 493] execute expressive queries on encrypted data, but often achieve good performance by permitting some leakage. This leakage can be exploited in statistical attacks to learn the query and database contents [94, 223, 225, 230, 269, 284, 292, 293, 305, 393, 410, 534]. SisoSPIR [266] improves performance and reduce leakages by splitting trust, but only shows how to traverse a B-tree obliviously, which is not enough to compute multi-predicate aggregates. Some encrypted databases achieve good performance by using secure hardware [168, 187, 420, 503, 536]. These solutions require additional trust assumptions due to known side-channel attacks. Another set of encrypted databases are tailored to the IoT setting, but these systems do not provide the same security and functionality as Waldo: they use encryption schemes that leak information about the database contents, reveal the query to the server, or do not support filtering [240, 451, 452, 522].

Collaborative analytics. Collaborative analytics, a related line of work, allows mutually distrusting parties to run analytics queries over their combined data. Although the setting is different than ours, like Waldo, these works split trust across parties and leverage MPC techniques to run database queries. Senate [408], Secrecy [324], SMCQL [47], and Conclave [504] support more complex analytics queries than Waldo, although they use more heavyweight tools to do so.

Secure Aggregation. Prior work also explores aggregating data across many users without a single trusted server. Some of these systems split trust across multiple servers using secret sharing [20, 121, 135, 165, 301, 455]. Like Waldo, they support aggregation on private data, but unlike Waldo, they do not support private queries.

3.9.2 Subsequent related work

We now describe some particularly relevant related work since the time of publication. Vizard uses two non-colluding servers and distributed point functions to support analytics queries while hiding access patterns [86]. TVA uses techniques from multi-party computation to support time-series analytics, like Waldo, but uses four servers to support more advanced filtering and window selection, as well as out-of-order records [173]. HEDA uses fully homomorphic encryption specifically for aggregation queries in an encrypted database [433]. HE3DB is an encrypted database that uses fully homomorphic encryption to support SQL queries, including operators like “GROUP BY” and “JOIN” [61]. CoVault uses techniques from multi-party computation with secure hardware in order to run private analytics queries on data from many users [147]. TimeClave executes time-series analytics queries inside Intel SGX and hides enclave access patterns via a new oblivious RAM

construction [41]. Solmssen also showed how to use function secret sharing for time-series queries, but his work used two semi-honest servers and supported simpler queries [461]. Orca uses function secret sharing like Waldo, but for the task of secure training and inference in conjunction with GPUs [271]. We refer the reader to Section 2.8.2 for related work on bootstrapping distributed-trust systems.

3.10 Security analysis

We use the simulation paradigm [326] of multiparty computation (MPC) to prove Waldo's security guarantees, similarly to how we proved security for DORY in Section 2.9. We consider a probabilistic polynomial time (PPT) adversary $\mathcal{A}$ who *statically* corrupts at most one of the three servers. Under $\mathcal{A}$'s control, the corrupted server is allowed to be *malicious*, meaning that it can deviate arbitrarily from the protocol specification. To keep the proof description simple, we first assume that client behaves honestly. At the end of this section, we extend to the case of malicious clients as well as collusion between client and a malicious server.

Proving security under the simulation paradigm requires defining two worlds: the real world where the actual protocol is run by honest parties, and an ideal world where an ideal functionality $\mathcal{F}$ takes inputs from the parties and directly outputs the result to the concerned party. $\mathcal{A}$ controls the behavior of the corrupted server as well as observes its view during the protocol run. To make the ideal world view of $\mathcal{A}$ indistinguishable from the real world, we need to define a simulator $\mathcal{S}$ whose job is to "simulate" messages to $\mathcal{A}$ that are similar to what the client and honest servers send to the corrupted server in the real world. However, $\mathcal{S}$ can only interact with the corrupted server and $\mathcal{F}$. If the ideal functionality $\mathcal{F}$ correctly models the leakage that our protocols claim to have and if $\mathcal{A}$ cannot distinguish between the two worlds, then we deem our protocols as being secure. To account for the case where an adversary is able to influence the operations issued by clients, and see their final result, we allow $\mathcal{A}$ to freely choose the queries issued by clients and see the result. For simplicity, we assume that predicates are only connected via conjunctions (see Section 3.4.2 for a discussion of negations, which are necessary to support disjunctions).

Ideal Functionality $\mathcal{F}$. We first define our stateful ideal functionality $\mathcal{F}$ which stores the current time series database and responds to requests in the following way:

1. $\mathsf{Init}(1^\lambda, 1^{\tilde{s}}, \mathsf{schema})$: $\mathcal{F}$ runs initialization given security parameters $\lambda, \tilde{s}$ and a schema layout parameter schema, where for $\mathsf{WaldoTable}$, $\mathsf{schema} = (N, F, 2^{\ell_1}, \ldots, 2^{\ell_F})$ with N number of records in the window, F is the number of features and ℓ_i the feature size for ith feature, and for $\mathsf{WaldoTree}$, $\mathsf{schema} = (2^\ell, \mathsf{type})$ with ℓ being the feature size and type is a user-defined aggregation function.
2. $\mathsf{WaldoTable.Append}(t, v_1, \ldots, v_F)$: $\mathcal{F}$ updates the current index to store record with timestamp t and value $v_1 \in \mathbb{Z}_{2^{\ell_1}}, \ldots, v_F \in \mathbb{Z}_{2^{\ell_F}}$.
3. $\mathsf{WaldoTree.Append}(t, v)$: $\mathcal{F}$ updates the tree index with the new entry for timestamp t and value $v \in \mathbb{Z}_{2^\ell}$.

4. $\mathsf{WaldoTable.Query}(P_1 \wedge \cdots \wedge P_n, \mathsf{feature}, \mathsf{type}) \rightarrow x$: $\mathcal{F}$ aggregates by type for feature over the boolean formula composed of predicates $P_1, \ldots, P_n$ and outputs the result x *only* to the client.
5. $\mathsf{WaldoTree.Query}(t_1, t_2) \rightarrow x$: $\mathcal{F}$ aggregates by type over time interval (t_1, t_2). It finally outputs the result x *only* to the client.

We allow $\mathcal{F}$ to leak $\mathsf{Leak}(\mathcal{F}) = (\mathsf{schema}, \mathsf{struct}_Q)$, where query structure struct_Q is defined as: (1) $(\mathsf{Init}, \lambda, \tilde{s})$ for Init, (2) (Append, t) for Append, (3) $(\mathsf{Query}, n, \mathsf{feature}, \mathsf{type}, \mathsf{kind}, \mathsf{fid})$ for WaldoTable.Query with type denoting the aggregation function (e.g. sum, count), kind denoting point or range predicates, and fid denoting a vector of feature ids corresponding to each predicate, and (4) (Query) for WaldoTree.Query.

Definition 4. Let Π be a protocol for an encrypted time-series database which takes as input requests from clients, say Q. The functionality $\mathcal{F}$ as defined above models the functionality provided by Π as a trusted party. Let $\mathcal{A}$ be an adversary who observes the view of a statically corrupted server during the protocol run and gets the final client output. Let $\mathsf{View}^{\mathrm{Real}}_{\Pi(Q)}$ denote $\mathcal{A}$'s view in the real world experiment. In the ideal world, a simulator $\mathcal{S}$ generates a simulated view $\mathsf{View}^{\mathrm{Ideal}}_{\mathcal{S},\mathsf{Leak}(\mathcal{F}(Q))}$ to $\mathcal{A}$ given only the leakage of $\mathcal{F}$. Then, $\forall$ non-uniform algorithms $\mathcal{A}$ PPT in $\lambda, \tilde{s}$, where $\lambda, \tilde{s}$ are computational and statistical security parameters, respectively, $\exists$ a PPT algorithm $\mathcal{S}$ s.t.

$$\begin{aligned} \Pr[Q \leftarrow \mathcal{A}(1^\lambda, 1^{\tilde{s}}); b \xleftarrow{R} \{0,1\}; \mathcal{A}(\mathsf{View}_b, Q) = b] \\ \leq \frac{1}{2} + \mathrm{negl}(\lambda) + \mathrm{negl}(\tilde{s}) \\ \text{where } \mathsf{View}_0 = \mathsf{View}^{\mathrm{Real}}_{\Pi(Q)}, \mathsf{View}_1 = \mathsf{View}^{\mathrm{Ideal}}_{\mathcal{S},\mathsf{Leak}(\mathcal{F}(Q))} \end{aligned}$$

Theorem 2. Using Definition 4 (Section 3.10), Waldo securely evaluates (with abort) the ideal functionality $\mathcal{F}$ (Section 3.10) when instantiated with secure distributed point and comparison functions and a pseudo-random function, all with a computational security parameter of λ.

Proof. We begin by first providing a construction for our simulator $\mathcal{S}$ for the ideal world. We work in the hybrid model [326], where invocations of sub-protocols can be replaced with that of the corresponding functionalities, as long as the sub-protocol is proven to be secure. We will operate in the $\mathcal{F}_{\mathrm{Correlated}}$-hybrid model, where we assume the existence of a secure protocol $\Pi_{\mathrm{Correlated}}$ (described in prior works [31, 479]), which realizes the ideal functionality $\mathcal{F}_{\mathrm{Correlated}}$ that generates 3-party RSS shares of the value 0 (used in Reshare) or a random value (used in RandCoeff).

Simulator Construction. Without loss of generality, let S_1 be the corrupted party. Depending on the current request Q and given access to $\mathsf{Leak}(\mathcal{F}(Q))$, $\mathcal{S}$ does the following:

1. *On receiving* $(\mathsf{Init}, \mathsf{schema}, \lambda, \tilde{s})$ *from* $\mathcal{F}$: $\mathcal{S}$ stores it locally and forwards it to $\mathcal{A}$ (who we assumed corrupts S_1).
2. *On receiving* (Append, t) *from* $\mathcal{F}$: $\mathcal{S}$ samples RSS shares of a *randomly* sampled record satisfying the structure dictated by the schema and appends them to the local database. $\mathcal{S}$ then forwards S_1's share to $\mathcal{A}$.

3. *On receiving* (Query) *from* $\mathcal{F}$: $\mathcal{S}$ samples a *random* query Q satisfying the structure dictated by schema and generates corresponding DCF keys. $\mathcal{S}$ then sends the keys for S_1 to $\mathcal{A}$. At the end, $\mathcal{S}$ receives the final output shares of S_1 from $\mathcal{A}$. If received shares aren't exactly as expected ($\mathcal{S}$ has the RSS shares and FSS keys of S_1 to check this), then output $\perp$ to $\mathcal{A}$.
4. *On receiving* $(\mathsf{Query}, n, \mathsf{feature}, \mathsf{type}, \mathsf{kind}, \mathsf{fid})$ *from* $\mathcal{F}$: $\mathcal{S}$ samples a *random* query Q satisfying the structure $(\mathsf{schema}, n, \mathsf{feature}, \mathsf{type}, \mathsf{kind}, \mathsf{fid})$. $\mathcal{S}$ generates FSS keys (DPF keys if $\mathsf{kind} = 0$ and DCF otherwise) for Q and stores them locally. It then sends the keys for S_1 to $\mathcal{A}$. Given access to $\mathcal{F}_{\text{Correlated}}$, $\mathcal{S}$ follows the rest of the steps (for multiplication) emulating the actions performed by S_2, S_3 in the real protocol. Since $\mathcal{S}$ has access to the FSS keys as well as all the database shares, it can generate the expected versions of messages from $\mathcal{A}$ that it should see. If any of the messages deviate, $\mathcal{S}$ sets an abort flag and continues. At the end, $\mathcal{S}$ receives the final output share of S_1 from $\mathcal{A}$. If the abort flag is set, then output $\perp$ to $\mathcal{A}$.

We now prove that the view generated by $\mathcal{S}$ in the ideal world is indistinguishable from the real world for a computationally (in λ, s) bounded $\mathcal{A}$ through a sequence of hybrids $\mathcal{H}_0, \ldots, \mathcal{H}_5$.

Hybrid 0. We start with the ideal world as our initial hybrid.

Hybrid 1. Simulator $\mathcal{S}$ replaces FSS keys for the random query Q with the outputs of FSS simulators $\mathcal{S}_{\text{DPF}}$ and $\mathcal{S}_{\text{DCF}}$ [75, 79]. Since $\mathcal{S}$ has access to the database shares of all the three servers and the simulated FSS keys, it can check if the messages received from $\mathcal{A}$ are exactly as *expected* or not. From the security of FSS schemes for DPF and DCF, it follows that $\mathcal{A}$'s advantage in distinguishing $\mathcal{H}_0$ from $\mathcal{H}_1$ is $\text{negl}(\lambda)$.

Hybrid 2. We let our ideal functionality $\mathcal{F}$ forward the real queries $\mathsf{WaldoTable.Query}(P_1 \wedge \cdots \wedge P_n, \mathsf{feature}, \mathsf{type})$, $\mathsf{WaldoTree.Query}(t_1, t_2)$ to $\mathcal{S}$. Recall that we allow $\mathcal{A}$ to specify these queries. $\mathcal{S}$ generates FSS keys for the real query and replaces the simulated keys from $\mathcal{H}_1$ with the real ones. From the security of aforementioned FSS schemes, it holds that $\mathcal{A}$'s advantage in distinguishing $\mathcal{H}_1$ from $\mathcal{H}_2$ remains $\text{negl}(\lambda)$.

Hybrid 3. We now allow $\mathcal{F}$ to also forward the real append queries $\mathsf{WaldoTable.Append}(t, v_1, \ldots, v_F)$ and $\mathsf{WaldoTree.Append}(t, v)$ to $\mathcal{S}$. $\mathcal{S}$ generates and distributes RSS shares of the real incoming record and uses that instead of RSS shares for randomly sampled record. Although $\mathcal{A}$ specifies the append query and knows the incoming record's value, it only sees RSS shares of at most one server. These limited shares are independent of the actual record values. Therefore, $\mathcal{A}$'s view in $\mathcal{H}_2$ and $\mathcal{H}_3$ is identical.

Hybrid 4. This hybrid corresponds to the real world with calls to $\mathcal{F}_{\text{Correlated}}$. The only difference in $\mathcal{H}_4$ over $\mathcal{H}_3$ is the reliance on MACs to detect malicious behavior. As mentioned earlier, we allow the adversary to observe the client's final output for every query that it issues. $\mathcal{A}$'s view is distinguishable from $\mathcal{H}_3$ when the client's output is erroneous and it still doesn't abort. In Lemma 5, we prove that the probability of $\mathcal{A}$ cheating and not getting caught during MAC verification step is $\text{negl}(\tilde{s})$, for statistical security parameter $\tilde{s}$. Thus the distinguishing advantage of $\mathcal{A}$ between $\mathcal{H}_3$ and $\mathcal{H}_4$ is $\text{negl}(\tilde{s})$.

Hybrid 5. In our final hybrid, we replace the calls to $\mathcal{F}_{\text{Correlated}}$ with the PRF calls to realize $\Pi_{\text{Correlated}}$ for $\mathsf{Reshare}$ and $\mathsf{RandCoeff}$. From the security of $\Pi_{\text{Correlated}}$ [31] (relies on PRF security),

we have that $\mathcal{A}$ cannot tell $\mathcal{H}_4$ and $\mathcal{H}_5$ apart with probability any better than negl(λ) over a random guess.

This concludes our proof that given views of either the real or ideal world, $\mathcal{A}$ cannot correctly guess which view it is, except with probability $\leq \frac{1}{2} + \text{negl}(\tilde{s}) + \text{negl}(\lambda)$. □

Lemma 5. *In our proposed protocols, a cheating server is caught by the client during MAC verification step with probability* $1 - \text{negl}(\tilde{s})$*, where* $\tilde{s} = s - \log(s+1)$ *is the statistical security parameter.*

Proof. Any locally introduced error cascades down in the subsequent computation across all servers due to exchange of malformed messages from the adversary. This can be modeled as an equivalent additive error of adversary's choosing in the protocol messages it sends. In particular, the only time servers communicate with each other is during Reshare, and any error introduced by $\mathcal{A}$ at this point, translates to the same error in the RSS shares that are next established. We formalize this using a non-uniform PPT algorithm $(e_{1,j}, \ldots, e_{N,j}) \leftarrow \text{ChooseError}(j, t)$, where j denotes the communication round in the protocol and t is the current state available to $\mathcal{A}$. t includes all the information that $\mathcal{A}$ has about the distribution of secrets, initial protocol state as well as the transcript so far.

Our MAC check is: $\alpha \cdot \sum_{i,j} \chi_{i,j} x_{i,j} \stackrel{?}{=} \sum_{i,j} \chi_{i,j} \sigma_{i,j}$, where $\chi_{i,j}$ are random coefficients sampled from $\mathbb{Z}_{2^{k+s}}$ and $x_{i,j}$ (resp. $\sigma_{i,j}$) are all messages (resp. their MACs) whose secret shares are exchanged during jth communication round in the protocol. No server knows the values $\chi_{i,j}$. Given that $\exists i, j$ such that $e_{i,j} \neq 0$ (otherwise, there is no tampering), then passing the MAC check requires: $\alpha \cdot (\sum_{i,j} \chi_{i,j}(x_{i,j} + e_{i,j})) - \sum_{i,j} \chi_{i,j} \sigma_{i,j} + \Delta = 0$, where Δ is the corrective error $\mathcal{A}$ needs for the check to pass. It was shown in [126] that given α is sampled uniformly from $\mathbb{Z}_{2^{k+s}}$, the probability of passing the check $2^{-s+\log(s+1)}$. □

Client and Server Collusion. In the case of an adversarial client, the only goal of collusion can be to learn information about database entries that aren't accessible to the client based on the set access control policies. If the client tries to access databases outside its clearance, the two honest servers will immediately abort the protocol before even sending their first protocol message. Given that shares of the database with the remaining server (colluding with the client) are independent of the actual contents of the database, and neither the client nor the server receive any message from the two honest servers except an abort, it is straightforward to see that the case of a malicious client is securely dealt with. Note that the case of a malicious non-colluding client is subsumed in this preceding argument.

3.11 Conclusion

We presented Waldo, a private time-series database that operates in the malicious three-party honest-majority setting. While prior work [83, 84] only supports time-based filtering and reveals the queried time intervals, Waldo enables multi-predicate filtering while hiding the filter values and search access patterns. Waldo contributes new techniques that build on top of function secret

sharing to enable Waldo to evaluate predicates non-interactively. Our MPC baseline uses $9-82\times$ more bandwidth between servers than Waldo (for different numbers of records), and our ORAM baseline uses $20-152\times$ more bandwidth between the client and server(s) than Waldo (for different numbers of predicates).

Chapter 4

Snoopy: An oblivious, scalable object store

4.1 Introduction

Organizations increasingly outsource sensitive data to the cloud for better convenience, cost-efficiency and availability [180, 300, 450]. Encryption cannot fully protect this data: how the user accesses data (the "access pattern") can leak sensitive information to the cloud [94, 150, 225, 269, 284, 291]. For example, the way in 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 [63, 97, 128, 206, 392, 432, 442, 471–473, 517]. 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 [361, 432, 473], scale to large data sizes via data parallelism [334], and improve request throughput [128, 442, 517]. 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 [94, 225, 269, 284, 291]. 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 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 [432, 472, 473].

Thus high-throughput oblivious storage systems are all built on hierarchical [472] or tree-like [432,473] structures and either require a centralized coordination point (e.g., a query log [97,517] or trusted proxy [63, 128, 442, 471]) or inter-client communication [76]. 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?*

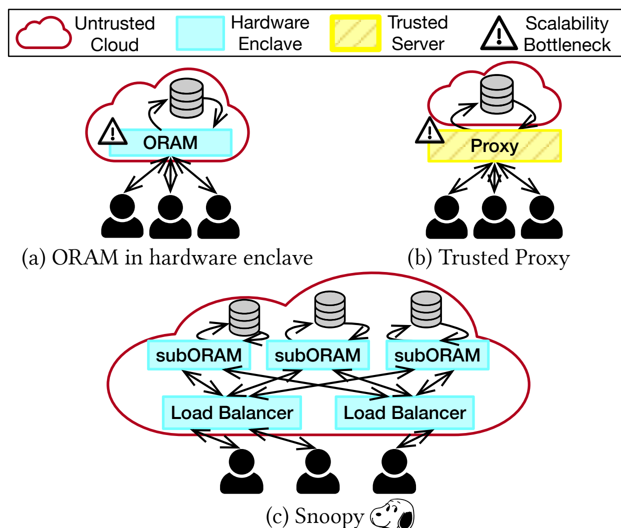
Removing the scalability bottleneck. In this work, we propose Snoopy (scalable nodes for oblivious object repository), a high-throughput oblivious storage system that scales similarly to a plaintext storage system. While our system is secure for any workload, we design it for high-throughput workloads. 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 [21, 361, 445]. 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 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 [81, 239, 311, 315, 363, 448, 498, 520]. 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 [21, 361, 445] 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 [128] 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 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 the load balancer and subORAM to (1) leverage efficient oblivious algorithms to defend against



(a) ORAM in hardware enclave (b) Trusted Proxy

(c) Snoopy

Figure 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.

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 (Section 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 [21, 361, 445] 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 [21, 361, 445], particularly in the hardware enclave setting. We leverage a specialized data structure to process batches efficiently

and obliviously in a single linear scan (Section 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 (Section 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 [432, 473]. In the low-throughput regime, although Snoopy is still secure, its latency will likely be higher than that of non-batching systems like ConcurORAM [97], TaoStore [442], or PrivateFS [517]. For large data sizes and low request volume, a system like Shroud [334] 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 [74, 124, 312]; 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 [81, 239, 363, 448], branch prediction [315], paging-based attacks [498, 520], and memory bus snooping [311]. By using oblivious algorithms, Snoopy defends against this class of attacks. Snoopy does not defend against enclave integrity attacks such as rollback [401] and transient execution attacks [109, 424, 447, 495, 496, 499, 500], 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 [227, 328]).

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 [29]; 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 Section 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 [401]. We discuss how Snoopy can integrate with existing rollback-attack solutions in Section 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 Section 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 [111, 372, 480] or denial-of-service attacks due to memory corruptions [228, 270]. We additionally consider transient execution attacks [109, 424, 447, 495, 496, 499, 500] 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 [74], Keystone [312], or Sanctum [124], 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 the 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 Section 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 Section 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 [251]: if one operation happens after another in real time, then the second will always see the effects of the first (see Section 4.4.3 for how we achieve this). We include a linearizability proof in Section 4.13.

4.2.1 Formalizing security

We formalize our system and prove its security in Section 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 [206]). 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 18.

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 18.

Using these experiments, we present our security definition:

Definition 6. 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

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

where λ is the security parameter, the real and ideal experiments are defined informally above and formally in Figure 18, 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 10. We prove the security of Snoopy using any subORAM scheme that is secure under this modified definition.

Theorem 7. *Given a two-tiered oblivious hash table [102], an oblivious compare-and-set operator, and an oblivious compaction algorithm, the subORAM scheme described in Section 4.5 and formally defined in Figure 19 is secure according to Definition 10.*

Theorem 8. *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 10), Snoopy, as described in Section 4.4 and formally defined in Figure 21, is secure according to Definition 6.*

All of the tools we use in the above theorems can be built from standard cryptographic assumptions. We prove both theorems in Section 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 [432, 473]. 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 [63, 471]. 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 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).

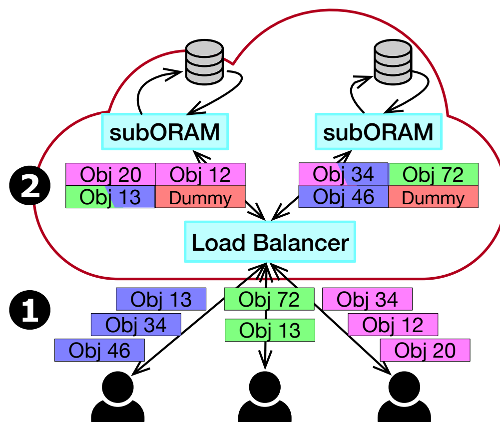


Figure 2: Secure distribution of requests in Snoopy. ❶ The load balancer receives requests from clients. ❷ At the end of the epoch, the load balancer generates a batch of requests for each subORAM, padding with dummy requests as necessary.

4.3.1 System architecture

Snoopy's system architecture (Figure 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 [36].

The role of the *load balancer* is to partition requests received during the last epoch into equally sized batches while providing security and efficiency (Section 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 (Section 4.5). Snoopy can be deployed using any oblivious storage scheme for hardware enclaves [21, 361, 445] 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 [21], cloud electronic health records, and Signal's private contact discovery [346]. Privacy-preserving cryptocurrency light clients can also benefit from Snoopy. These allow lightweight clients to query full nodes for relevant transactions [349]. 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 [339, 459], oblivious storage systems like Obladi [128] 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 [9, 355] 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 [211].

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 (Section 4.4.1), designing oblivious algorithms to efficiently generate batches while protecting the contents of the requests (Section 4.4.2), and scaling the load balancer across machines (Section 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 Section 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) protect against dropping requests, 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 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) [128]. 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

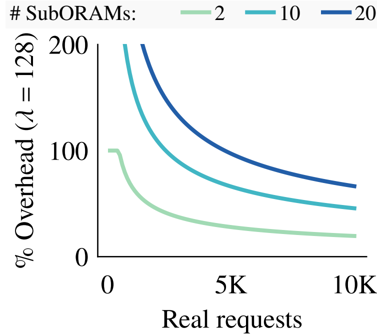


Figure 3: Dummy request overhead. A 50% overhead means for every two real requests there is one dummy request.

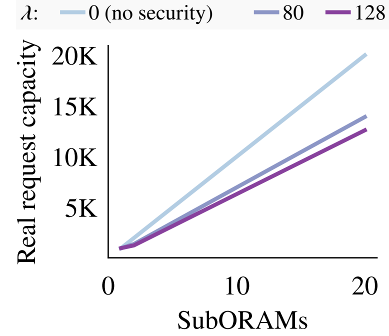


Figure 4: The total real request capacity of our system for an epoch, assuming ≤1K requests per subORAM per epoch.

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 [422]: 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 3 and Figure 4. Figure 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 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 Section 4.11:

Theorem 9. *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 [120]. Then for the following function $f(R,S)$ that outputs a batch size, the probability that a request is dropped is negligible in λ:*

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

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., $W(we^w) = w$ [120]. 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 9 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 (Section 4.4.2). Theorem 9 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) [59, 362, 422].

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 and matching subORAM responses to client requests.

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.

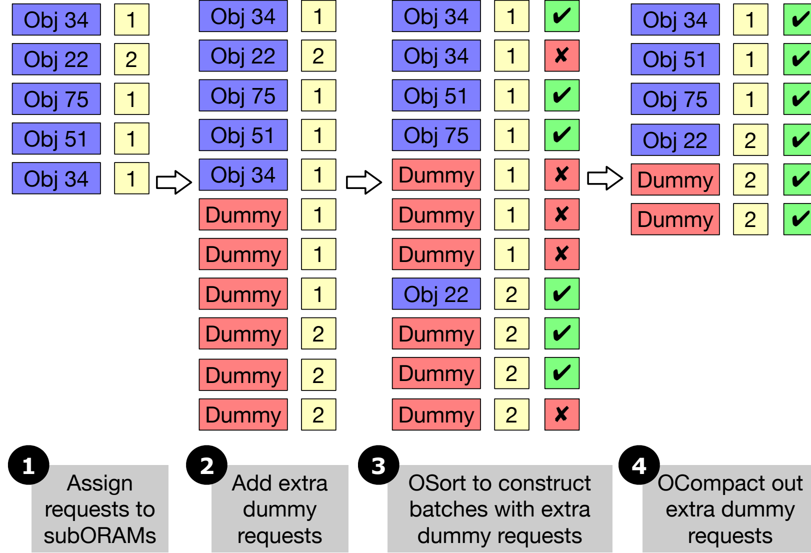


Figure 5: Generating batches of requests at the load balancer.

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 [46]. 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 [210]. 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 generate batches in three steps, which we show in Figure 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

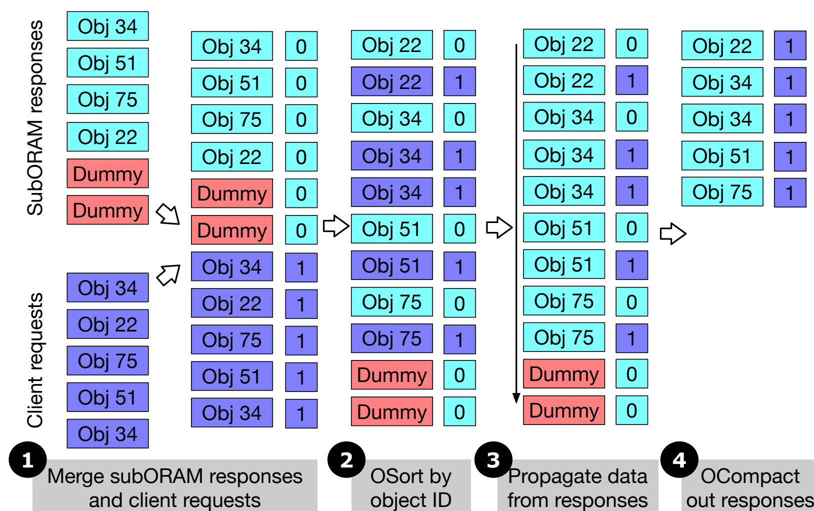


Figure 6: Mapping subORAM responses to client requests at the load balancer.

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.

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 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 [63, 128, 442, 471]. 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 (Section 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 [140, 161]. 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 [346]. 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 [102, 302].

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 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 [102]. 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 ~10× 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,

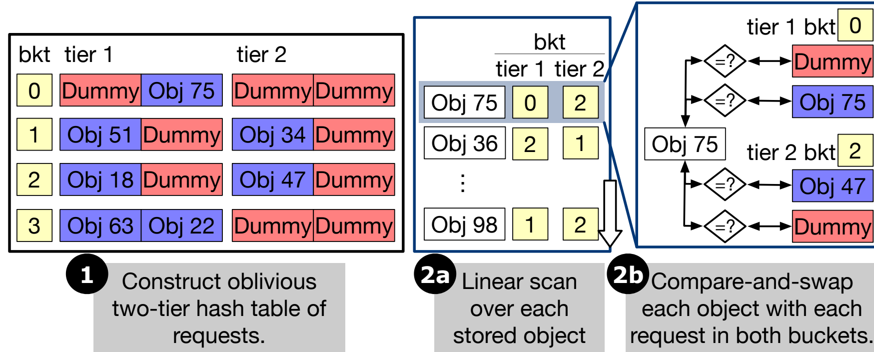


Figure 7: Processing a batch of requests at a subORAM.

and the security analysis [102].

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 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 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 [386] and Intel SGX v2.13. We use gRPC v1.35 for communication and OpenSSL for cryptographic operations. Our bitonic sort [46] and oblivious compaction [210] 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. Our implementation is open-source [8].

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 [389]. 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 [406, 409].

	Redis [430]	Obladi [128]	Oblix [361]	**Snoopy**
Oblivious	✗	✓	✓	✓
No trusted proxy	✓	✗	✓	✓
High throughput	✓	✓	✗	✓
Throughput scales with machines	✓	✗	✗	✓

Table 8: Comparison of baselines based on security guarantees (oblivious), setup (no trusted proxy), and performance properties (high throughput and throughput scales).

4.8 Evaluation

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

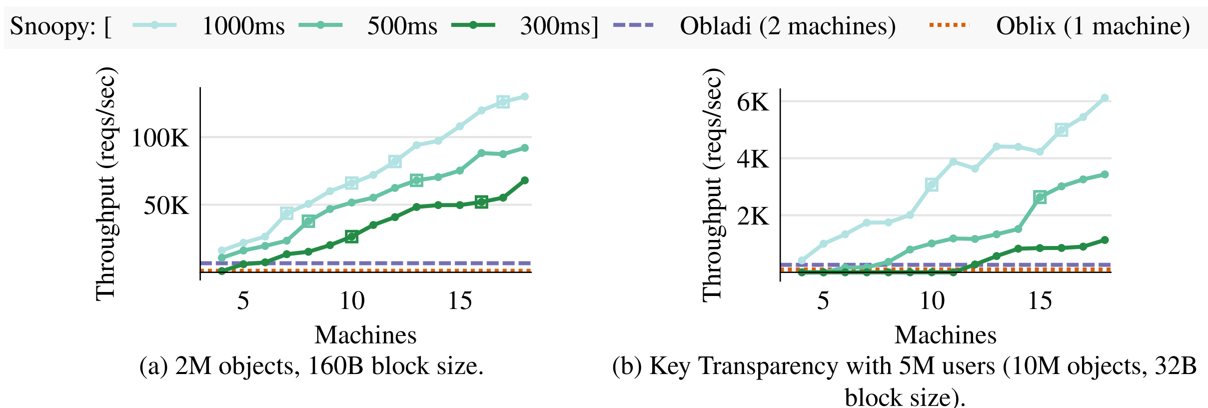
1. How does Snoopy's throughput scale with more compute, and how does it compare to existing systems? (Section 4.8.2)
2. How does adding compute resources help Snoopy reduce latency and scale to larger data sizes? (Section 4.8.3)
3. How do Snoopy's individual components perform? (Section 4.8.4)
4. Given performance and monetary constraints, what is the optimal way to allocate resources in Snoopy? (Section 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 [430] on `D4d_v4` instances, Obladi [128] 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 [361]).

4.8.1 Baselines

We compare Snoopy to three state-of-the-art baselines: Obladi [128] is a batched, high-throughput oblivious storage system, Oblix [361] efficiently leverages enclaves for oblivious storage, and Redis [430] is a widely used plaintext key-value store. Each baseline provides a different set of security guarantees and performance properties (Table 8).

Obladi. Obladi [128] uses batching and parallelizes RingORAM [432] 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



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

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

Figure 9: 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.

primary 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 [361] 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 [361]).

Redis. To measure the overhead of security (obliviousness), we compare Snoopy to an insecure baseline Redis [430], 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 [357], 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 9a 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 (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

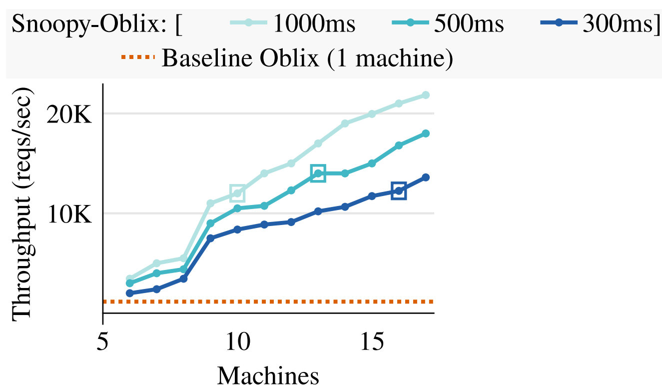


Figure 10: Throughput of Snoopy using Oblix [361] as a subORAM (2M objects, 160B block size). We measure throughput with different maximum average latencies.

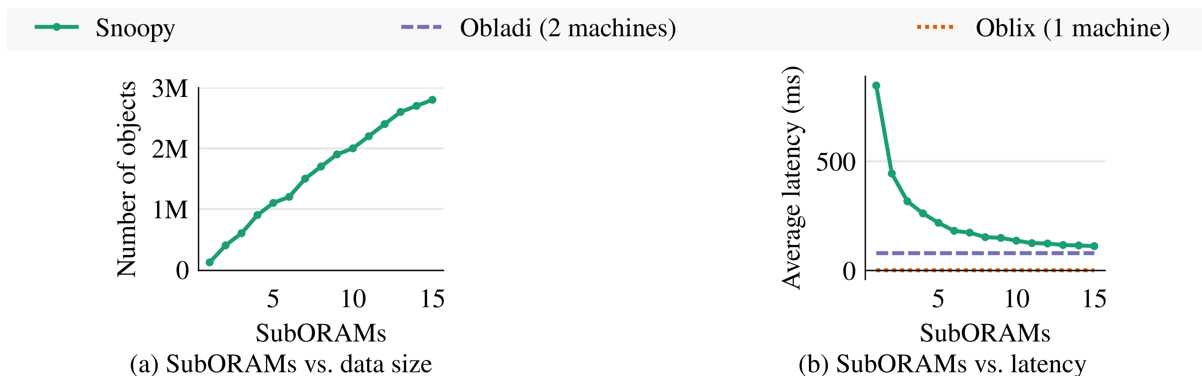
for 1s latency. Relaxing the latency requirement improves throughput because we can group requests into larger batches, reducing the overhead of dummy requests.

We generate Figure 9a 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 Equation (4.2) would require a subORAM to process a batch in ≤ 120ms). Both the load balancer and subORAM are memory-bound, as the EPC size is limited and enclave paging costs are high (Section 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 ≥5 machines and Obladi with ≥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 (Section 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× 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 9b shows throughput for parameter settings that support key transparency (KT) [9, 355] 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) [355]. 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 9b 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



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

Figure 11: (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.

reqs/sec with 500ms latency, and 6.1K reqs/sec with 1s latency. Note that the throughput in Figure 9b is much lower than Figure 9a because each KT operation requires 24 ORAM accesses.

Oblix as a subORAM. In Figure 10, we run Oblix [361] as a subORAM instead of Snoopy's throughput-optimized subORAM (Section 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 9a) improves throughput by 4.85× with 17 machines and 500ms latency.

4.8.3 Scaling for latency and data size

While Snoopy is designed specifically for throughput scaling (Section 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 11. Figure 11a 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 11b 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 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.

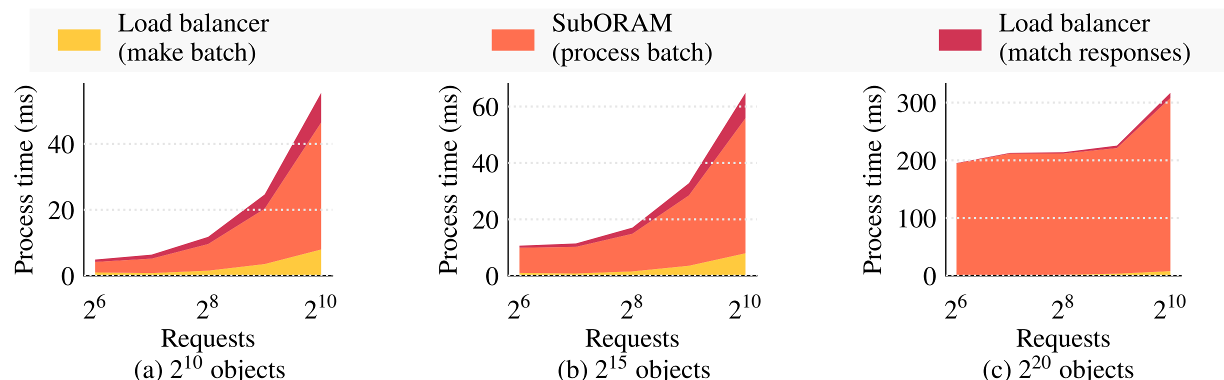


Figure 12: Breakdown of time to process one batch for different data sizes (one load balancer and one subORAM).

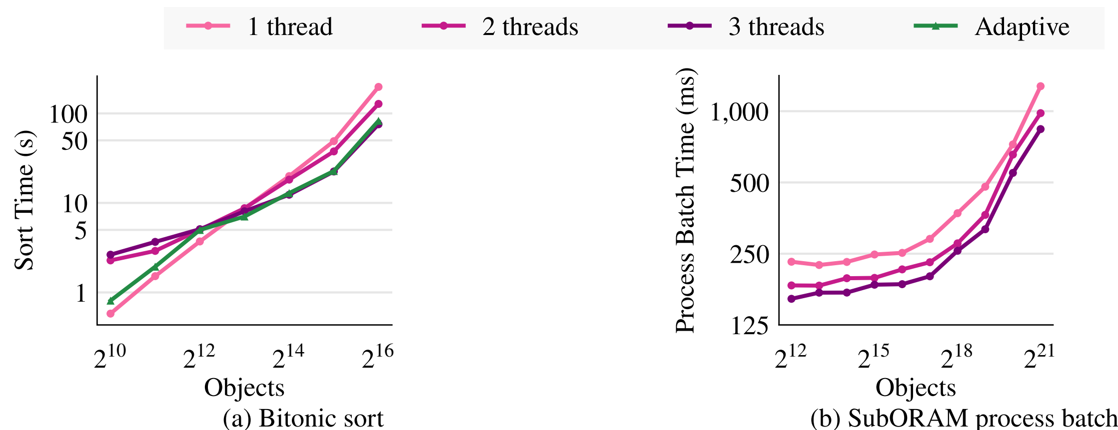


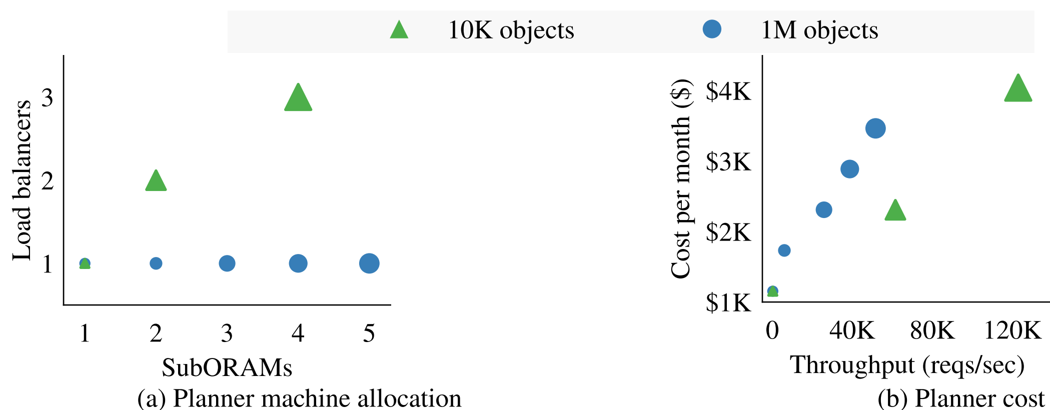
Figure 13: (a) Parallelizing bitonic sort across multiple threads. (b) Parallelizing batch processing at the subORAM across multiple enclave threads (batch size 4K requests).

4.8.4 Microbenchmarks

Breakdown of batch processing time. Figure 12 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 13a, we show how parallelizing bitonic sort across threads reduces latency, especially for larger data sizes. For smaller data sizes, the coordination overhead 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 13b, 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



(a) Planner machine allocation

(b) Planner cost

Figure 14: 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.

the linear scan over the all objects in the subORAM (Section 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 14, we use our planner to find the optimal resource allocation for different performance requirements. Figure 14a 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 14b, 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.

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 [401]. Prior work has explored how to defend against such attacks [80, 348]. Fault tolerance and rollback prevention are not the focus of this chapter, 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 [348] 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 [263]. 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 (Section 4.8.5).

Private Information Retrieval (PIR). Snoopy's techniques can also be applied to the problem of private information retrieval (PIR) [112, 113]. 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 [248, 267]. Existing systems develop relevant batching [29, 236] and preprocessing [290] techniques.

4.10 Related work

We first describe related work at the time of publication of the original paper [139] (Section 4.10.1), and then we include subsequent related work (Section 4.10.2).

4.10.1 Related work at the time of publication

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 [361], ZeroTrace [445], Obliviate [21], Pyramid ORAM [123], and POSUP [253] do not support concurrency. Snoopy, in contrast, optimizes for throughput and leverages batching for security and scalability. ObliDB [168] supports SQL queries by integrating PathORAM with hardware enclaves, but uses an oblivious memory pool unavailable in Intel SGX.

GhostRider [328] and Tiny ORAM [181] use FPGA prototypes designed specifically for ORAM. While no general-purpose, enclave-based ORAM supports request parallelism, MOSE [252] and Shroud [334] leverage data parallelism to improve the latency of a single request on large datasets. MOSE runs CircuitORAM [103] 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 [517] and ConcurORAM [97] 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 [442] and Obladi [128] 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 [490], 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. [76], OPRAMs have been explored in subsequent work [101–103, 108] and expanded to other models of parallelism [425].

Scaling out ORAMs. Several ORAMs support distributing compute and/or storage across multiple servers. Oblivistore [471] distributes partitions of SSS-ORAM [472] 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 [63] 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 [442].

Pancake [222] 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 [59, 362, 422]. Berenbrink et al. [59] assume a sufficiently large number of bins to derive an overflow probability n^{-c} for n bins and some constant c (Onodera and Shibuya [385] apply this bound in the ORAM setting). Raab and Steger [422] use the first and second moment method to derive a bound where overflow probability depends on bucket load. Ramakrishna's [426] 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. [434] provide an alternate formulation that can be evaluated by a symbolic computation tool, but we were unable to efficiently evaluate it with SymPy.

4.10.2 Subsequent related work

We will now describe some related work since the time of publication of the original paper [139].

Recent work has explored how to improve the properties of oblivious storage systems. Shortstack and QuORAM examine the problem of providing obliviousness and availability even when there are failures [343, 505]. We discuss fault tolerance in Snoopy in Section 4.9, although it is not a core part of our design. BULKOR makes bulk loading in PathORAM more efficient, and it supports deployment on enclaves via doubly oblivious primitives [323]. Waffle [344] builds on Pancake [222] to provide obliviousness against an attacker that can chose the sequence of requests, as opposed to an adversary that can see accesses but cannot affect the requests (as in Pancake). EnigMap constructs an oblivious map with an enclave and optimizes for the case where there is limited external memory [487]. EnigMap targets the application of Signal private contact discovery, as well as key transparency and databases.

Researchers have also proposed enclave-based oblivious systems that provide functionality different from that of Snoopy. GraphOS allows a client to outsource a graph to a storage server and obliviously run graph queries [98]. Boomerang uses enclaves for metadata-hiding messaging, and it leverages our load balancer algorithm for adding dummy requests [272].

Recent work has also improved core oblivious building blocks. Ngai et al. develop oblivious sorting and shuffling algorithms that scale across enclaves [378] Sasy, Johnson, and Goldberg introduce oblivious sorting and shuffling algorithms in the offline/online model [446]. Gu et al. designed a new oblivious sorting algorithm for enclaves that minimizes compute and page swaps [229].

4.11 Parameter analysis

Theorem 9. *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 [120]. Then for the following function $f(R,S)$ that outputs a batch size, the probability that a request is dropped is negligible in λ:*

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

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 $\mu = \mathbb{E}[X]$, 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 [120].

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 [472, 473]. 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:

- $\mathsf{E_P} \leftarrow \mathsf{Load}(\mathsf{P})$: The load function takes a program P and produces an enclave DAG $\mathsf{E_P}$ 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.
- $(\mathsf{out}, \gamma) \leftarrow \mathsf{Execute}(\mathsf{E_P}, \mathsf{in})$: 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. $\mathsf{Execute}$ 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 $\mathsf{Execute}$ 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 [445]. However, ZeroTrace only considers a single enclave whereas we consider a DAG of enclaves (similar to Opaque [536]). 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 Section 4.12.7.

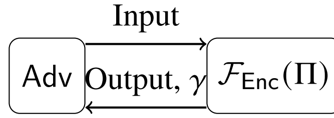


Figure 15: Real experiment for protocol Π running inside the enclave ideal functionality $\mathcal{F}_{\mathsf{Enc}}$ where γ is the trace.

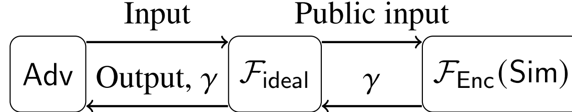


Figure 16: Ideal experiment where adversary interacts with the ideal functionality (computes the output for the given input) and the ideal functionality sends the public information to a simulator program running inside the enclave ideal functionality ($\mathcal{F}_{\mathsf{Enc}}$) to generate the trace γ.

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 [536]. 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 (Section 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 Section 4.9).

4.12.3 Oblivious storage definitions

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

- OStoreInitialize(1^λ, $\mathbf{O}$), takes as input a security parameter λ and an object store $\mathbf{O}$ and runs initialization.
- $\mathsf{V} \leftarrow$ OStoreBatchAccess(R), a protocol where the client's input is a batch 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 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 a real experiment and an ideal experiment, similarly to DORY (Section 2.9) and Waldo (Section 3.10). In the real experiment (Figure 15), the adversary interacts with an enclave DAG loaded with the real protocol, and in the ideal experiment (Figure 16), 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

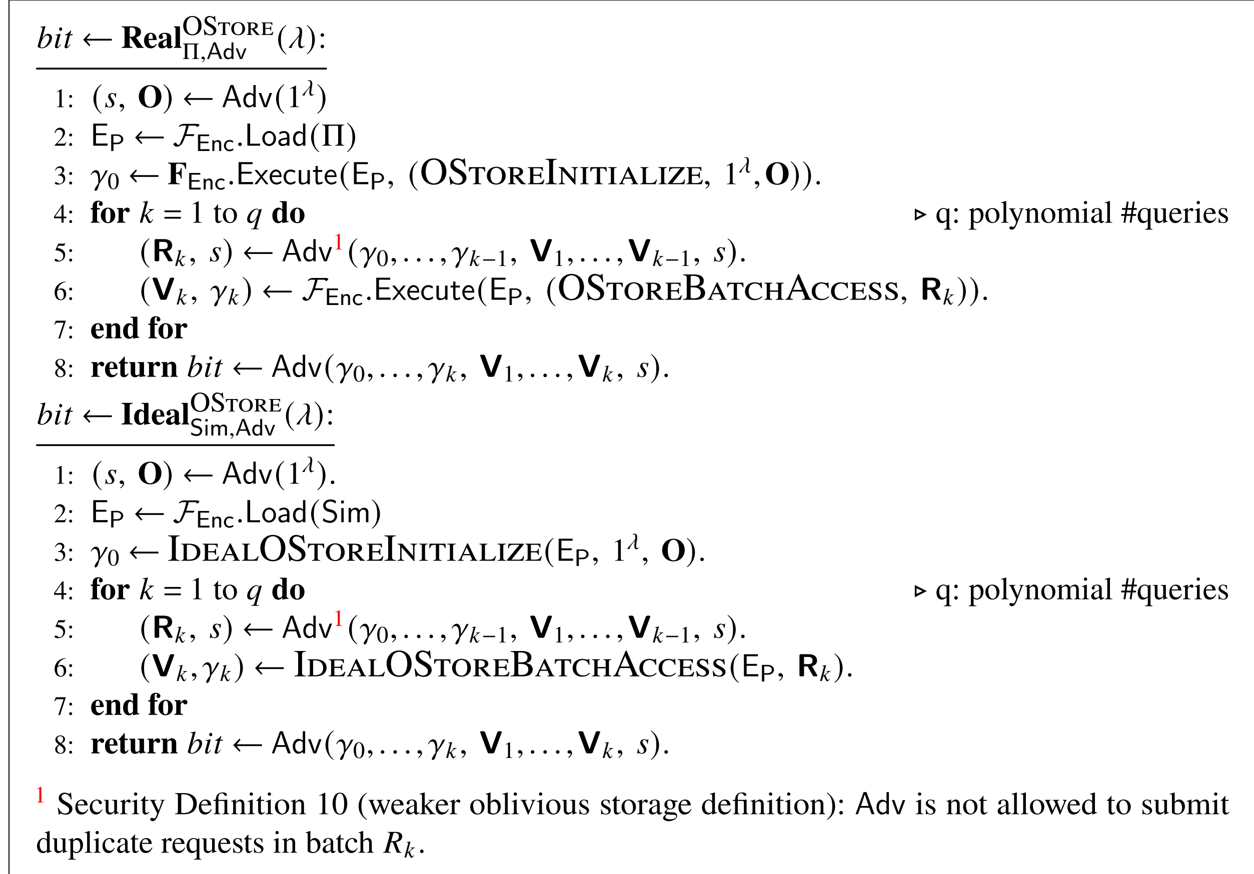


Figure 17: Real and ideal experiments for an oblivious storage scheme.

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 [472, 473] (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 is secure, and then we prove that our Snoopy construction is secure when built on top of a secure

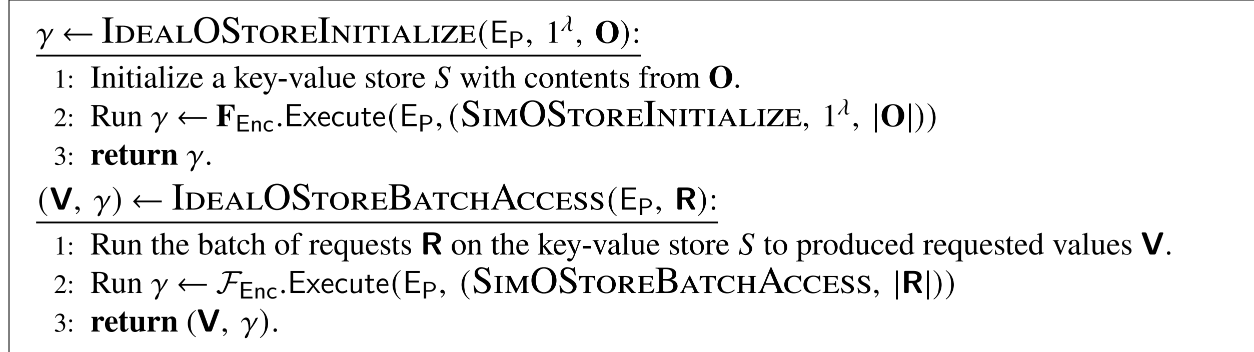


Figure 18: Ideal functionalities.

subORAM. To do this, we need a slightly different notion of subORAMs. In particular, our SubORAM construction cannot be proven secure with Definition 6, 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 10. (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 18 (see note 1), and the randomness is taken over the random bits used by the algorithms of Π, Sim, and Adv.

4.12.4 Oblivious building blocks

We use the following oblivious building blocks:

- $\mathsf{OCmpSwap}(b, x, y)$: If $b = 1$, swap x and y.
- $\mathsf{OCmpSet}(b, x, y)$: If $b = 1$, set $x \leftarrow y$.
- $L' \leftarrow \mathsf{OSort}(L, f)$: Obliviously sorts the list L by some ordering function f, outputs sorted list L'.
- $L' \leftarrow \mathsf{OCompact}(L, B)$: 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 [46] and oblivious compaction using Goodrich's algorithm [210]. We set the client's memory to be constant size in both. OCmpSwap and OCmpSet are standard oblivious building blocks, as described in Oblix [361]. 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:

- $\mathsf{SimOCmpSwap}(\mathsf{addr}\langle x\rangle, \mathsf{addr}\langle y\rangle))$: Simulates swapping x and y given a hidden input bit.
- $\mathsf{SimOCmpSet}(\mathsf{addr}\langle x\rangle), \mathsf{addr}\langle y\rangle))$: Simulates setting x to y given a hidden input bit.
- $\mathsf{SimOSort}(\mathsf{addr}\langle L\rangle, n, f)$: Simulates sorting list L of length n by ordering function f.
- $\mathsf{SimOCompact}(\mathsf{addr}\langle L\rangle, n, \mathsf{addr}\langle B\rangle, m)$: 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 [102], which is a two-tiered oblivious hash table that consists of the polynomial-time algorithms (Construct, GetBuckets):

- $T \leftarrow \mathsf{OHashTable.Construct}(D)$: Given some data D, output a two-tiered oblivious hash table T.
- $(B_1, B_2) \leftarrow \mathsf{OHashTable.GetBuckets}(T, \mathsf{idx})$: 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 [102], we can assume the existence of a simulator SimOHashTable with algorithms (Construct, GetBuckets):

- $T \leftarrow \mathsf{SimOHashTable.Construct}(\mathsf{addr}\langle D\rangle, n)$: Given the address of data D of size n, simulate constructing an oblivious hash table.
- $(B_1, B_2) \leftarrow \mathsf{SimOHashTable.GetBuckets}(T, \mathsf{addr}\langle \mathsf{idx}\rangle)$: 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 19 that provides the interface defined in Section 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 7. *Given a two-tiered oblivious hash table [102], an oblivious compare-and-set operator, and an oblivious compaction algorithm, the subORAM scheme described in Section 4.5 and formally defined in Figure 19 is secure according to Definition 10.*

Proof. We construct our simulator in Figure 20 (we leave some empty lines so that corresponding operations in Figure 19 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 19 and Figure 20).

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.

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 corresponding 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 experiments are indistinguishable. The correctness of the results follows from Theorem 12, 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 21, as well as a load balancer scheme in Figure 23 and Figure 25 (we leave some empty lines in the

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

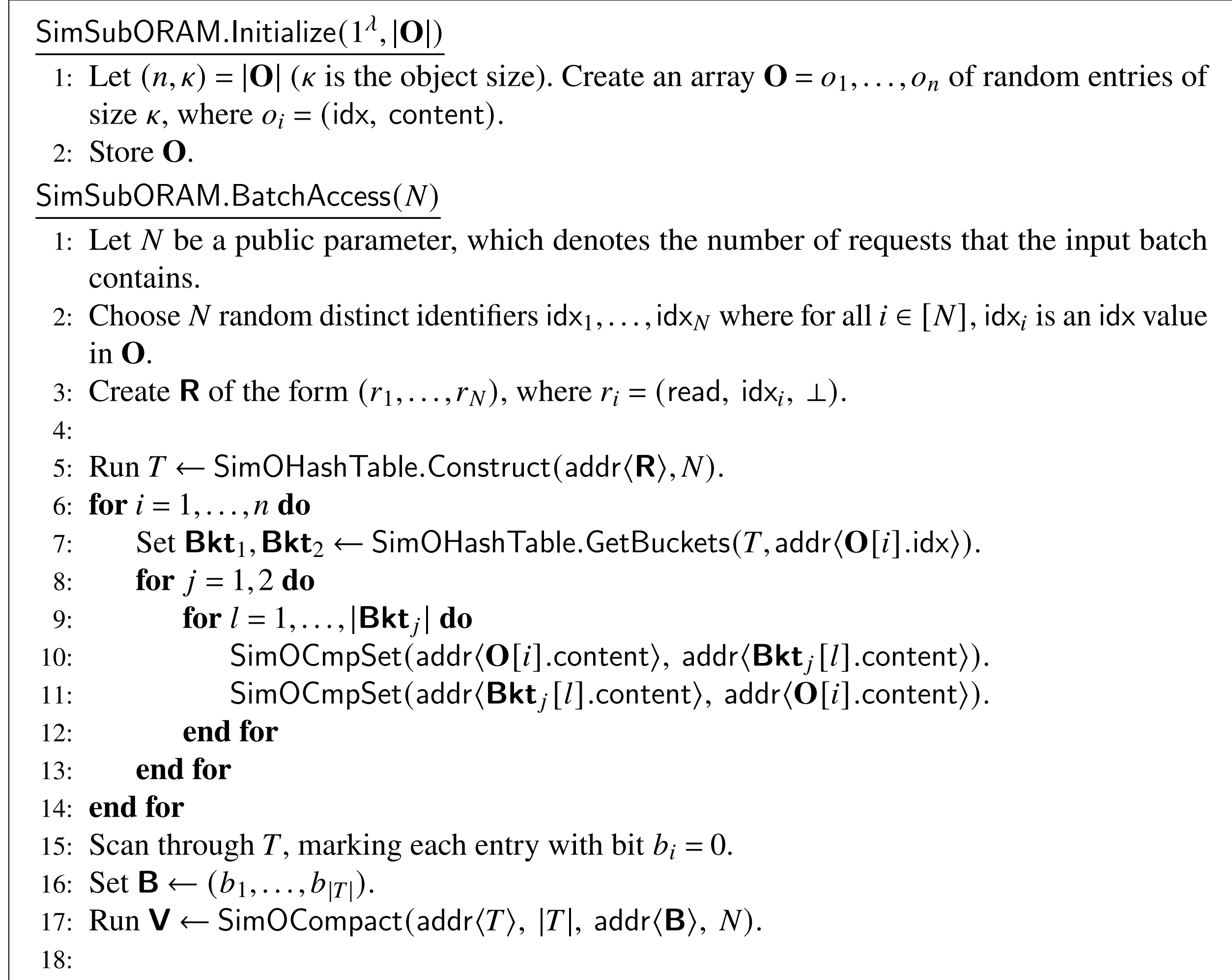
$\mathbf{V} \leftarrow$ SubORAM.BatchAccess($\mathbf{R}$)
1: Parse $\mathbf{R}$ as $(r_1, \ldots, r_N)$, where r_i = (type, idx, content).
2: **if** $\mathbf{R}$ contains duplicates **then**
3: **return** $\perp$.
4: **end if**
5: Set $T \leftarrow$ OHashTable.Construct($\mathbf{R}$).
6: **for** $i = 1, \ldots, n$ **do**
7: Set $\mathbf{Bkt}_1, \mathbf{Bkt}_2 \leftarrow$ OHashTable.GetBuckets(T, $\mathbf{O}[i]$.idx).
8: **for** $j = 1, 2$ **do**
9: **for** $l = 1, \ldots, |\mathbf{Bkt}_j|$ **do**
10: OCmpSet(($\mathbf{Bkt}_j[l]$.idx $\stackrel{?}{=}$ $\mathbf{O}[i]$.idx), $\mathbf{O}[i]$.content, $\mathbf{Bkt}_j[l]$.content).
11: OCmpSet(($\mathbf{Bkt}_j[l]$.idx $\stackrel{?}{=}$ $\mathbf{O}[i]$.idx) $\wedge$ ($\mathbf{Bkt}_j[l]$.type $\stackrel{?}{=}$ write), $\mathbf{Bkt}_j[l]$.content, $\mathbf{O}[i]$.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 19: Our subORAM construction.

protocol figures and corresponding simulator figures so that corresponding operations have the same line number).

Theorem 8. *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 10), Snoopy, as described in Section 4.4 and formally defined in Figure 21, is secure according to Definition 6.*

Proof. Our Snoopy construction is presented in Figure 21, with the corresponding simulator in Figure 22. 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.



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

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 $\mathbf{O}$.

$\mathsf{SimSubORAM.BatchAccess}(N)$

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 20: Simulator algorithms SimSubORAM = (Initialize, BatchAccess).

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 8, 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 21 and Figure 22, 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 23, Figure 24, Figure 25, Figure 26).

Initialization.

- (Lines 1-2) The load balancer algorithm takes an array **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.
- (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.

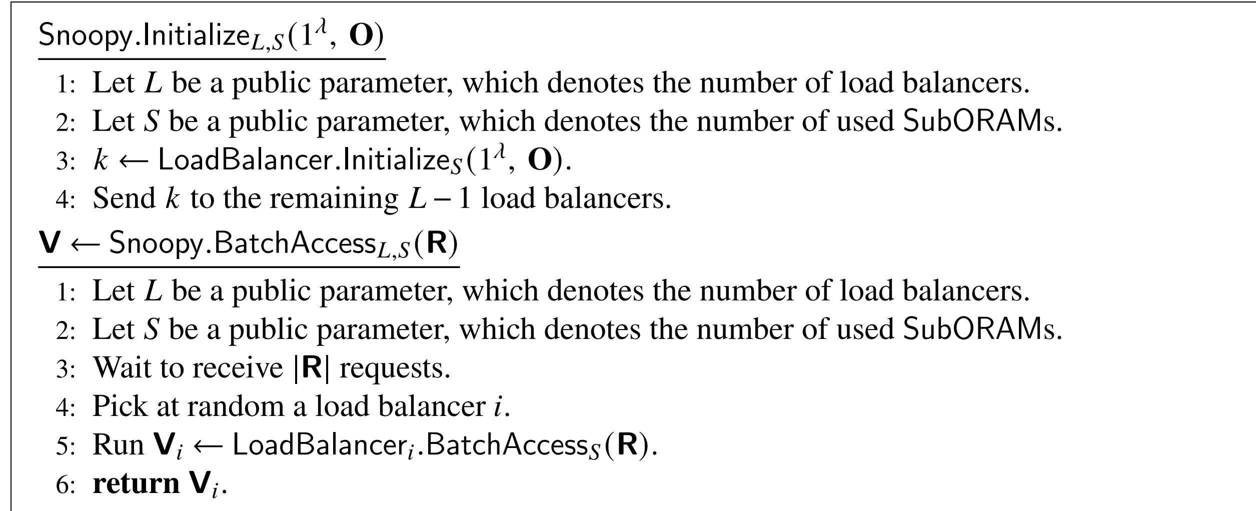
- (Line 18) These lines are identical and create an array where the number of objects is based on public information and the object size is a public parameter.
- (Line 19) These lines set the same function.
- (Line 20) By the security of the underlying sorting algorithm, the oblivious sort and the corresponding 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 9, and so the probability that a batch overflows is negligible in λ. Finally, Theorem 11 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 25, 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



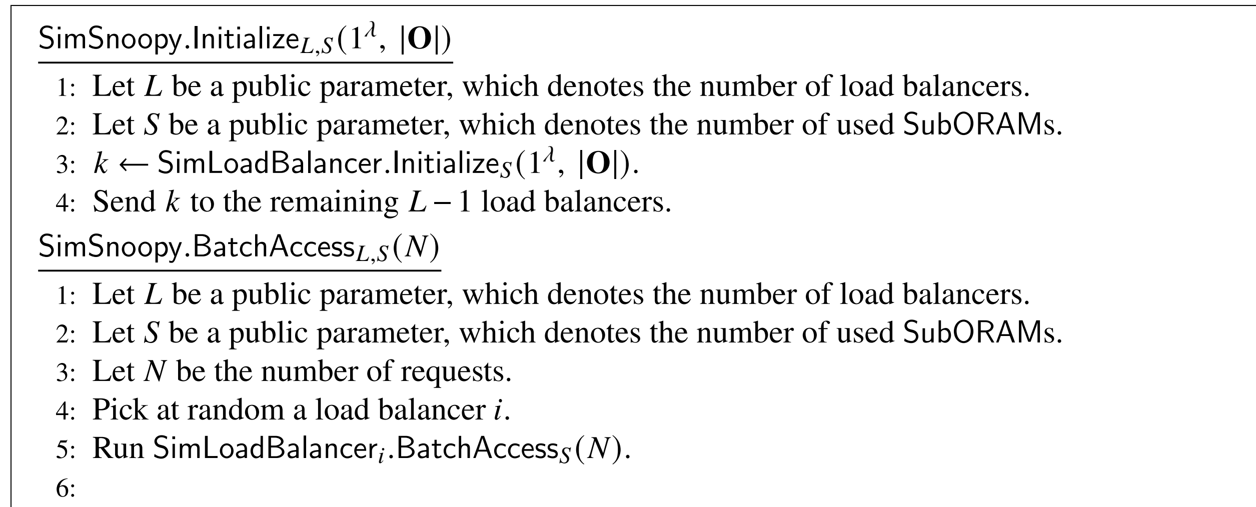
$\underline{\mathsf{Snoopy.Initialize}_{L,S}(1^\lambda,\ \mathbf{O})}$

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.

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

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 21: Our Snoopy construction.



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

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.

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

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 22: Simulator algorithms SimSnoopy = (Initialize, BatchAccess).

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:

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

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 $\mathsf{prev} \leftarrow \bot$.
12: **for** $i = 1, \ldots, |\mathbf{O}|$ **do**
13: **if** $\mathbf{O}[i].t \neq \mathsf{prev}$ **then**
14: Let $y_x \leftarrow i$.
15: Let $x \leftarrow x + 1$.
16: Let $\mathsf{prev} \leftarrow \mathbf{O}[i].t$.
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 23: 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.

- 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 ($o \underset{rt}{\longrightarrow} o'$) 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 $o \longrightarrow o'$ if o' follows o in the linearization) such that:

- The linearization respects the real-time order of operations in the set: If $o \underset{rt}{\longrightarrow} o'$ then $o \longrightarrow o'$ **(C1)**.

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

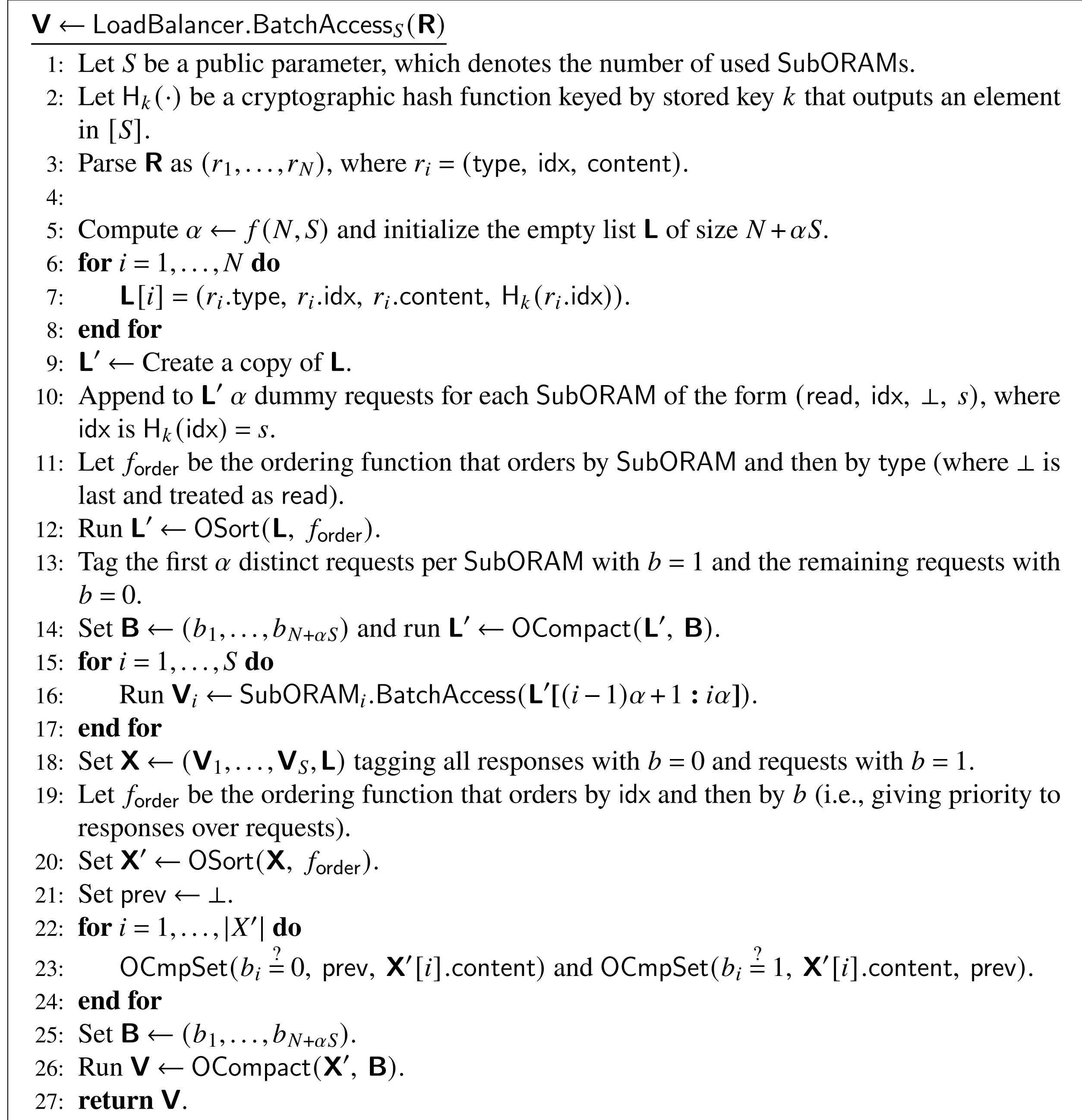
1: Let $(n, \kappa) = |\mathbf{O}|$. ▷ κ is the size of the object
2: Create an array $\mathbf{O}$ $(1, o_1), (2, o_2), \ldots, (n, o_n)$ of the form (idx, content), where o_i is a random entry of size κ.
3: Let H be a keyed cryptographic hash function that outputs an element in $[S]$.
4: Sample a secret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

Figure 24: 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.

- 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 $o \underset{rt}{\longrightarrow} o'' \underset{rt}{\longrightarrow} o'$, 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 11. *Snoopy is linearizable when the subORAM is instantiated with a oblivious storage scheme that is secure according to Definition 10.*



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

1: Let S be a public parameter, which denotes the number of used SubORAMs.
2: Let $\mathsf{H}_k(\cdot)$ be a cryptographic hash function keyed by stored key k that outputs an element in $[S]$.
3: Parse $\mathbf{R}$ as $(r_1, \ldots, r_N)$, where $r_i = (\mathsf{type},\ \mathsf{idx},\ \mathsf{content})$.
4:
5: Compute $\alpha \leftarrow f(N, S)$ and initialize the empty list $\mathbf{L}$ of size $N + \alpha S$.
6: **for** $i = 1, \ldots, N$ **do**
7: $\quad \mathbf{L}[i] = (r_i.\mathsf{type},\ r_i.\mathsf{idx},\ r_i.\mathsf{content},\ \mathsf{H}_k(r_i.\mathsf{idx}))$.
8: **end for**
9: $\mathbf{L}' \leftarrow$ Create a copy of $\mathbf{L}$.
10: Append to $\mathbf{L}'$ α dummy requests for each SubORAM of the form $(\mathsf{read},\ \mathsf{idx},\ \perp,\ s)$, where idx is $\mathsf{H}_k(\mathsf{idx}) = s$.
11: Let f_{order} be the ordering function that orders by SubORAM and then by type (where $\perp$ is last and treated as read).
12: Run $\mathbf{L}' \leftarrow \mathsf{OSort}(\mathbf{L},\ f_{\mathsf{order}})$.
13: Tag the first α distinct requests per SubORAM with $b = 1$ and the remaining requests with $b = 0$.
14: Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{N+\alpha S})$ and run $\mathbf{L}' \leftarrow \mathsf{OCompact}(\mathbf{L}',\ \mathbf{B})$.
15: **for** $i = 1, \ldots, S$ **do**
16: $\quad$ Run $\mathbf{V}_i \leftarrow \mathsf{SubORAM}_i.\mathsf{BatchAccess}(\mathbf{L}'[(i-1)\alpha + 1 : i\alpha])$.
17: **end for**
18: Set $\mathbf{X} \leftarrow (\mathbf{V}_1, \ldots, \mathbf{V}_S, \mathbf{L})$ tagging all responses with $b = 0$ and requests with $b = 1$.
19: Let f_{order} be the ordering function that orders by idx and then by b (i.e., giving priority to responses over requests).
20: Set $\mathbf{X}' \leftarrow \mathsf{OSort}(\mathbf{X},\ f_{\mathsf{order}})$.
21: Set $\mathsf{prev} \leftarrow \perp$.
22: **for** $i = 1, \ldots, |X'|$ **do**
23: $\quad \mathsf{OCmpSet}(b_i \stackrel{?}{=} 0,\ \mathsf{prev},\ \mathbf{X}'[i].\mathsf{content})$ and $\mathsf{OCmpSet}(b_i \stackrel{?}{=} 1,\ \mathbf{X}'[i].\mathsf{content},\ \mathsf{prev})$.
24: **end for**
25: Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{N+\alpha S})$.
26: Run $\mathbf{V} \leftarrow \mathsf{OCompact}(\mathbf{X}',\ \mathbf{B})$.
27: **return** $\mathbf{V}$.

Figure 25: Our load balancer construction.

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

SimLoadBalancer.BatchAccess(N)

1: 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.
2: Let $\mathsf{H}_k(\cdot)$ be a cryptographic hash function keyed by stored key k that outputs an element in $[S]$.
3: Choose N random identifiers $\mathsf{idx}_1, \ldots, \mathsf{idx}_N$ where for all $i \in [N]$, idx_i is an idx value in $\mathbf{O}$.
4: Create $\mathbf{R}$ of the form $(r_1, \ldots, r_N)$, where $r_i = (\mathsf{read}, \mathsf{idx}_i, \perp)$.
5: Compute $\alpha \leftarrow f(N, S, \lambda)$ and initialize the empty list $\mathbf{L}$ of size $N + \alpha S$.
6: **for** $i = 1, \ldots, N$ **do**
7: $\quad \mathbf{L}[i] = (r_i.\mathsf{type}, r_i.\mathsf{idx}, r_i.\mathsf{content}, \mathsf{H}_k(r_i.\mathsf{idx}))$.
8: **end for**
9: $\mathbf{L}' \leftarrow$ Create a copy of $\mathbf{L}$.
10: Append to $\mathbf{L}'$ α dummy requests for each SubORAM of the form $(\mathsf{read}, \mathsf{idx}, \perp, s)$, where idx is $\mathsf{H}_k(\mathsf{idx}) = s$.
11: Let f_{order} be the ordering function that orders by SubORAM and then by type (where $\perp$ is last and treated as read).
12: Run SimOSort(addr$\langle \mathbf{L}' \rangle$, $|\mathbf{L}'|$, f_{order}).
13: Tag the first α requests per SubORAM with $b = 1$ and the remaining requests with $b = 0$.
14: Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{N+\alpha S})$ and run SimOCompact(addr$\langle \mathbf{L}', \rangle, N + \alpha S$, addr$\langle \mathbf{B} \rangle$, αS).
15: **for** $i = 1, \ldots, S$ **do**
16: $\quad$ Run $\mathbf{V}_i \leftarrow$ SimSubORAM$_i$.BatchAccess(α).
17: **end for**
18: Let $\mathbf{X}$ be an array of $N + \alpha S$ objects the same size as the objects in $\mathbf{L}$ with a tag bit.
19: Let f_{order} be the ordering function that orders by idx and then by b (i.e., giving priority to responses over requests).
20: Run SimOSort(addr$\langle \mathbf{X} \rangle$, $|\mathbf{X}|$, f_{order}).
21: Set prev $\leftarrow \perp$.
22: **for** $i = 1, \ldots, |\mathbf{X}'|$ **do**
23: $\quad$ SimOCmpSet(addr$\langle$prev$\rangle$, addr$\langle \mathbf{X}'[i]$.content$\rangle$) and SimOCmpSet(addr$\langle \mathbf{X}'[i]$.content$\rangle$, addr$\langle$prev$\rangle$).
24: **end for**
25: Set $\mathbf{B} \leftarrow (b_1, \ldots, b_{N+\alpha S})$.
26: Run SimOCompact(addr$\langle \mathbf{X}' \rangle$, $|\mathbf{X}'|$, addr$\langle \mathbf{B} \rangle$, N).
27:

Figure 26: Load balancer simulator for SimLoadBalancer.BatchAccess.

$o_1 \to o_2 \to \ldots \to ..o_n$ be the resulting linearization. We prove the aforementioned statement in two steps: (1) the statement holds true for $o_n \to o_{n+1}$, 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 $o \underset{rt}{\longrightarrow} o'' \underset{rt}{\longrightarrow} o'$. 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 12) 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 12. *Our subORAM (Figure 19) always returns the value of the latest write to an object, provided that it is instantiated from a two-tiered oblivious hash table [102], 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 the chapter, 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 19) 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

Snoopy is 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 [128].

The end-to-end encrypted messaging application Signal uses techniques from Snoopy in their private contact discovery system, which matches Signal users to users' contacts [118]. Signal uses some of our techniques for horizontally scaling ORAM, along with techniques from Oblix [361] and PathORAM [473] in their new, deployed system. Signal reported that the new system allowed them to use roughly 10 servers instead of roughly 600 for private contact discovery [516].

Part II

Secure accounts

Chapter 5

SafetyPin: An encrypted backup system that resists hardware attacks

5.1 Introduction

Modern mobile phones and tablets back up sensitive data to the cloud. To protect users' privacy, this data must be encrypted under keys that are not available to the cloud provider. Unfortunately, with 3.8 billion smartphone users, it is impractical to expect them all to store, say, a 128-bit AES backup key. Not everyone has a computer, or trustworthy friends who can keep shares of a backup key, or even a safe place to store a backup key on paper. As a result, mobile OSes have fallen back to protecting backups with the least common denominator: device screen-lock PINs. Using PINs is good for security because a user's screen-lock PIN never leaves her device (so the cloud provider never learns it). Using PINs is good for usability because users generally remember them.

Unfortunately, PINs have such low entropy (e.g, six decimal digits) that no feasible amount of key stretching can protect against brute-force PIN-guessing attacks. Instead, modern backup systems—such as those from Apple [296], Google [507], and Signal [337]—rely on secure hardware devices like hardware-security modules (HSMs) to thwart brute-force attacks. Specifically, devices encrypt their backup keys under the public keys of HSMs, but each device includes a hash of its screen-lock PIN as part of the plaintext. HSMs return decrypted plaintext only to clients that can supply this PIN hash. Furthermore, HSMs limit the number of decryption attempts for any given user account. For fault tolerance, a device typically encrypts its backup key to the public keys of some number of HSMs, allowing any one of the these to recover the backup key.

This status quo still falls short of acceptable privacy for two reasons. First, HSMs are not perfect, yet each HSM in these systems is a single point of security failure for millions of users' backup keys. Second, these systems make it difficult for clients to detect security breaches. For instance, if a malicious insider working in a data center physically steals an HSM, then to anyone outside the company it looks like an unremarkable single hardware failure. Alternatively, if an insider successfully guesses someone's PIN, the victim may have no idea her backup was ever compromised.

This chapter presents SafetyPin, a PIN-based encrypted-backup system with stronger security

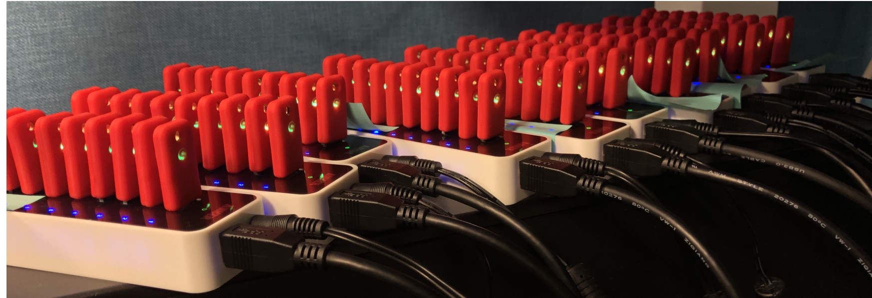


Figure 1: Our cluster of 100 low-cost hardware security modules (SoloKeys [463]) on which we evaluate SafetyPin.

properties. The key idea behind SafetyPin is that recovering any user's backed-up data either requires (a) guessing the user's PIN or (b) compromising a very large number of HSMs—e.g., 6% of all HSMs operated by a provider. (The 6% figure here is a tunable system parameter.) Such large-scale attacks would typically need to span multiple data centers, be harder for insiders to pull off undetected against physical devices, cost more, and also likely cause service disruptions visible to end users.

One way to achieve SafetyPin's security goal would be to threshold-encrypt the client's hashed PIN and backup key in such a way that decrypting the client's backup key would require the participation of 6% of all HSMs in the system. Unfortunately, this approach lacks scalability. If each client recovering a backup must interact with 6% of the system's HSMs, adding more HSMs improves security without improving throughput. As the number of HSMs in the system increases, we would like the system's overall throughput to increase in tandem with its security (i.e., the attacker's cost).

To achieve scalability, SafetyPin takes a different approach: devices threshold-encrypt their backup keys to a small cluster of n HSMs such that decryption requires the participation of most HSMs in the cluster. The cluster size n is independent of the total number of HSMs in the system, and depends on both the fraction of compromised HSMs the system can tolerate and the fraction of HSMs that can fail-stop. (For example, to tolerate the compromise of 6% of HSMs where half of a cluster is allowed to fail-stop, we can set the cluster size $n = 40$.) This design achieves our scalability goal, since each device need only communicate with a small fixed number of HSMs during recovery. This design also achieves our security goal because the cluster of n HSMs that can decrypt a client's backup depends on the client's secret PIN, via a primitive we introduce called *location-hiding encryption*. Hence, even if an attacker compromises 6% of the HSMs in the system as a whole, the chances that the attacker compromises a "useful" set of HSMs—i.e., at least half of the HSMs in the device's chosen cluster—is very small. More precisely, we show that if the total number of HSMs in the system is large enough (a few hundred or more), the probability that an attacker can decrypt a backup via HSM compromise is not much higher than the probability of simply guessing the client's PIN.

In modern backup systems, each HSM only needs to monitor the number of PIN attempts for a small subset of users, but because of our location-hiding encryption primitive, every HSM needs to be able to verify the number of PIN attempts for every user. To maintain this information scalably, the

HSMs use a new type of distributed log. Third parties can monitor this log to alert users whenever a backup-recovery attempt is underway. Since a compromised service provider may see which HSMs a mobile device interacts with during recovery (and could compromise those HSMs to recover the users' backed-up data), HSMs revoke their ability to decrypt backups after completing the recovery process. Implementing this revocation requires adapting "puncturable encryption" [215] to storage-limited HSMs. While our prototype is focused on PIN-protected backups, these primitives have potentially broader applicability to problems such as private storage in peer-to-peer systems and cryptocurrency "brain wallets."

We implemented SafetyPin on low-cost SoloKey HSMs [463]. We evaluate the system using a cluster of 100 SoloKeys (Figure 1) and an Android phone (representing the client device). Generating a recovery ciphertext on the client, excluding the time to encrypt the disk image, takes 0.37 seconds. To process 1B recoveries a year, or 123K recoveries per hour, we estimate that we would need 3,100 SoloKeys. In a SafetyPin deployment of 3,100 HSMs, tolerating the compromise of 6% of the HSMs (i.e., 194 HSMs), the client must interact with a cluster of 40 HSMs during recovery. Running our backup-recovery protocol across a cluster of this size takes 1.01 seconds.

Limitations. A limitation of SafetyPin is that the set of HSMs a device uses for recovery can leak information about the user's PIN. In particular, an attacker who controls the data center can learn a salted hash of the user's PIN during recovery. This is unfortunate in the common case that people re-use the same PIN after recovery [136, 194, 246, 454]. We discuss one mitigation in Section 5.8. Also, while it is possible to detect when PINs can safely be re-used, we have not yet implemented this functionality.

In addition, SafetyPin is more expensive than today's PIN-based backup systems. SafetyPin requires the data center operator to operate a much larger fleet of HSMs (roughly $50-100\times$ larger) than the standard HSM-based backup systems require. SafetyPin clients must also download roughly 2MB of keying material per day in a SafetyPin deployment supporting one billion recoveries per year, due to the periodic rotation of large HSM keys. Even so, we expect that the cost of storing and transferring disk images (GBs/user) will dwarf these costs.

5.2 The setting

Entities. Our encrypted-backup system involves three entities, whose roles we describe here.

Client. Initially, the client holds (1) a username with the service provider, (2) a human-memorable passphrase or PIN, (3) a disk image to be backed up, and (4) the public keys of the service provider's HSMs. Later on, the client should be able to recover her backed-up data using only her username, her PIN, and access to the other components of the backup system.

In SafetyPin, as in today's PIN-based backup systems, security depends on the client having access to the HSMs' true public keys: If a malicious service provider can swap out the HSMs' true public keys for its own public keys without detection, the service provider can immediately break security. Using a distributed log (Section 5.6) can ensure that all clients see a common set of HSM

Device	Price	g^x/sec.	Storage	FIPS
SoloKey [463]	$20	8	256 KB*	
YubiHSM 2 [530]	$650	14	126 KB	
SafeNet A700 [441]	$18,468	2,000	2,048 KB	✓
Intel i7-8569U (CPU)	$431	22,338	n/a	

Table 2: Hardware security modules offer physical security protections but are computationally weak compared to a standard CPU.

he g^x/sec is NIST P256 elliptic-curve point-multiplications per second. “FIPS?” refers to whether the device meets the FIPS 140-2 standard for HSMs. (* The 256 KB storage on the SoloKey is shared between code and data.)

public keys, to prevent targeted attacks. Hardware-attestation techniques, as used in the FIDO [416] and SGX [264] specs, can provide another defense.

We also assume the provider has traditional account authentication (e.g., Gmail passwords) to prevent random third parties from consuming PIN guesses, but we omit this from the discussion for simplicity.

Service provider. The service provider offers the encrypted-backup service to a pool of clients and it maintains the data centers in which the backup system runs. For example, the service provider could be a mobile-phone vendor, such as Apple or Google. The service provider’s data centers contain the network infrastructure that connects the HSMs. They also contain large amounts of (potentially untrustworthy) storage and computing resources. Our security properties will hold against a service provider that becomes compromised at any point after the system is set up.

Hardware security modules (HSMs). The service provider’s data centers contain thousands of hardware security modules. An HSM is a tamper-resistant computing device meant for storing cryptographic secrets. HSMs have fully programmable processors but are typically resource-poor (see Table 2). It is possible to lock an HSM’s firmware before deployment, which makes remote compromise and key-extraction attacks more difficult. Each HSM has a public key and stores the corresponding secret key in its secure memory.

The attack scenario. The service provider (Apple, Google, etc.) spends vast amounts of money acquiring a large user base for products that store user data in the cloud. The provider risks reputational damage and journalistic scrutiny if it cannot ensure the durability and confidentiality of user data.

A service provider can deploy SafetyPin as a way to build trust among its user base and to protect its own infrastructure against future compromise. By enlisting third-party organizations to monitor the SafetyPin deployment’s public distributed log, the provider can build further public trust in the system.

At some point after the provider deploys SafetyPin, a powerful attacker wishes to steal user data. The attacker may have malicious insiders working for the provider. It may physically compromise data centers to steal HSMs. It may intercept shipments to tamper with some of the HSMs on their

way to the data center. The attacker could also be a state actor employing legal pressure to gain access to data centers. Nonetheless, the attacker is sensitive to both the cost of attacks and the risk of public exposure.

Both the attack cost and risk of exposure increase with the number of HSMs the attacker must compromise. For instance, while a malicious insider working at a data center may be able to abscond with a single HSM—passing the missing device off as a hardware failure—removing 100 HSMs is a much riskier proposition. A state actor who can order the provider to hand over HSMs may be dissuaded if doing so will attract press coverage either by making non-targeted clients' data unrecoverable or creating a damning public audit trail.

The attacker may compromise clients as well as the provider. For instance, the attacker may have a good guess at a target user's PIN, perhaps because of CCTV footage showing the user unlocking a mobile device. While SafetyPin cannot prevent the attacker from gaining access to the data with the correct PIN, the risk will be higher to the attacker if stolen PINs cannot be used without exposing the attack in SafetyPin's public distributed log.

Notation. The set $\mathbb{Z}^{>0}$ refers to the set of natural numbers $\{1, 2, 3, \dots\}$. For a positive integer n, we let $[n] = \{1, \dots, n\}$ and we use $\perp$ to denote a failure symbol. For strings a and b, we write their concatenation as $a \| b$. Throughout, we use λ to denote the security parameter, and we typically take $\lambda = 128$ (i.e., for 128-bit security).

5.3 System goals

SafetyPin implements an encrypted-backup functionality, which consists of two routines:

- the *backup* algorithm, which the client uses to produce its encrypted backup, and
- the *recovery* protocol, in which the client uses HSMs to recover the backup plaintext from ciphertext.

We define these protocols with respect to a number of HSMs $N \in \mathbb{Z}^{>0}$ and a finite PIN space $\mathcal{P} \subseteq \{0, 1\}^*$. For convenience, we define the *master public key* mpk for a data center to be all N HSMs' public keys: $\mathsf{mpk} = (\mathsf{pk}_1, \dots, \mathsf{pk}_N)$. The syntax of an encrypted-backup system is then as follows:

$\mathsf{Backup}(\mathsf{mpk}, \mathsf{user}, \mathsf{pin}, \mathsf{msg}) \to \mathsf{ct}$. Given the master public key mpk, a client username user, the client's PIN $\mathsf{pin} \in \mathcal{P}$, and a message $\mathsf{msg} \in \{0, 1\}^*$ to be backed up, output a recovery ciphertext ct. This routine runs on the client and requires no interaction with HSMs. The client uploads the resulting ciphertext ct to the service provider.

$\mathsf{Recover}^{\mathcal{S}, \mathcal{H}_1, \dots, \mathcal{H}_N}(\mathsf{mpk}, \mathsf{user}, \mathsf{pin}, \mathsf{ct}) \to \mathsf{msg}$ or $\perp$. The client initiates the recovery routine, which takes as input the master public key mpk, a client username user, a PIN $\mathsf{pin} \in \mathcal{P}$, and a recovery ciphertext ct.

During the execution of $\mathsf{Recover}$, the client interacts with the service provider $\mathcal{S}$ and a subset of the HSMs $\mathcal{H}_1, \dots, \mathcal{H}_N$. Each HSM $\mathcal{H}_i$ holds the master public key mpk, and its secret decryption key sk_i. During recovery, the data center provides the client's username user to each HSM.

The recovery routine outputs a backed-up message $\mathsf{msg} \in \{0, 1\}^*$ or a failure symbol $\perp$.

We now describe the security properties that such a system should satisfy. We work in an asynchronous network model; we use standard cryptographic primitives to set up authenticated and encrypted channels between the client, service provider, and HSMs.

Property 1: Security. If the client obtains the HSMs' true public keys, then even an attacker that:

- controls the service provider (in particular, is an active network attacker inside the data centers and has control of the service provider's servers and storage),
- compromises an f_{secret} (e.g., $f_{\text{secret}} = \frac{1}{16}$) fraction of HSMs in the data center *before* the client begins the recovery process, and
- compromises all of the HSMs in the data center *after* the recovery protocol completes,

still should learn nothing about any honest client's encrypted message (in a semantic-security sense [207]) beyond what it can learn by guessing that client's PIN.

Discussion: The adversary can inspect all clients' recovery ciphertexts and then choose to compromise a large set of HSMs that depends on these ciphertexts. Such attacks are relevant when, for example, a state actor with the power to compromise many HSMs targets the backed-up data of a specific set of users.

Two important caveats are: (1) SafetyPin does not protect against an attacker compromising HSMs while recovery is in progress (see Figure 4) and (2) as implemented, SafetyPin does not protect the PIN: an adversary that observes which HSMs the client contacts during recovery may learn a salted hash of the PIN after recovery completes. Section 5.6.3 discusses how to detect and mitigate this leakage by protecting the salt.

Property 2: Scalability. The recovery protocol should require the client to interact with a constant number of HSMs, independent of the number of HSMs in the data center. (This constant may depend on the security parameter and on the fraction of HSMs whose compromise the system can tolerate.) Hence, providers can deploy additional HSMs to scale capacity. Concretely, when we configure the system to tolerate the compromise of $f_{\text{secret}} = \frac{1}{16}$ of the data center's HSMs, our protocol requires the client to communicate with 40 HSMs during recovery.

Property 3: Fault tolerance. Every client should be able to recover her encrypted message even if a constant fraction f_{live} (e.g. $f_{\text{live}} = \frac{1}{64}$) of the HSMs in the data center fail-stop.

Setting parameters. For the remainder of this chapter, we set the fraction of compromised HSMs that the system can tolerate to $f_{\text{secret}} = \frac{1}{16}$ and the fraction of HSMs that can fail while still allowing the client to recover her backup to $f_{\text{live}} = \frac{1}{64}$. This choice is reasonable because large companies have more than 16 data centers, while smaller companies can collaborate on a shared deployment with 16 physical security perimeters. By adjusting the other parameters, it is possible to achieve any $0 < f_{\text{secret}} < 1$ or $0 < f_{\text{live}} < 1$. (In Section 5.9.2, we discuss how the choice of these values affects other system parameters.)

5.4 Architecture overview

We now describe our encrypted-backup protocol (Figure 3) and explain how it satisfies the design goals of Section 5.3. We will discuss possible extensions and deployment considerations in

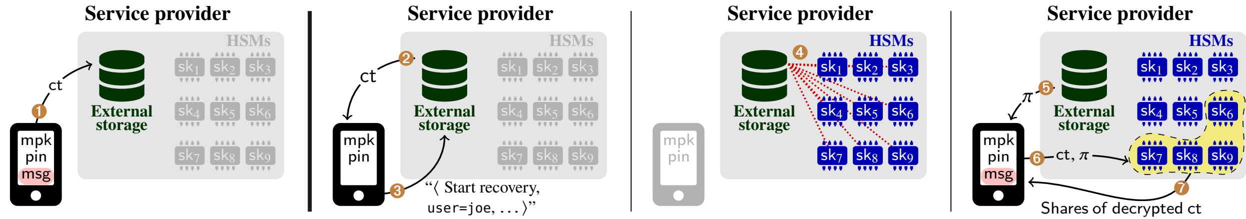


Figure 3: An overview of the recovery-protocol flow. Each HSM i holds a secret key sk_i. The client holds a vector mpk of all HSMs' public keys. ❶ During backup, the client uses its PIN and the master public key to encrypt its data msg into a recovery ciphertext ct. The client then uploads this recovery ciphertext ct to the service provider. ❷ During recovery, the client downloads its recovery ciphertext. ❸ The client asks the data center to log its recovery attempt. ❹ The service provider collects a batch of client log-insertion requests, updates the log, and aggregates the new log into a Merkle tree. The service provider and HSMs run a log-update protocol. At the end of this protocol, each HSM holds the root of the Merkle tree computed over the latest log. ❺ The service provider sends the client a Merkle proof π that the client's recovery attempt is included in the latest log (i.e., in the latest Merkle root). ❻ The client sends the recovery ciphertext ct and log-inclusion proof π to the subset of HSMs needed to decrypt the recovery ciphertext. ❼ The HSMs check the proof and return shares of the decrypted ciphertext to the client. The client uses these to recover the backed-up data msg.

Section 5.8.

5.4.1 The back-up process

The client begins the back-up process holding

- the public keys of all HSMs in the data center,
- its secret PIN, and
- a disk image to be backed up (the "message").

To back up its disk image, the client samples a subset of n HSMs out of the N total HSMs in the data center where $n \ll N$. The client chooses this subset by hashing (a) public information: the service name, its username, and a public salt the client chooses at random, and (b) its secret PIN. The client then encrypts its message with a random AES encryption key, and then splits this AES key into n threshold shares using Shamir secret sharing [453], such that any threshold t of the shares suffice to recover the AES key. The client then encrypts one share to the public key of each HSM in its chosen subset. To ensure that a ciphertext is bound a username, we use a username-specific hash function when encrypting (Section 5.11.4).

The client's recovery ciphertext then consists of: its public salt, the AES-encrypted message, the n encrypted shares of the AES key, and a configuration-epoch number that the service provider can use to identify the set of HSMs that were in service at the time the client created its backup. The client computes the ciphertext locally and uploads it to the backup service provider, with no HSM interactions required.

To explain why this construction is scalable: since only a constant number of HSMs $n \ll N$ participate in the decryption process, the system scales well as the number of HSMs in the data center increases.

To explain why this construction should be secure: if the attacker cannot guess the client's PIN, the attacker does not know which set of n HSMs (out of the N total) it needs to compromise to recover the client's AES key. So, the best attacks are either to: guess the client's PIN or compromise a large fraction of the data center.

This argument requires that each individual key-share ciphertext leak no information about which HSM can decrypt it—a cryptographic property known as "key privacy" [51]. However, even key-private encryption schemes do not always remain secure against an adversary that adaptively compromises secret keys, which leads to our first technical challenge:

Challenge 1. *How can we ensure that the client's recovery ciphertext "leaks nothing" about which HSMs are required to decrypt the client's message, even against an attacker who can adaptively compromise HSMs?*

In Section 5.5, we explain how to solve this problem using *location-hiding encryption*, a new cryptographic primitive.

5.4.2 The recovery process

The client begins the recovery process holding:

- the public keys of all HSMs in the data center,
- its secret PIN, and
- its recovery ciphertext (which the client can fetch from the service provider).

First, the client asks the service provider to record its recovery attempt in the *append-only log*, implemented collectively by the service provider and HSMs. The log holds a mapping of identifiers to values. The service provider can insert new identifier-value pairs into the log but the service provider cannot modify or delete the values of defined identifiers, ensuring that there is at most one immutable value for each identifier.

The recovery attempt is logged as follows. The client begins by using public information (service name, username, and salt in the recovery ciphertext) along with its secret PIN to recover the subset of n HSMs it picked during backup. The client then hashes these values together with some randomness to produce a cryptographic commitment h to the identities of these HSMs and to its recovery ciphertext. The client then asks the service provider to insert the identifier-value pair (user, h) into the log, where user is the client's username. (In this discussion, we use the client's username as the key for simplicity. In practice, to preserve privacy, we might use an opaque device-install UUID.)

The service provider collects a batch of these log-insertion requests, produces a Merkle-tree [360] digest over the updated log, and runs a log-update protocol with the HSMs. At the end of this protocol, the HSMs hold the updated log digest. The service provider then returns to the client a Merkle proof π proving that the pair (user, h) appears in the latest log digest.

Since the service provider and HSMs run the log-update protocol periodically (e.g., every 10 minutes), the client will have to wait a few minutes on average to decrypt its backup. The client already has to download its large encrypted disk image, which will likely take minutes, so these steps can proceed in parallel.

The client then contacts its chosen set of n HSMs over an encrypted channel, such as TLS. The client sends to each HSM: its username, the opening of its commitment h (i.e., the values and randomness used to construct the commitment h), and the Merkle inclusion proof π. Each HSM

- recomputes the commitment h and checks the inclusion proof π (to confirm that the recovery attempt is logged), and
- decrypts its share of the client's AES key.

If both of these checks pass, the HSM returns the AES-key share to the client.

Given any t of these decryption-key shares, the client can recover the AES key used to encrypt its backup. The client can then use this AES key to decrypt its backed-up message.

Since at most one log entry can exist per username, the use of the log ensures that each user can make at most one recovery attempt. In this way, the system defeats brute-force PIN-guessing attacks. With a slight modification, it is possible to allow each user to make a fixed number (e.g., 5) guesses, or a fixed number of guesses per time period (e.g., 5 per month).

A counter-intuitive property of this scheme is that the client never explicitly provides its PIN to the HSMs. The fact that the client knows which subset of the HSMs to contact implicitly proves the client's knowledge of the PIN because the set of n HSMs is much smaller than the total number of HSMs N.

This overview leaves some technical details unexplained. In particular:

Challenge 2. *How do the HSMs implement the append-only log without sacrificing scalability or security?*

A straightforward way to implement the log would be to have each HSM store the entire state of the log. But then every HSM would have to participate in every recovery attempt, which would not meet our scalability goals. Another implementation would be to have the data-center operators maintain the log, but then malicious data centers could violate the append-only property, and thus mount brute-force PIN-guessing attacks, without HSMs noticing.

In Section 5.6, we explain how the HSMs can collectively maintain such an append-only log in a scalable and secure manner. At a high level, the (potentially adversarial) data center maintains the state of the log, which we represent as a list of identifier-value pairs. Every time the data center wants to insert an identifier-value pair into the log, the data center must prove to a random subset of the HSMs that the identifier to be inserted is undefined in the current log. Provided that at least one honest HSM audits each log-insertion, we can guarantee that the values associated with log identifiers are immutable (i.e., that we maintain the log's append-only property). In this way, (a) each HSM needs to participate in only a vanishing fraction of the recovery attempts and (b) even an attacker who can compromise many of the HSMs cannot break the append-only nature of the log.

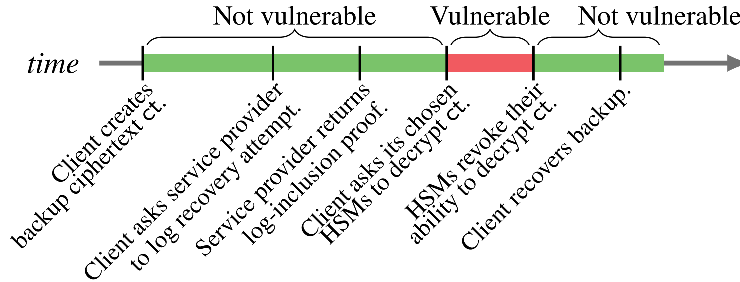


Figure 4: Since HSMs in SafetyPin revoke their ability to decrypt a client's recovery ciphertext, SafetyPin protects against HSM compromise attacks that take place before recovery begins and after it completes. An attacker who can compromise HSMs while recovery is in progress can break security.

One remaining issue is that an attacker who observes the data center network may see which HSMs a client interacts with during recovery and decide to compromise that exact set of HSMs after recovery completes.

Challenge 3. *For scalability, the client should only communicate with a small number of HSMs during recovery. But then how can we protect against an attacker who compromises these HSMs after recovery completes?*

Our idea is as follows: after a client runs the recovery protocol, each participating HSM *revokes* its ability to decrypt that client's recovery ciphertext. So, even if an after-the-fact attacker compromises the HSMs that participated in recovery, the attacker learns no useful information. The only window of vulnerability is at the moment after the client contacts its HSMs and before the HSMs complete revocation (Figure 4). We describe how to make this work on resource-limited HSMs in Section 5.7.

5.5 Protecting the mapping of users to HSMs with location-hiding encryption

In this section, we define and construct *location-hiding encryption*, which the client uses to encrypt its backup data.

The location-hiding encryption routine takes as input (1) a set of N public keys, (2) a short PIN, and (3) a message, and outputs a ciphertext. In our application, the N public keys are the public keys of the N HSMs in the data center.

The cryptosystem has three main properties, which we formalize in Section 5.11:

1. Security. To successfully decrypt the ciphertext, an attacker must either (a) guess the PIN or (b) control more than a constant fraction f_{secret} of the N total secret keys. This security property must hold even if the attacker can adaptively compromise an f_{secret} fraction of the N secret keys. In our application, this implies that unless an adversary can guess the PIN or compromise a constant f_{secret} fraction of the HSMs in the data center, it learns nothing about the client's backed-up data.

2. Scalability. Given the PIN used to encrypt the message, it is possible to decrypt the message using a small subset of the N secret keys corresponding to the N public keys used during encryption. In our application, a client who knows the correct PIN can recover its backup by interacting with only a small cluster of n HSMs (for some parameter $n \ll N$) out of the N total HSMs, So as N grows, each HSM needs to participate in a vanishing fraction of the total recovery attempts.

3. Fault tolerance. Given the PIN, it is possible to decrypt a ciphertext even if a random fraction f_{live} of all secret keys are unavailable. In our application, this implies clients can recover their backups even if an f_{live} fraction of all HSMs fail.

We call this primitive "location-hiding encryption" because there is a small set of n HSMs that the attacker could compromise to decrypt the ciphertext, but the cryptosystem *hides the location* of these HSMs within the larger pool of N HSMs.

5.5.1 Our construction

Our construction of location-hiding encryption is just a careful composition of existing primitives. However, it takes some analysis to prove that the composition provides the desired security properties. We describe our construction here in prose and we include the security definitions and proofs in Section 5.11. The construction makes use of a public-key encryption scheme (hashed ElGamal encryption [71,166]) and an authenticated encryption scheme (e.g., AES-GCM). We provide domain separation between hashed ElGamal ciphertexts by prepending the client's username and other recovery parameters to the hash function used in encryption and decryption (see Section 5.11.4).

Setup. In our construction, each HSM i, for $i \in [N]$, holds a keypair $(\mathsf{pk}_i, \mathsf{sk}_i)$ for the public-key encryption scheme. Let $t \in \mathbb{N}$ be a threshold such that if each HSM fails with probability f_{live}, then in a random sample of n HSMs, there are at least t non-failed HSMs with extremely high probability. Our instantiation takes $t = n/2$ for $f_{\mathsf{live}} = \frac{1}{64}$.

Encryption. The encryption routine takes as input a list of N public keys $(\mathsf{pk}_1, \dots, \mathsf{pk}_N)$, a PIN, and a message msg. To encrypt the message using our location-hiding encryption scheme:

1. Sample a random AES key k and a random salt.
2. Split k into t-out-of-n-Shamir secret shares $k_1, \dots, k_n$ [453].
3. Hash the PIN and salt and use the result as a seed to generate a list of n random indices $i_1, \dots, i_n \in [N]$.
4. Encrypt each key-share k_j with public key pk_{i_j}.
5. Finally, return (a) the salt, (b) the n public-key ciphertexts, and (c) the AES encryption of msg under key k.

Decryption. To decrypt given the ciphertext and PIN:

1. Hash the salt and PIN to reconstruct the set of indices $i_1, \dots, i_n \in [N]$ used during encryption.
2. Use secret keys $\mathsf{sk}_{i_1}, \dots, \mathsf{sk}_{i_n}$ to decrypt the n shares of the AES key k. (In fact, only t of the shares are necessary.)
3. Using the recovery routine for Shamir secret sharing, recompute the AES key k from its shares.

4. Decrypt and return msg using the AES key k.

Notice that the decryption routine only uses the PIN to sample the set of secret keys used for decryption. In our application, this implies that the client never needs to explicitly provide its PIN (or even a hash of its PIN) to the HSMs; contacting the right subset of HSMs is enough to ensure that the client provided the correct PIN.

The intuition behind the security analysis is straightforward: with hashed ElGamal encryption, the ciphertext reveals no information about which n public keys (out of the N total where $n \ll N$) were used during encryption. Thus, the ciphertext reveals no information about which secret keys the attacker must compromise unless the attacker can guess the PIN. Without these secret keys, the attacker cannot learn anything about k, and therefore cannot decrypt the message.

The following theorem, which we prove as Theorem 22 in Section 5.11.6 makes this argument precise:

Theorem (Informal). *The location-hiding encryption scheme of Figure 15 instantiated with the hashed ElGamal encryption scheme (Section 5.11.4) over a group $\mathbb{G}$ is* secure *(in the sense of Definition 15) for certain values of n and N, provided that:*

- *the computational Diffie-Hellman problem is hard in $\mathbb{G}$,*
- *the authenticated-encryption scheme is secure, and*
- *we model the hash functions used in the construction as random oracles.*

There are two reasons why the security analysis is non-trivial: First, we must ensure that the ciphertext leaks nothing about the n keys to which it was encrypted (i.e., that it is *key-private* [51]). Second, we must ensure that the encryption scheme remains secure even if an attacker can adaptively compromise secret keys. This is known as security under *selective-opening attack* [52, 175, 256]. Showing that both properties hold at once is the source of the technical complexity.

5.6 The distributed log

In SafetyPin, the HSMs collectively maintain a *distributed log*, which any external party can read and replay. The service provider maintains the log state and the HSMs monitor log insertions to ensure that the service provider does not violate the log's append-only property.

We use this log for two primary purposes:

1. **Limiting PIN guesses.** To prevent an attacker from brute-force guessing a client's PIN, we use the log (as described in Section 5.4) to enforce a global limit on the number of recovery attempts that the HSMs allow per username.
2. **Monitoring recovery attempts.** The service provider logs each recovery attempt, so any SafetyPin client can inspect the log to learn whether someone (e.g., a foreign attacker or snooping acquaintance) has tried to recover their backed-up data. A client could then take mitigating action—such as contacting their service provider, a law-enforcement agency, or the press.

A third use for the log—which comes directly from related work [183] and which we have not yet implemented—is to **manage HSM group membership**. Whenever the service provider wants

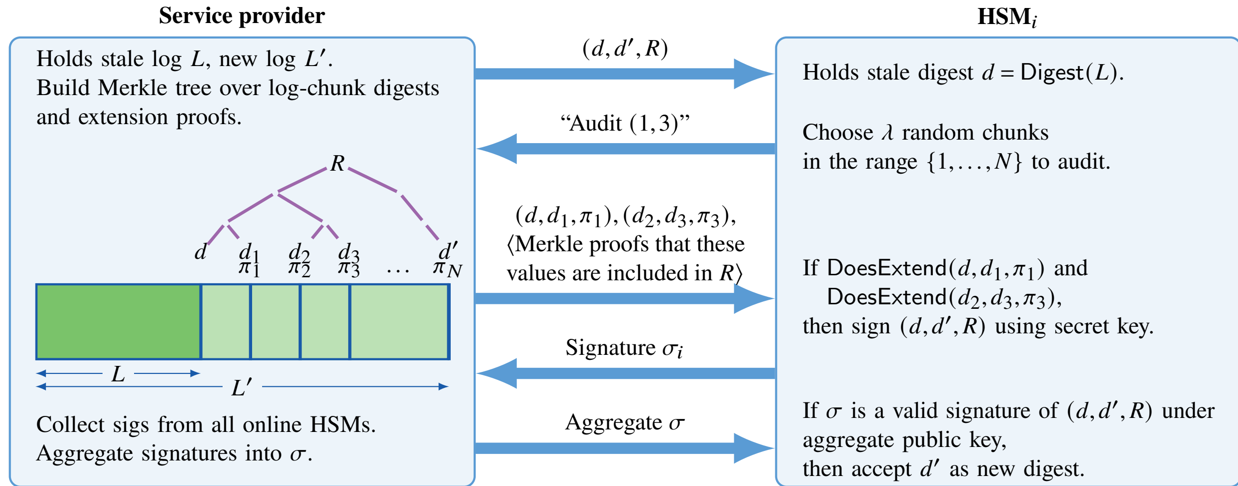


Figure 5: The protocol that the service provider and HSMs use to update the HSM's log digest.

to add or remove an HSM from the data center, the service provider operator could record this information in the log before the other HSMs will accept the change. All SafetyPin clients can thus verify that they are communicating with the same set of HSMs. In addition, clients can also detect suspicious changes in the set of HSMs in the data center. (For example, if the service provider replaces all HSMs in the data center over the course of a day.)

The log is simply a list of identifier-value pairs maintained by the service provider. Clients can insert identifier-value pairs in order to record recovery attempts, and HSMs maintain a digest of the log state. Our distributed log must satisfy the following key property:

> *If any honest HSM ever accepts that an identifier-value pair* $(\mathsf{id}, \mathsf{val})$ *is included in the log, the HSM should never accept that* $(\mathsf{id}, \mathsf{val}')$ *is included in the log, for any value* $\mathsf{val}' \neq \mathsf{val}$.

5.6.1 Underlying data structure

Terminology. The log L is a list of key-value pairs. Since we use the word "key" in this chapter to refer to cryptographic keys, we call log keys "identifiers." We say that a log L' "extends" a log L if (a) L is a prefix of L' and (b) every identifier in L' appears at most once.

Our distributed log uses an authenticated data structure [382, 478, 489] that implements the following five routines:

$\mathsf{Digest}(L) \to d$. Return a constant-size digest d representing the current state of the log.

$\mathsf{ProveIncludes}(L, \mathsf{id}, \mathsf{val}) \to \{\pi_{\mathsf{Inc}}, \perp\}$. Output a proof π_{Inc} that attests to the fact that the identifier-value pair $(\mathsf{id}, \mathsf{val})$ is in the log represented by digest $d = \mathsf{Digest}(L)$.

$\mathsf{DoesInclude}(d, \mathsf{id}, \mathsf{val}, \pi_{\mathsf{Inc}}) \to \{0, 1\}$. Return "1" iff π_{Inc} proves that the log that digest d represents contains $(\mathsf{id}, \mathsf{val})$.

$\mathsf{ProveExtends}(L, L') \rightarrow \{\pi_{\mathsf{Ext}}, \perp\}$. Output a proof π_{Ext} that $d' = \mathsf{Digest}(L')$ represents a log that extends the log that digest $d = \mathsf{Digest}(L)$ represents.

$\mathsf{DoesExtend}(d, d', \pi_{\mathsf{Ext}}) \rightarrow \{0, 1\}$. Return "1" iff π_{Ext} proves that the log that digest d' represents extends the log that digest d represents.

The inclusion and extension proofs must be complete (honest verifiers accept valid proofs) and sound (honest verifiers reject invalid proofs). We define the properties that we need to hold below:

Inclusion completeness. Informally, $\mathsf{DoesInclude}$ accepts valid log-inclusion proofs produced by $\mathsf{ProveIncludes}$. That is, for all logs L, all identifier-value pairs $(\mathsf{id}, \mathsf{val}) \in L$, if $d \leftarrow \mathsf{Digest}(L)$ and $\pi_{\mathsf{Inc}} \leftarrow \mathsf{ProveIncludes}(L, \mathsf{id}, \mathsf{val})$, then $\mathsf{DoesInclude}(d, \mathsf{id}, \mathsf{val}, \pi_{\mathsf{Inc}}) = 1$.

Inclusion soundness. Informally, for all efficient adversaries that output a log L, an identifier-value pair $(\mathsf{id}, \mathsf{val}) \notin L$, and a false inclusion proof π^*_{Inc}, it holds that, for digest $d \leftarrow \mathsf{Digest}(L)$, the probability $\Pr[\mathsf{DoesInclude}(d, \mathsf{id}, \mathsf{val}, \pi^*_{\mathsf{Inc}}) = 1]$ is negligible.

Extension completeness. Informally, $\mathsf{DoesExtend}$ accepts valid log-extension proofs produced by $\mathsf{ProveExtends}$. That is, for all logs L and L', where L' extends L, if $d \leftarrow \mathsf{Digest}(L)$, $d' \leftarrow \mathsf{Digest}(L')$, and $\pi_{\mathsf{Ext}} \leftarrow \mathsf{ProveExtends}(L, L')$, then $\mathsf{DoesExtend}(d, d', \pi_{\mathsf{Ext}}) = 1$.

Extension soundness. Informally, for all efficient adversaries that output a pair of logs L and L' (such that L' does *not* extend L and L does not contain duplicate identifiers), and a false proof π^*_{Ext}, if we compute $d \leftarrow \mathsf{Digest}(L)$ and $d' \leftarrow \mathsf{Digest}(L')$, the probability $\Pr[\mathsf{DoesExtend}(d, d', \pi_{\mathsf{Ext}}) = 1]$ is negligible.

Implementing the data structure. Nissim and Naor [382] show that it is possible to implement these log primitives using only Merkle trees [360]. At a very high level: the digest of the log is just the root of a Merkle tree computed over all of the entries of the log, represented as a binary search tree indexed by id. A log-inclusion proof π_{Inc} is a Merkle proof of inclusion relative to this root. A log-extension proof π_{Ext} is a proof that: (1) every identifier inserted to the new log did not exist in the old log and (2) the new digest represents the old log tree with the new values inserted. It is possible to prove both assertions using a number of Merkle proofs proportional to the number of log insertions.

We now describe how to implement the above routines using a construction as in Nissim and Naor [382]. In the following discussion, we define the "log tree" for a log L to be a binary search tree, ordered by identifiers id. Each internal node in the tree—in addition to containing a value—contains the cryptographic hash of its two child nodes, as in a Merkle tree.

$\mathsf{Digest}(L) \rightarrow d$. Construct the log tree for L by inserting the elements of L into a binary-search tree one at a time. (We can use any type of self-balancing binary-search tree here.) As the digest d, output the hash of the root of the log tree.

$\mathsf{ProveIncludes}(L, \mathsf{id}, \mathsf{val}) \rightarrow \{\pi_{\mathsf{Inc}}, \perp\}$. If $(\mathsf{id}, \mathsf{val}) \notin L$, output $\perp$. Otherwise, output the Merkle inclusion proof that proves that $(\mathsf{id}, \mathsf{val})$ is in the log tree rooted at $d = \mathsf{Digest}(L)$.

$\mathsf{DoesInclude}(d, \mathsf{id}, \mathsf{val}, \pi_{\mathsf{Inc}}) \rightarrow \{0, 1\}$. Treating the digest d as a log-tree root and π_{Inc} as a Merkle proof of inclusion, verify that $(\mathsf{id}, \mathsf{val})$ is included in the log tree rooted at d.

$\mathsf{ProveExtends}(L, L') \rightarrow \{\pi_{\mathsf{Ext}}, \perp\}$. We show how the routine works in the special case in which L' contains exactly one entry $(\mathsf{id}, \mathsf{val})$ that does not appear in L. To generalize to the case in which

there are many new entries in L', we run this routine once for each new entry and output the concatenation of all of the resulting proofs.

If L' does not extend L, output $\perp$. Otherwise, find the identifiers $\mathsf{id}_{\mathsf{left}}$ and $\mathsf{id}_{\mathsf{right}}$ that appear just before and after id in the lexicographical ordering of identifiers in the old log L. Let their corresponding values be $\mathsf{val}_{\mathsf{left}}$ and $\mathsf{val}_{\mathsf{right}}$.

The first portion of the proof π_{Ext} is a Merkle proof of inclusion of $(\mathsf{id}_{\mathsf{left}}, \mathsf{val}_{\mathsf{left}})$ and $(\mathsf{id}_{\mathsf{right}}, \mathsf{val}_{\mathsf{right}})$ in the old log tree rooted at digest $d = \mathsf{Digest}(L)$. These proofs prove that the new identifier id is not in the log represented by the old digest d.

Next, insert $(\mathsf{id}, \mathsf{val})$ into the log tree for L to get a log tree for L' and its corresponding digest. The second portion of the proof π_{Ext} is a Merkle proof of inclusion of every node in the log tree for L' that does not appear in the log tree for L. These proofs prove that the new digest d' represents the root of L's log tree, with the new pair $(\mathsf{id}, \mathsf{val})$ inserted.

$\mathsf{DoesExtend}(d, d', \pi_{\mathsf{Ext}}) \to \{0, 1\}$. Parse π_{Ext} as a series of Merkle inclusion proofs over log trees. Check that id lies between $\mathsf{id}_{\mathsf{left}}$ and $\mathsf{id}_{\mathsf{right}}$ in lexicographic order. Verify each of the Merkle proofs generated in $\mathsf{ProveExtends}$. Checking the Merkle proofs in this case also requires verifying that the nodes in log trees satisfy the proper ordering constraints of a binary search tree: the left child's value is less than the parent's value and the right child's value is greater than the parent's value. Accept if all of these proofs accept.

Security properties. The completeness properties are immediate. Inclusion soundness follows directly from the analysis of Merkle proofs, which in turn rely on the collision-resistance of the underlying hash function. To sketch the argument for extension soundness: The first part of the proof π_{Ext} convinces the verifier that the new identifier id does not appear in the log L represented by the old digest d. Therefore, the log L' that digest d' represents must not contain any duplicate identifiers, since L contains no duplicate identifiers. The second part of the proof π_{Ext} convinces the verifier that log tree rooted at d' is identical to the log tree rooted at d, except with the pair $(\mathsf{id}, \mathsf{val})$ added.

5.6.2 Building a distributed log

We now explain how to use the primitives of Section 5.6.1 to build our distributed append-only log.

Initializing the log. The service provider maintains the entire state of the log L. Each HSM stores a log digest d which, in steady state, is the digest of the log L that the service provider holds. Initially, the log L is empty and each HSM holds the digest of the empty log.

Inserting into the log. A client can insert an entry $(\mathsf{id}, \mathsf{val})$ into the log by simply sending the pair to the service provider. The service provider adds this entry to its log state L.

Proving log membership to HSMs. Before the HSMs allow a client to begin the recovery process, the HSMs require proof that the client's recovery attempt is logged. Assume for the moment that the service provider holds a log L and all HSMs hold the up-to-date digest $d = \mathsf{Digest}(L)$. (We will explain how the HSMs get the latest log digest in a moment.) Then, a client can prove inclusion of any pair $(\mathsf{id}, \mathsf{val})$ in the log by asking the service provider for an inclusion proof. The service provider computes $\pi_{\mathsf{Inc}} = \mathsf{ProveIncludes}(L, \mathsf{id}, \mathsf{val})$ and returns the inclusion proof to the client. The client then sends $(\mathsf{id}, \mathsf{val}, \pi_{\mathsf{Inc}})$ to the HSM, which can check $\mathsf{DoesInclude}(d, \mathsf{id}, \mathsf{val}, \pi_{\mathsf{Inc}})$ to be convinced

that $(\mathsf{id}, \mathsf{val})$ is in the log represented by its digest d. This inclusion check is fast—logarithmic in the log length.

Updating the log digest at the HSMs. After a sequence of log-insertions, the service provider holds a log state L'. The HSMs will be holding a digest $d = \mathsf{Digest}(L)$ of a stale log L. If the service provider is honest, the new log L' extends the old log L.

To update the log digest at the HSMs, the service provider will first send the new digest $d' = \mathsf{Digest}(L')$ to every HSM. Next, the data center must convince each HSM that this new digest d' represents a log that extends the log L that the old digest d represents.

One non-scalable way to achieve this would be for the service provider to send an extension proof $\pi_{\mathsf{Ext}} = \mathsf{ProveExtends}(L, L')$ to every HSM. The problem is that the time required to check this extension proof grows linearly with the number of new log entries. So if every HSMs checked the entire extension proof, the throughput of the system would not increase as the number of HSMs increases.

Instead, we use a randomized-checking approach, as in Figure 5. If there have been I insertions to the log since the last update, the service provider divides the updates into N chunks, each containing I/N insertions. The service provider then applies these chunks of updates to the old log L one at a time, producing a digest d_i and extension proof π_i for each of the N intermediate logs ($i \in \{1, \dots, N\}$). The service provider then sends the root R of a Merkle-tree commitment to these digests to each HSM.

Each HSM then asks the service provider for a random λ-size subset of the intermediate digests and extension proofs, where λ is a security parameter. The service provider returns the requested digests and extension proofs and proves that these values are included in the Merkle root R. Each HSM checks its requested intermediate extension proofs using $\mathsf{DoesExtend}(\cdot)$ and checks the Merkle proof relative to the root R. The HSMs auditing the first and last chunks also ensure that the intermediate digests match the old digest d and the new digest d', respectively.

If these extension and Merkle proofs are valid, each HSM signs the tuple (d, d', R) using an aggregate signature scheme [69], and returns the signature to the service provider. Once all online HSMs have signed, the service provider aggregates these signatures and broadcasts the aggregated signature to all HSMs. If any HSM fails during this process, the service provider notifies the HSMs and they restart this log-update process. (We later describe how the log can make progress even if HSMs fail during the log-update protocol.)

The HSMs check the aggregate signature on (d, d', R) relative to the HSMs' aggregate public key. If the signature is valid, the HSMs accept the new digest d'.

Security. If there are at most f_{secret} compromised HSMs, then even if f_{secret} honest HSMs are slow, $(1-2f_{\mathsf{secret}})N$ honest HSMs will participate in any successful protocol execution. If each of these HSM audits C chunks, then the probability that no honest HSM audits a particular log chunk is

$$\Pr[\text{fail}] = \left(1 - \tfrac{1}{N}\right)^{(1-2f_{\mathsf{secret}})N \cdot C} \le \exp\left((2f_{\mathsf{secret}} - 1) \cdot C\right).$$

(Here, we use the fact that $(1-x) \le \exp(-x)$.) If each HSM audits $C = \lambda \approx 128$ chunks, this failure probability is $\ll 2^{-128}$. In other words, some honest HSM will catch a cheating service provider with overwhelming probability. In addition, since all honest HSMs will expect a signature from all

honest HSMs, this will cause the updating operation to fail and the system to halt. For this analysis, we assume that the adversary cannot adaptively compromise HSMs while the recovery protocol is running without taking them offline.

Scalability. Each HSM must check the extension proofs on λ chunks, where each chunk contains a $1/N$ fraction of the total updates in each epoch. Thus each HSM checks a vanishing fraction ($\frac{\lambda}{N}$) of log insertions. Each HSM checks one aggregate signature, which requires time independent of the number of HSMs [69]. Thus, the total work that each HSM performs per epoch decreases as the number of HSMs N increases.

Because we use the log primarily to limit the number of PIN attempts, garbage collection is straightforward. The service provider simply creates a new empty log, effectively resetting the number of PIN attempts for every user (old copies of the log can still be inspected to monitor recovery attempts). To ensure that the service provider does not run garbage collection and clear the state too frequently, each HSM will run garbage collection for a fixed number of times (e.g. the expected number of garbage collections over two years) before refusing to respond to further requests. This bounds the number of times the service provider can garbage collect the log.

Making progress in spite of failures. If frequent node failures prevent the log from making progress, the HSMs can run the following more complicated log-update protocol. In this variant, each HSM chooses which log chunks to audit as a deterministic function of the Merkle root R and the HSM's own node ID.

Provided that we choose the constant C large enough, for *every* choice of the Merkle root, R at least one honest HSM will audit each log chunk. In particular, by taking $C \geq 384$ the probability that no honest HSM audits a particular chunk is less than 2^{-384}. The probability that there exists a 256-bit Merkle root R that causes no honest HSM to audit a particular chunk is then at most $2^{256} \cdot 2^{-384} = 2^{-128}$. So, no matter how the service provider influences the Merkle root R, at least one honest HSM will attempt to audit each chunk.

Using this method, given the Merkle root R, every HSM can deterministically compute the set of log chunks that every other HSM will audit. Then, if any HSM fails during the audit process, the remaining non-failed HSMs can recursively run our randomized-checking protocol to check the log chunks that the failed HSMs would have checked.

In this way, the protocol can make progress even if HSMs fail during the log-update process.

5.6.3 Transparency and external auditability

Our log design allows *anyone* to audit the log to ensure that the service provider correctly maintains the log's append-only property. Additional auditors only add to the security of the system by adding another layer of protection, as they can detect log corruptions in the event that more than f_{secret} HSMs are compromised. In particular, for any two log digests d and d', an auditor can ask the data center for the entire logs L and L' corresponding to both of these digests. The auditor confirms that d is the root of the log tree for L and that d' is the root of the log tree for L'. Finally, the auditor checks that L' extends L.

As an extra precaution, users could specify external parties (e.g., Let's Encrypt) as designated auditors during backup. During recovery, the HSMs would only complete the recovery if these auditors sign the latest log digest. In this way, mounting a brute-force PIN-guessing attempt against a user would require compromising the user's external auditors as well.

The transparency log can also help with PIN re-use. As discussed in Section 5.8, instead of storing the salt directly with the service provider, the salt itself can be encrypted using a second round of location-hiding encryption and a null PIN. After recovery, the salt will be destroyed as discussed in the next section. Once the salt has been destroyed, the device restoring a backup can use the log to determine if anyone else has ever fetched the salt. If not, then it is safe for the user to re-use the old PIN.

As described in Section 5.4, the log contains usernames, which could be sensitive. To prevent leaking usernames, we would replace usernames with random device identifiers that are rerandomized when the device is factory reset. However, even with this modification, the log still leaks information about when and how often users restore backups, which the service provider may not wish to make public. While we hope that organizations would make their logs public, we acknowledge that some may only share their logs with several hand-picked organizations for auditing or may not share their logs at all. In these cases, our security guarantees still hold, although some of the transparency benefits are lost.

5.7 Forward security by puncturable encryption

We would like our encrypted-backup system to provide forward secrecy [91]. During the recovery process, the client reveals the identity of the $n \ll N$ HSMs that can decrypt its backup. Without forward secrecy, an attacker can break into these n HSMs to recover the client's backed-up data. Forward secrecy ensures that after recovery, an attacker, even one who compromises all HSMs in the data center, learns no information about the client's backup.

One seemingly straightforward way to provide forward secrecy would be to use a new keypair for each backup. However, because the client cannot interact with the HSMs it is encrypting to during backup (as this would reveal their identities), using a unique keypair for every backup would require every HSM in the data center to generate a new keypair for every backup, running counter to our scalability goals.

5.7.1 Background: Puncturable encryption

We instead achieve forward secrecy using puncturable public-key encryption [93, 116, 152, 153, 215, 234]. A puncturable encryption scheme is a normal public-key encryption scheme $(\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$, with one extra routine:

$\mathsf{Puncture}(\mathsf{sk}, \mathsf{ct}) \to \mathsf{sk}_{\mathsf{ct}}$. Given a decryption key sk and a ciphertext ct, output a new secret key $\mathsf{sk}_{\mathsf{ct}}$ that can decrypt all ciphertexts that sk could decrypt except for ct.

Puncturable encryption for forward secrecy. To achieve forward security in SafetyPin, after an HSM decrypts its share of a client's recovery ciphertext ct, the HSM *punctures* its secret decryption

key. The punctured key allows the HSM to decrypt all ciphertexts except for ct. Thus, if an attacker compromises *all* HSMs in the data center after a client has recovered its backup, the attacker will be unable to decrypt any backup images that clients have already recovered. Furthermore, if an attacker compromises at most $f_{\mathsf{secret}} \cdot N$ HSMs total, where f_{secret} is a parameter of the system that we define in Section 5.3, then the attacker will not be able to recover any backed-up data whatsoever.

Existing tool: Bloom-filter encryption. Our implementation uses a puncturable encryption scheme called Bloom-filter encryption [152]. There are only two details of Bloom-filter encryption that are important for this discussion.

1. *The secret key is large.* If a key supports $P \in \mathbb{Z}^{>0}$ punctures and we want decryption to fail with probability at most $2^{-\lambda}$, then the secret key for Bloom-filter encryption is an array of roughly λP elements of a cryptographic group $\mathbb{G}$. After P punctures, the secret key may no longer decrypt messages and it is necessary to rotate encryption keys.
2. *Puncturing is simple.* Puncturing the secret key just requires deleting λ elements in the data array that comprises the secret key.

Concretely, when we set the Bloom-filter-encryption parameters to suitable values for experimental evaluation, each Bloom-filter encryption secret key has size over 64 MB. Even high-end HSMs have only 1–2 MB of storage (Table 2), so storing such large keys on an HSM would be impossible.

5.7.2 Outsourced storage with secure deletion

We describe how to efficiently outsource the storage of this large secret key in a way that preserves forward secrecy of the punctured key. In particular, the HSM can outsource the storage of its secret-key array to the untrustworthy service provider, while still retaining the ability to delete portions of the key. To do this, we draw on techniques for outsourcing the storage of any data array (not just secret keys) first described by Di Crescenzo et al. [157] and extended in subsequent work [429, 437].

Desired functionality. At a high level, the HSM has access to (a) a small amount of internal storage and (b) a large external block store, run by the service provider. The HSM wants to store an array of D data blocks at the provider $(\mathsf{data}_1, \dots, \mathsf{data}_D)$. The HSM should be able to subsequently *read* or *delete* these blocks.

The following security properties should hold, even if the attacker, controlling the service provider, may choose the data-array and sequence of operations the HSM performs:

- **Integrity.** If the service provider tampers with the stored data in a way that could cause a *read* to return an incorrect result, the read operation outputs $\perp$. Otherwise, the read operation for a block i returns the value of the last data that the client wrote to block i.
- **Secure deletion.** If the service provider compromises the HSM after the HSM has run the *delete* operation for the ith data block, the attacker learns nothing about the data stored in block i. (This property implies a confidentiality property: the service provider learns nothing about the outsourced data.)

For efficiency, the HSM storage requirements must be small (constant size) and the *read* and *delete* routines should run quickly (in time logarithmic in the size D of the data array). Unlike in

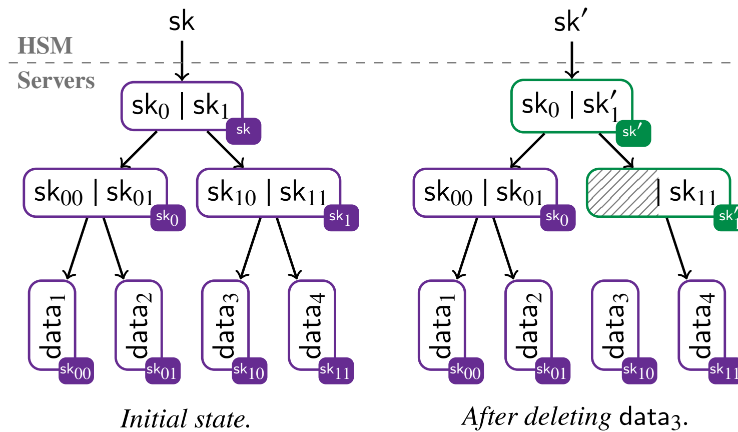


Figure 6: The outsourced-storage scheme has a tree of keys. An arrow $a \rightarrow b$ denotes that value b is stored encrypted under key a. A service provider that stores all values it sees and later compromises the HSM state (sk') still does not learn the deleted data_3 value.

ORAM [204, 206], our goal is not to hide the HSM's data-access pattern from the service provider. We aim only to hide the contents of the array.

5.7.3 Building secure outsourced storage

We explain the construction here in prose, drawing on techniques first described by Di Crescenzo et al. [157].

Running the setup phase. During the setup phase, the outsourced-storage scheme builds a binary tree with D leaves. Every node of the tree contains a fresh symmetric encryption key. During setup, for each node in the tree with key sk_i, we encrypt the keys of the child nodes sk_{i0} and sk_{i1} with sk_i and store this ciphertext $\mathsf{AE.Encrypt}(\mathsf{sk}_i, \mathsf{sk}_{i0} \| \mathsf{sk}_{i1})$ in outsourced storage. At the leaves of the tree, we encrypt the ith data block with the key sk_i at the ith leaf and we store the ciphertext $\mathsf{AE.Encrypt}(\mathsf{sk}_i, \mathsf{data}_i)$ in outsourced storage.

For example, in Figure 6, we use sk_0 to encrypt sk_{00} and sk_{01} and we store the result in outsourced storage. We use key sk_{01} to decrypt data item 2. Thus, knowing the root key sk is enough to decrypt the entire tree and access every data element in the array.

Reading a data block. To retrieve the data block at index i, the HSM reads in the ciphertexts along the path from the tree root to leaf i. The HSM then decrypts the chain of ciphertexts from the root down to recover the data block at index i. For example, in Figure 6, to retrieve data block 3, the HSM can use sk to decrypt sk_1, and sk_1 to decrypt sk_{10}, which it can use to decrypt data item 1.

Deleting a data block. To delete the data block at index i, the HSM recovers (as in retrieval) the keys along the path from the root to leaf i. At the node containing the key to decrypt data block i, the HSM deletes the key. It then chooses a fresh key and re-encrypts the other key at that node using the fresh key. To maintain the ability of the parent key to decrypt the child ciphertext, the HSM updates the parent of that node to contain the fresh key for its child and re-encrypts the parent's keys under a

new key. It continues this up the path to the root, where the HSM chooses a new key sk' to encrypt the root. The HSM replaces sk with sk', deleting the old sk, and then sends the new ciphertexts along the path from the root to leaf i back to the service provider. For example, in Figure 6, to delete data item 3, the HSM decrypts the keys $(\mathsf{sk}_0 \| \mathsf{sk}_1)$ and $(\mathsf{sk}_{10} \| \mathsf{sk}_{11})$. The HSM then deletes sk_{10}, chooses a new key sk'_1 to encrypt sk_{11}, and then chooses a new key sk' to encrypt sk_0 and sk'_1. The HSM then replaces sk with sk'.

Efficiency. The setup time is linear in the size of the data array D. The runtimes of retrieval and deletion are both logarithmic in D, and require only symmetric-key operations. The HSM stores only the constant-sized root encryption key sk.

Security intuition. An HSM can always recover the keys necessary to decrypt a data item, provided the HSM did not previously delete any of the keys necessary for decryption. Integrity follows immediately from the security of the underlying authenticated encryption scheme. Finally, we ensure secure deletion by deleting the key necessary to decrypt a certain data item and updating the root key. Without the old root key, it is impossible to access the key necessary to decrypt the deleted data item.

Putting it together. To summarize: the HSMs use a puncturable encryption scheme to prevent the compromise of HSM secrets at time T from allowing an adversary to learn about backed-up data that was recovered any time before T. We implement puncturable encryption using Bloom-filter encryption and outsource the storage of the large secret decryption key while allowing secure deletion.

5.8 Extensions and deployment considerations

A real-world SafetyPin implementation has to deal with a number of additional issues, which we discuss now.

Failure during recovery. As discussed in Section 5.7, after participating in recovery, HSMs revoke their ability to decrypt the recovered ciphertext. One consequence is that a client cannot recover the same backup ciphertext twice. This raises the question of what happens if a replacement device fails during or shortly after recovery, or if a communication failure during recovery prevents the new device from receiving the replies from the HSMs.

To solve this problem, when a client initiates recovery, it first generates a fresh per-recovery keypair $(\mathsf{sk}, \mathsf{pk})$ for a public-key encryption scheme. The client backs up this secret key sk using SafetyPin before initiating its recovery. Next, the client sends the public key pk to each HSM and then begins the backup-recovery process. Each HSM encrypts its replies to the client under pk, and each HSM sends a copy of each reply to the data center. If a client device fails during recovery, a second, replacement client device can retrieve the backed-up secret key sk and use these to decrypt the replies stored at the data center. This scheme nests arbitrarily, thereby handling any number of consecutive device failures during recovery.

Incremental backups. In practice, mobile devices often generate incremental backups rather than encrypting the entire disk image for each backup. SafetyPin supports incremental backups in the

Operation	Ops/sec
Pairing	0.43
ECDSA ver	5.85
ElGamal dec	6.67
$g^x \in \mathbb{G}_{\text{P256}}$	7.69

	Operation	Ops/sec
	HMAC-SHA256	2,173.91
	AES-128	3,703.70
I/O	RTT, HID (32b)	71.43
I/O	RTT, CDC (32b)	2,277.90
I/O	Flash read (32b)	≈166,000

Table 7: Microbenchmarks on SoloKey. Pairing is on BLS12-381 curve using the JEDI library [299]. Other public-key operations use NIST P256 curve.

following way. The user uses SafetyPin to store a single AES key, which the user also keeps on her phone. The user can then encrypt incremental backups under this AES key and upload the resulting ciphertext to the data center. When the user recovers, she recovers her AES key and can use this key to decrypt the incremental updates.

Multiple recovery ciphertexts. Clients back up their phones regularly (e.g., every three days), and will thus generate a series of recovery ciphertexts. We want to ensure that after a client recovers her backup from time t, the HSMs involved in recovery puncture their secret decryption keys so that they cannot decrypt that client's backups from earlier times $t' < t$, even if an attacker compromises all HSMs in the data center. To achieve this, in the puncturable-encryption step (Section 5.7), we have the client use the same salt for each recovery ciphertext it generates. In this way, the client will encrypt its series of backups to the same set of HSMs. When these HSM puncture their secret keys during the recovery process, they will destroy their ability to decrypt any previous recovery ciphertexts from the given client. After recovery, the client chooses a new salt to generate subsequent backups on its new device.

Preventing post-recovery PIN leakage. As we have discussed, an attacker that watches the client recover can learn a salted hash of the user's PIN, which can be used to mount an offline brute-force attack to learn the user's PIN.

One approach to protect against this attack would be to have each user store their salt in secret-shared form at a random set S_{salt} of HSMs, where S_{salt} is included in the client's recovery ciphertext. Then, provided that the attacker does not compromise this set of HSMs, the attacker would learn no useful information on the user's PIN, even after recovery. An attacker could always compromise every HSM in S_{salt}, but an attacker that can compromise only a f_{secret} fraction of HSMs in the data center would not be able to mount this attack against too many clients' salts. We hope to model and prove this multi-user PIN-protection property in future work.

5.9 Implementation and evaluation

We implemented SafetyPin on an experimental data cluster of 100 hardware security devices (Figure 1). Our prototype does not include all features necessary for a real-world deployment, including those described in Section 5.8.

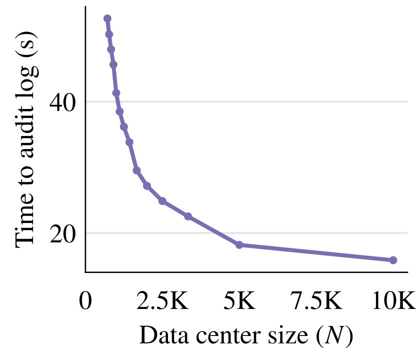


Figure 8: Log-audit time after inserting 10K recovery attempts for a log with roughly 100M recovery attempts. We only measure the auditing time for 100 HSMs as we only had 100 SoloKeys; we distribute the work as if there were N HSMs.

HSM. For the HSMs, we used SoloKeys [463], a low-cost open-source USB FIDO2 security key. SoloKeys use a STM32L432 microcontroller with an ARM Cortex-M4 32-bit RISC core clocked at 80MHz and 265KB of memory. The device is not side-channel resistant, but has a true random number generator and can lock its firmware. We add roughly 2,500 lines of C code to the open-source SoloKey firmware [462].

By default, SoloKeys communicate with the USB host via USB HID, an interrupt-based USB class used typically for keyboards and mice that has a maximum throughput of 64KBps. To improve performance, we rewrote parts of the firmware to use USB CDC, a high-throughput USB class commonly used for networking devices. This gave a roughly 32× increase in I/O throughput (Table 7).

For the puncturable-encryption scheme (Section 5.7.1), we use a variant of Bloom-filter encryption [152] that avoids the need for pairings [68] but increases the size of the HSMs' public keys. For the aggregate signature scheme needed for the log, we use BLS-style multisignatures [67] over the JEDI [299] implementation of the BLS12-381 curve.

Our implementation does not encrypt communication between the client and HSMs. Based on the time to run AES-128 and ElGamal encryption on the SoloKeys, we estimate that transport-layer encryption would add two ElGamal decryptions and 2KB of AES operations per recovery, increasing recovery time by approximately 0.3 seconds, or 30%. This overhead is comparatively high because processing a recovery only requires a handful of symmetric and public key operations.

Service Provider. Our service provider host is a Linux machine with an Intel Xeon E5-2650 CPU clocked at 2.60GHz. Our service-provider implementation is roughly 3,800 lines of C/C++ code (excluding tests) and uses OpenSSL.

Client. Our client device is a Google Pixel 4. Our implementation is roughly 2,300 lines of C/C++ code (excluding tests) and uses OpenSSL.

5.9.1 Microbenchmarks

Log. Figure 8 demonstrates how increasing the number of HSMs reduces the log-digest update time. We assume that the log is periodically garbage collected (i.e., approximately once a month), so that

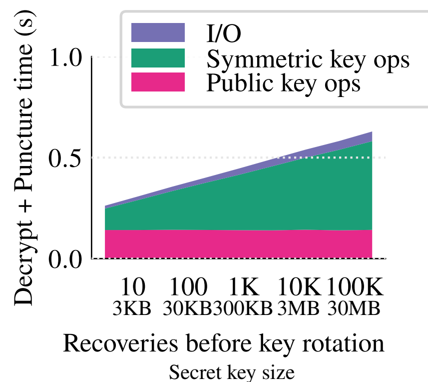


Figure 9: Time to run puncturable encryption on a single HSM as the maximum number of allowed punctures (and also secret key size) grows. The cost of our outsourced storage scheme dominates, though the access time is logarithmic in the size of the key.

it holds at most a hundred million recovery attempts at once. If the HSMs run the log-update process every 10 minutes, each HSM spends approximately 11% of its active cycles auditing the log. The choice of how often to update the log is a tradeoff between how long users must wait to recover their backups and the total number of write cycles to non-volatile storage permitted by the hardware.

Puncturable encryption. Figure 9 shows the cost of performing a decrypt-and-puncture operation as the number of supported punctures increases. The AES operations associated with our scheme for outsourced storage with secure deletion (Section 5.7.2) dominate the cost.

Another way to implement outsourced storage with secure deletion would be to have the HSM store the outsourced array encrypted under a single AES key k. To delete an item, the HSM would read in the entire array, delete the item, and write out the entire array encrypted under a fresh key k'. With this approach, a deletion takes 48 minutes for a 64 MB array (the size of our outsourced secret keys). Our scheme thus improves system throughput by roughly $4,423\times$.

Each HSM punctures its secret key (Section 5.7.1) once after each decryption it performs. Since our puncturable-encryption scheme only supports a fixed number of punctures, each HSM must periodically rotate its encryption keys. We configure our puncturable-encryption scheme to allow each HSM to perform roughly 2^{18} decryptions before it must rotate its keys (rotation is triggered when half of the elements of the secret key have been deleted). Key rotation is expensive: we estimate (based on the number of public-key operations required) that key rotation on our HSMs will take roughly 75 hours. Each HSM spends approximately 139.4 hours processing recoveries and maintaining the log between key rotations. Therefore, each HSM spends roughly 56% of its cycles rotating its keys, and each HSM can process 1,503.9 recoveries per hour on average.

5.9.2 End-to-end costs

Parameters. We estimate that on average, each user will run recovery once a year. (There are 3.8B smartphone users [467] and 1.5B smartphones sold annually [468], so we expect $1.5/3.8 = 0.39 \ll 1$

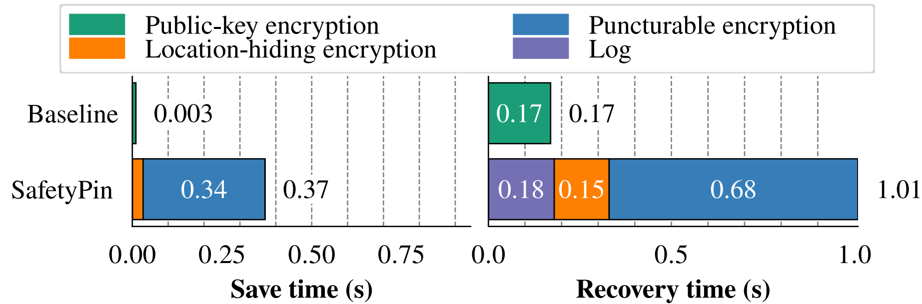


Figure 10: Breakdown of time to save (on Android Pixel 4 phone) and recover (using our SoloKey cluster). We do not consider the time to encrypt or decrypt disk images.

recovery/user/year.) We calculate that a SafetyPin deployment of $N = 3,100$ HSMs could support one billion users. So, we treat our small cluster of 100 HSMs as a representative slice of a larger data center of $N = 3,100$ HSMs. Within this larger data center, each client shares its recovery keys among a cluster of $n = 40$ HSMs. This choice of n is based on the size of the data center N and PINs with six decimal digits, and is dictated by Theorem 22. We set the puncturable encryption keys to allow 2^{20} punctures, as we found this provides a reasonable tradeoff between the time to decrypt and puncture and the time between key rotations. With these parameters, we maintain secrecy if at most an $f_{\text{secret}} = \frac{1}{16}$ fraction of the HSMs are compromised (or $f_{\text{secret}} \cdot N \approx 194$ total). We allow data recovery if at most an $f_{\text{live}} = \frac{1}{64}$ fraction fail due to benign hardware failures (or $f_{\text{live}} \cdot N \approx 48$ total).

Baseline. We compare against an encrypted-backup system modeled on the ones that Google and Apple use [296, 507]. To backup, the client selects a fixed cluster of five HSMs and encrypts her recovery key and a hash of her PIN under the cluster's public key. At recovery, the client sends the recovery ciphertext and a hash of her PIN to the cluster, and any HSM in the cluster can decrypt the ciphertext, check that the PIN hashes match, and return the recovery key. To defeat brute-force PIN-guessing attacks, each HSM independently limits the number of recovery attempts allowed on a given ciphertext.

Client overhead. Figure 10 gives the overhead of generating a backup in SafetyPin, compared to the baseline. The backup process takes 0.37 seconds. SafetyPin recovery ciphertexts are 16.5KB, versus 130B for our baseline, though we expect encrypted disk image to dominate the ciphertext size.

SafetyPin increases the bandwidth cost at the client. In the baseline scheme, the client downloads five public keys—one from each of its five chosen HSMs. In SafetyPin, the client must fetch a copy of all HSMs' public keys. (This way, the service provider does not learn the subset of HSMs to which the client is encrypting its backup.) So, when a client first joins the system, the client must download all these keys (11.5MB). Whenever an HSM rotates its puncturable-encryption keys, clients must download the HSM's new public key. In a deployment of $N = 3,100$ HSMs supporting one billion recoveries annually, we estimate that each SafetyPin client must download 1.97MB of keying material daily. Increasing the puncturable encryption failure probability would decrease client bandwidth, although this would require decreasing the fraction of HSMs allowed to fail, f_{live}. If a client goes offline for several days, it must download the rotated public keys for each day it spent offline (roughly 2MB/day), up to a maximum of 11.5MB (the size of all HSMs' keys). However, the

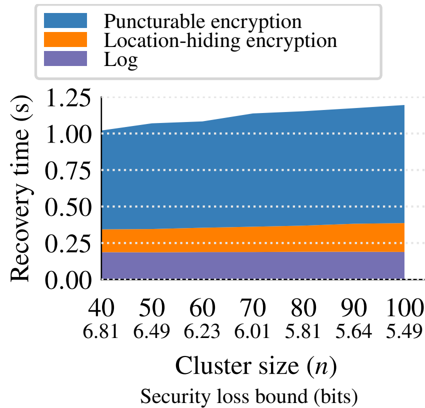


Figure 11: Recovery time grows slowly as cluster size n increases. The bits of security lost refers to the difference between the advantage of an attacker against a SafetyPin deployment with the given value of n and an attacker trying to guess the user's PIN.

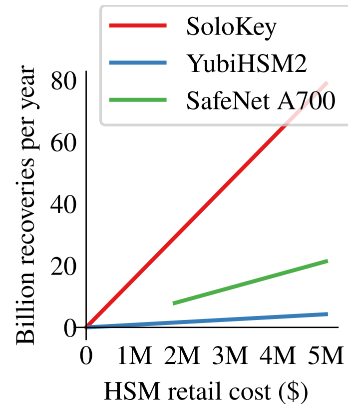


Figure 12: Estimated number of SafetyPin-protected recoveries per year supported by clusters of different HSM models and costs. We use g^x/sec to compute the expected throughput of more powerful HSMs based on our measurements using SoloKeys (Table 2).

client only needs to store the public keys for the n HSMs comprising its chosen recovery cluster which amounts to 9.02KB.

Recovery time. At a cluster size of $n = 40$ HSMs, Figure 11 shows that the end-to-end recovery time takes 1.01 seconds. Puncturable-encryption operations dominate recovery time (Figure 10), since these require expensive elliptic-curve operations for ElGamal decryption and many I/O and AES operations in order to perform secure deletion (Section 5.7.2.

Tail latency. In a deployment of SafetyPin, it will be important to consider not only the average throughput of the SafetyPin cluster, but also the request latency. Since recovery requests will arrive concurrently and in a bursty fashion, we will need to overprovision the system slightly to ensure that request tail latency does not grow too high, even under large transient loads. In Figure 13, we model how many HSMs are required to achieve various 99th-percentile latencies, while handling different average throughputs. We compute these values by modeling incoming requests using a Poisson process and each HSM using a M/M/1 queue with service times derived from our experimental results. As the figure demonstrates, by increasing the total number of HSMs, we can reduce the tail latency even when accounting for request contention. We anticipate that recovery time will in practice be dominated by the time to download the encrypted disk image, and so as long as the tail latency is less than or close to this time, any delay is unlikely to be noticed by the user.

Financial cost. Figure 12 shows how throughput scales as the outlay on HSMs increases and Table 14 presents dollar-cost estimates for SafetyPin deployments with different types of HSMs. For a configuration that tolerates the compromise of 50 high-quality HSMs, we estimate that adding SafetyPin to an unencrypted backup system would increase the system's dollar cost by 2.5%.

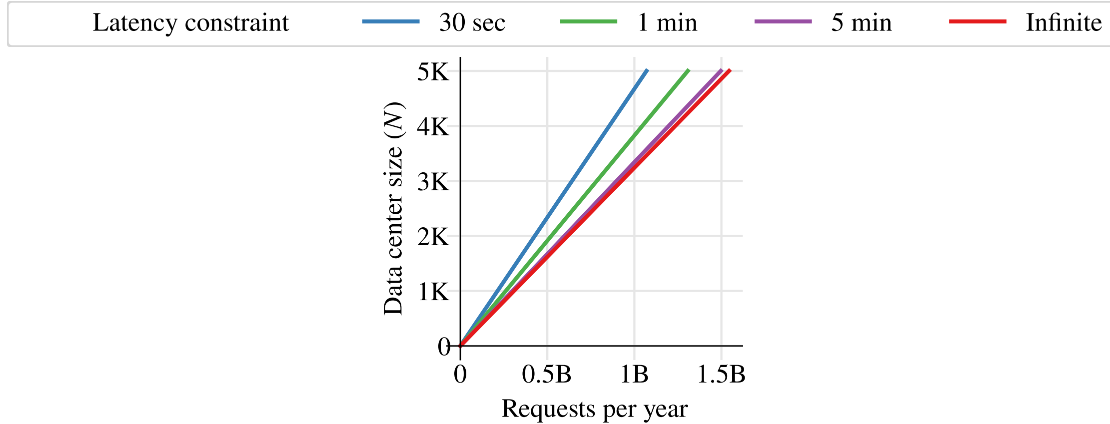


Figure 13: Data center sizes necessary to process different request rates with various 99th-percentile latency requirements.

	HSM Qty.	f_{secret}	N_{evil}	Cost
SoloKey [463]	3,037	1/16	189	$60.7K
YubiHSM2 [530]	1,732	1/16	108	$1.1M
SafeNet A700 [441]	40	1/20	2	$738.7K
– 10 evil HSMs	320	1/32	10	$3.0M
– 50 evil HSMs	800	1/16	50	$14.8M
Estimated cost of storing 4GB × 10^9 users per year:				$600M

Table 14: The estimated hardware cost of a SafetyPin deployment supporting one billion users, if each user recovers once per year. The N_{evil} number is how many corrupt HSMs the deployment tolerates.

We estimate the storage cost using AWS S3 infrequent access [23] ($0.0125 per GB/month). We estimate YubiHSM2 and SafeNet HSM throughput using their data sheets (Table 2). When computing the number of HSMs necessary to service a billion users, we account for key-rotation time. A cluster of 40 SafeNet HSMs can meet the throughput demands of one billion users, so we also consider larger deployments tolerating more compromised HSMs.

5.10 Related work

In Section 5.10.1, we describe related work from before the publication of the original paper [137], and in Section 5.10.1, we include subsequent related work.

5.10.1 Related work before publication

Today's encrypted-backup systems rely either on the security of hardware security modules [214,296], secure microcontrollers [11], or secure enclaves [337, 354]. Vulnerabilities in these hardware components leave encrypted-backup systems open to attack. And there is ample evidence of vulnerabilities in both HSMs [10, 85, 190, 244, 282, 364, 377, 403] and enclaves [64, 81, 109, 169,

213, 238, 313, 316, 372, 495, 497, 498], and reason for concern about hardware backdoors as well [49, 298, 483, 524].

Many companies including Anchorage [26], Unbound Tech [494], Curv [130], and Ledger Vault [310], offer systems for secret-sharing cryptocurrency secret keys across multiple hardware devices. Unlike SafetyPin, these solutions use a small fixed set of HSMs, so they cannot simultaneously provide scalability and protection against adaptive HSM compromise.

Mavroudis et al. [350] propose building a single trustworthy hardware security module from a large array of potentially faulty hardware devices. To achieve this, they use cryptographic protocols for threshold key-generation, decryption, and signing. Like Myst, SafetyPin distributes trust over a large number of hardware devices. Unlike Myst, SafetyPin focuses on hardware-protected PIN-based encrypted backups, rather than more traditional HSM operations, such as decryption and signing.

In recent theoretical work, Benhamouda et al. show how to scalably store secrets on proof-of-stake blockchains when an adversary can adaptively corrupt some fraction of the stake [57]. They face many of the same cryptographic challenges that we tackle in Section 5.5; their theoretical treatment complements our implementation-focused approach. While they use proactive secret sharing to periodically re-share the secret and hide the secret from an adversary controlling some fraction of the stake, our approach allows a party with some low-entropy secret to recover the high-entropy secret.

Di Crescenzo et al. show how to use a small amount of erasable memory to outsource the storage of a much larger data array in non-erasable memory while providing secure deletion [157]. Their construction uses a tree-based approach where reading, writing, or deleting an element requires a number of symmetric key operations logarithmic in the size of the data array. Subsequent work has applied similar techniques to B-trees [429] and dynamically sized arrays [437]. These works are part of a larger body of work on secure deletion using cryptography [43, 62, 70, 195, 404, 405, 428, 482].

Transparency logs inspire our log design [9, 28, 309, 322, 355]. Enhanced certificate transparency proposes a randomized checking technique to split the work of checking the correspondence between a Merkle tree ordered lexicographically and another ordered by time [440], which is similar in spirit to the way that we split verification across HSMs. The proofs we provide to the HSMs about the state of the log draw on work on authenticated data structures [347, 397, 478] and cryptocurrency light clients [373]. Kaptchuk et al. show how public ledgers can be used to build stateful systems from stateless secure hardware [277], and they show how their techniques can be applied to Apple's encrypted-backup system. This work is complementary to ours, as they show how to securely manage state in cases where HSMs do not have secure internal non-volatile storage (an assumption we make in SafetyPin).

5.10.2 Subsequent related work

Since publication, WhatsApp has also proposed a solution for encrypted backups based on HSMs [515]. Gareth et al. analyzed the security of this protocol [144].

Subsequent work on encrypted backups has shown how to provide security guarantees similar in spirit to those of SafetyPin: secure hardware should not be a single point of security failure. We observed that secret key backups could benefit from splitting trust across different trust domains,

which could be instantiated as different types of secure hardware in different clouds [138]. Signal has begun deployment of a new backup architecture, SVR3, that splits trust across different types of enclaves and cloud providers in order to provide privacy in the event that an attacker compromises a subset of the enclave types [119]. Juicebox allows users to back up their keys by splitting trust across "hardware realms" (HSMs) and "software realms" (managed by different organizations) [491].

Orsini et al. develop a key recovery systems that does not require the user to remember a PIN. However, it requires clients to refute recovery attempts initiated by an attacker, and so the client needs to come online periodically for security [390].

Acsesor also addresses the problem that it is challenging for users to manage cryptographic keys. Acsesor helps users manage their keys by splitting trust across some number of "guardians" that are chosen by the user and enact policies [105].

Chen et al. build a system for blind cloud data with the goal of protecting data from cloud providers while still allowing clients to access their data via a passphrase. Their solution splits trust between an application server and a storage server, but they do not provide protection against a compromised application provider that can access both the application server and storage.

Little, Qin, and Varia examine the problem of account recovery in the event that the service provider does not have identifying information for any users (e.g., an email address for verification) [327]. Their solution does not use secure hardware, but splits trust across multiple servers.

Fábrega et al. observed that end-to-end encrypted backups can be vulnerable to injection attacks, as backup size can leak information [171]. A real-world SafetyPin deployment would need to consider defenses against this class of attacks.

5.11 Security analysis

Additional notation. We use $x \leftarrow 7$ to indicate assignment. For a finite set S, we use $x \xleftarrow{R} S$ to denote taking a uniform random sample from S. The notation $\text{poly}(\cdot)$ refers to a fixed polynomial function and $\text{negl}(\cdot)$ refers to a fixed negligible function.

5.11.1 Syntax

We first define the syntax of a location-hiding encryption scheme. The scheme is parameterized by a total number of HSMs N, a cluster size $n \ll N$, a PIN space $\mathcal{P}$, a message space $\mathcal{M}$, and two constants:

(a) the fraction f_{live} of the N HSMs in a cluster whose benign failure we can tolerate while still allowing message recovery, and

(b) the fraction f_{secret} of the N total HSMs whose compromise we can tolerate while still providing security.

In our construction, we take $f_{\mathsf{live}} = \frac{1}{64}$ and $f_{\mathsf{secret}} = \frac{1}{16}$, but any constants $0 < f_{\mathsf{live}}, f_{\mathsf{secret}} < 1$ would suffice with an adjustment to the parameters.

Formally, a location-hiding encryption scheme consists of the following five algorithms:

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$. On input security parameter λ, expressed in unary, output a public-key encryption keypair.

$\mathsf{Encrypt}(\mathsf{pk}_1, \dots, \mathsf{pk}_N, \mathsf{salt}, \mathsf{pin}, \mathsf{msg}) \to \mathsf{ct}$. Given a list of N public keys, generated using KeyGen, a randomizing salt $\mathsf{salt} \in \{0,1\}^*$, a PIN $\mathsf{pin} \in \mathcal{P}$, and a message $\mathsf{msg} \in \mathcal{M}$, output a ciphertext ct.

$\mathsf{Select}(\mathsf{salt}, \mathsf{pin}) \to (i_1, \dots, i_n)$. Given a salt and PIN, output the indices $(i_1, \dots, i_n) \in [N]^n$ of the n secret keys needed to reconstruct the encrypted message.

$\mathsf{Decrypt}(\mathsf{sk}_{i_j}, i_j, \mathsf{ct}) \to \sigma_j$. Given a ciphertext ct, a secret key sk_{i_j} and an index i_j, produce the j-th secret share σ_{i_j} of the plaintext message.

$\mathsf{Reconstruct}(\sigma_1, \dots, \sigma_n) \to \mathsf{msg}$. Given n shares of the plaintext message msg returned by Decrypt, reconstruct msg.

To understand the syntax, it may be helpful for us to explain how we use these routines in our encrypted-backup application:

1. **Setup.** During device provisioning, each HSM runs $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(1^\lambda)$ to generate its keypair $(\mathsf{pk}_i, \mathsf{sk}_i)$. The HSM stores the secret sk_i in its internal memory and it publishes pk_i.
2. **Backup.** To back up its message msg with salt salt and PIN pin, given HSM public keys $(\mathsf{pk}_1, \dots, \mathsf{pk}_N)$ the client runs

$$\mathsf{ct} \leftarrow \mathsf{Encrypt}(\mathsf{pk}_1, \dots, \mathsf{pk}_N, \mathsf{salt}, \mathsf{pin}, \mathsf{msg}).$$

   The client uploads its recovery ciphertext ct to the service provider.
3. **Recovery.** During recovery, the client fetches its salt salt and recovery ciphertext ct from the service provider. It then computes

$$(i_1, \dots, i_n) \leftarrow \mathsf{Select}(\mathsf{salt}, \mathsf{pin})$$

   to identify the cluster of n HSMs it must communicate with during recovery.

   For each HSM $i_j \in \{i_1, \dots, i_n\}$, the client asks HSM i_j to decrypt the ciphertext ct. (Our full protocol requires interaction with the service provider, but we elide those details here.) The HSM, who holds the secret key sk_{i_j}, computes:

$$\sigma_j \leftarrow \mathsf{Decrypt}(\mathsf{sk}_{i_j}, i_j, \mathsf{ct}).$$

   Finally, the client recovers its plaintext as

$$\mathsf{msg} \leftarrow \mathsf{Reconstruct}(\sigma_1, \dots, \sigma_n).$$

5.11.2 Definitions

Intuitively, the correctness property states that if a random f_{live} fraction of the secret keys are unavailable, decryption still succeeds.

Experiment 14 (Location-hiding encryption: Correctness)**.** We define the following correctness experiment, which is parameterized by a location-hiding encryption scheme with parameters $(N, n, \mathcal{P}, \mathcal{M}, f_{\mathsf{live}}, f_{\mathsf{secret}})$, a PIN $\mathsf{pin} \in \mathcal{P}$, a message $\mathsf{msg} \in \mathcal{M}$, and a security parameter $\lambda \in \mathbb{N}$. The experiment consists of the following steps:

$$
\begin{aligned}
(\mathsf{pk}_i, \mathsf{sk}_i) &\leftarrow \mathsf{KeyGen}(1^\lambda), \text{ for } i = 1, \dots, N \\
\mathsf{salt} &\xleftarrow{\mathbb{R}} \{0,1\}^\lambda \\
\mathsf{ct} &\leftarrow \mathsf{Encrypt}(\mathsf{pk}_1, \dots, \mathsf{pk}_N, \mathsf{salt}, \mathsf{pin}, \mathsf{msg}) \\
F &\leftarrow \emptyset \\
&\text{Add each element of } \{1, \dots, N\} \text{ to } F \\
&\text{with independent probability } f_{\mathsf{live}}. \\
\{i_1, \dots, i_n\} &\leftarrow \mathsf{Select}(\mathsf{salt}, \mathsf{pin}) \\
\sigma_j &\leftarrow \begin{cases} \bot & \text{for } j \in F \\ \mathsf{Decrypt}(\mathsf{sk}_{i_j}, i_j, \mathsf{ct}) & \text{otherwise} \end{cases} \\
\mathsf{msg}' &\leftarrow \mathsf{Reconstruct}(\sigma_1, \dots, \sigma_n)
\end{aligned}
$$

The output of the experiment is 1 if $\mathsf{msg} = \mathsf{msg}'$ and 0 otherwise.

Definition 13 (Correctness)**.** Formally, we say that a location-hiding encryption scheme on parameters $(N, n, \mathcal{P}, \mathcal{M}, f_{\mathsf{live}}, f_{\mathsf{secret}})$ is *correct* if, for all PINs $\mathsf{pin} \in \mathcal{P}$, all messages $\mathsf{msg} \in \mathcal{M}$, on security parameter $\lambda \in \mathbb{N}$ Experiment 14 outputs 1, except with probability $\mathrm{negl}(\lambda)$.

This notion of correctness only guarantees message reconstruction if some shares σ_i, computed using the Decrypt routine, are deleted or unavailable. We do not consider the stronger notion of correctness, in which message reconstruction is possible even if some of the shares σ_i are corrupted (rather than just missing).

The security property for location-hiding encryption states that no efficient adversary can distinguish the encryption of two chosen messages with probability much better than guessing the PIN after roughly N guesses. This property should hold *even if the adversary may adaptively corrupt* up to $f_{\mathsf{secret}} \cdot N$ of the N total secret keys.

For a location-hiding encryption scheme on parameters $(N, n, \mathcal{P}, \mathcal{M}, f_{\mathsf{live}}, f_{\mathsf{secret}})$, an adversary $\mathcal{A}$, and a security parameter $\lambda \in \mathbb{N}$, let $W_{\lambda,\beta}$ denote the probability that the adversary outputs "1" in Experiment 16 with bit $\beta \in \{0,1\}$. Then we define the advantage of $\mathcal{A}$ at attacking a location-hiding encryption scheme $\mathcal{E}$ as:

$$\mathsf{LHEncAdv}[\mathcal{A}, \mathcal{E}](\lambda) := \left| W_{\lambda,0} - W_{\lambda,1} \right|.$$

Definition 15 (Security)**.** Let $\mathcal{E}$ be a location-hiding encryption scheme. on parameters $(N, n, \mathcal{P}, \mathcal{M}, f_{\mathsf{live}}, f_{\mathsf{secret}})$, such that $N = \mathrm{poly}(\lambda)$, $n = \mathrm{poly}(\lambda)$, and $|\mathcal{P}|$ and $|\mathcal{M}|$ grow as (possibly superpolynomial)

Experiment 16 (Location-hiding encryption: Security)**.** We define the following security experiment, which is parameterized by an adversary $\mathcal{A}$, a location-hiding encryption scheme with parameters $(N, n, \mathcal{P}, \mathcal{M}, f_{\mathsf{live}}, f_{\mathsf{secret}})$, a security parameter λ, and a bit $\beta \in \{0,1\}$.

- The challenger runs:

$$(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(1^\lambda), \text{ for } i = 1, \dots, N$$
$$\mathsf{salt} \xleftarrow{R} \{0,1\}^\lambda$$
$$\mathsf{pin} \xleftarrow{R} \mathcal{P}$$

  and sends $(\mathsf{pk}_1, \dots, \mathsf{pk}_N)$ to the adversary.
- The adversary chooses two messages $\mathsf{msg}_0, \mathsf{msg}_1 \in \mathcal{M}$ and sends these to the challenger.
- The challenger computes

$$\mathsf{ct} \leftarrow \mathsf{Encrypt}(\mathsf{pk}_1, \dots, \mathsf{pk}_N, \mathsf{salt}, \mathsf{pin}, \mathsf{msg}_\beta)$$

  and sends $(\mathsf{salt}, \mathsf{ct})$ to the adversary.
- The adversary may make $f_{\mathsf{secret}} \cdot N$ *corruption queries*. At each query:
  - the adversary sends to the challenger an index $i \in [N]$ and
  - the challenger sends to the adversary sk_i.
- Finally, the adversary outputs a bit $\beta' \in \{0,1\}$.

The output of the experiment is the bit β'.

functions of λ. Then we say that the location-hiding encryption scheme $\mathcal{E}$ is *secure* if, for all efficient adversaries $\mathcal{A}$,

$$\mathsf{LHEncAdv}[\mathcal{A}, \mathcal{E}](\lambda) \leq O(N/|\mathcal{P}|) + \mathrm{negl}(\lambda).$$

Remark 17 (Understanding the security definition.). Since our security definition allows the adversary to corrupt up to $f_{\mathsf{secret}} \cdot N$ of the secret keys, the adversary can always reconstruct the plaintext with probability roughly $\frac{f_{\mathsf{secret}} \cdot N}{n|\mathcal{P}|}$ using the following attack to decrypt a ciphertext ct with salt salt:

- Pick two messages $\mathsf{msg}_0, \mathsf{msg}_1 \xleftarrow{R} \mathcal{M}$.
- Pick a candidate PIN $\mathsf{pin}' \xleftarrow{R} \mathcal{P}$.
- Run $I = (i_1, \dots, i_n) \leftarrow \mathsf{Select}(\mathsf{pin}', \mathsf{salt})$.
- Corrupt the keys in I and use them to decrypt ct.
  - If ct decrypts to either msg_0 or msg_1, guess the corresponding bit.
  - Otherwise, try again with another PIN.

If the attacker can corrupt $f_{\mathsf{secret}} \cdot N$ keys, it can try at least $f_{\mathsf{secret}} \cdot N/n$ PINs and its success probability is at least $f_{\mathsf{secret}} \cdot N/(n|\mathcal{P}|)$. Our security analysis shows that this is nearly the best attack

possible against our location-hiding encryption scheme, up to low-order terms.

Remark 18 (Chosen-ciphertext security). A stronger and more realistic definition of security would allow the adversary to make decryption queries to the HSMs in the penultimate stage of the attack game, as in the standard chosen-ciphertext attack (CCA) security game [127, 423]. Since our underlying public-key encryption scheme (hashed ElGamal) is already CCA secure, we believe that—at the cost of some additional complexity in the definitions and proofs—it would be able to achieve a CCA-type notion for location-hiding encryption.

5.11.3 A tedious combinatorial lemma

We will need the following lemma about random sets to prove Theorem 22. The proof of this lemma uses no difficult ideas, but keeping track of the constants involved is a bit tedious.

Definition 19. Let n, N, and Φ be positive integers. Say that a set $S \subseteq [N]$ "$n/2$-covers" a list $L \in [N]^n$ if at least $n/2$ elements of the list L appear in the set S.

Then, let $\mathsf{Cover}_{N,n,\Phi}(\alpha, \beta)$ denote the probability, over the random choice of lists $L_1, \ldots, L_\Phi \xleftarrow{R} [N]^n$, that there exists a set $S \subseteq [N]$ of size αN that $n/2$-covers more than βN of the lists.

Lemma 20. *For $N > en \approx 2.71n$ and $\Phi < 2^{n/2}$, we have*

$$\mathsf{Cover}_{N,n,\Phi}(\tfrac{1}{16}, \tfrac{3}{n}) \leq 2^{-N/4}.$$

Proof. Let $\alpha = 1/16$ and $\beta = 3/n$. Our task is then to bound the probability that there exists a set of size αN that $n/2$-covers more than βN lists.

Fix a set $S \subseteq [N]$ of size αN. We compute the probability that S $n/2$-covers more than βN of the chosen lists.

The probability that a fixed set S $n/2$-covers a list L is at most $\binom{n}{n/2} \cdot \alpha^{n/2}$, over the random choice of $L \xleftarrow{R} [N]^n$. Using the inequality $\binom{n}{k} \leq (ne/k)^k$, we can bound this probability by $(2e\alpha)^{n/2}$.

Then, the probability that a fixed set S $n/2$-covers some subset of βN of the Φ lists is then

$$\binom{\Phi}{\beta N} \left((2e\alpha)^{n/2}\right)^{\beta N} \leq \left(\frac{\Phi e}{\beta N} \cdot (2e\alpha)^{n/2}\right)^{\beta N}$$

Finally we apply the union bound over all $\binom{N}{\alpha N}$ possible choices of the set S to get the final probability:

$$\begin{aligned}
&\binom{N}{\alpha N} \cdot \left(\frac{\Phi e}{\beta N} \cdot (2e\alpha)^{n/2}\right)^{\beta N} \\
&\leq (e/\alpha)^{\alpha N} \cdot \left(\frac{\Phi e}{\beta N} \cdot (2e\alpha)^{n/2}\right)^{\beta N} \\
&\leq (e/\alpha)^{\alpha N} \cdot \left(\frac{\Phi e}{\beta N} \cdot (2e\alpha)^{n/2}\right)^{\beta N}
\end{aligned}$$

and since $(e/\alpha)^\alpha = (16e)^{1/16} < 2^{1/2}$ and $\beta = 3/n$,

$$\leq 2^{N/2} \cdot \left(\frac{\Phi e n}{3N} \cdot (2e\alpha)^{n/2} \right)^{\beta N}$$

and since we have assumed $N > en$, $en/N \leq 1$, so

$$\leq \left[2^{1/2} \cdot \left(\Phi \cdot (2e\alpha)^{n/2} \right)^{\beta} \right]^N$$

and letting $\alpha = 1/16$ implies $2e\alpha = e/16 < 2^{-3/2}$, so

$$\leq 2^{N/2} \left(\Phi \cdot 2^{-3n/4} \right)^{3N/n}$$

and since $\Phi < 2^{n/2}$,

$$\leq 2^{N/2} \left(2^{-n/4} \right)^{3N/n} \leq 2^{-N/4}.$$

□

5.11.4 Our construction

Our construction, which is parameterized by a standard public-key encryption scheme, appears in Figure 15. We instantiate the public-key encryption scheme with hashed ElGamal encryption, which we recall here.

Hashed ElGamal We instantiate the location-hiding encryption scheme of Section 5.5 with the "Hashed ElGamal" encryption scheme. The scheme uses a cyclic group $\mathbb{G} = \langle g \rangle$ of prime order p, in which we assume that the computational Diffie-Hellman problem is hard. It also uses an authenticated encryption scheme $(\mathsf{AE.Encrypt}, \mathsf{AE.Decrypt})$ with key space $\mathcal{K}$ and a hash function $\mathsf{Hash}' : \mathbb{G} \to \mathcal{K}$.

A keypair is a pair $(x, g^x) \in \mathbb{Z}_p \times \mathbb{G}$, where $x \xleftarrow{R} \mathbb{Z}_p$. To encrypt a message $m \in \{0,1\}^\ell$ to public key $X \in \mathbb{G}$, the encryptor computes $\big(g^r, \mathsf{AE.Encrypt}(\mathsf{Hash}'(X^r), m)\big)$.

A standard argument [71] shows that Hashed ElGamal satisfies semantic security against chosen-ciphertext attacks [127, 423].

In our use of hashed ElGamal, we can provide domain separation between different ciphertexts by prepending inputs to the hash function Hash' during encryption and decryption with: (1) the client's username, (2) the salt associated with the ciphertext, and (3) the public keys of the n public keys to which the client encrypted the ciphertext. All of these values are available to the client during encryption and to the HSMs during decryption.

Location-hiding encryption scheme. The construction is parameterized by:

- a universe size $N \in \mathbb{N}$,
- a cluster size $n \in \mathbb{N}$,
- a PIN-space $\mathcal{P}$,
- a recovery threshold $t \in \mathbb{N}$,
- a public-key encryption scheme $(\mathsf{PKE.KeyGen}, \mathsf{PKE.Encrypt}, \mathsf{PKE.Decrypt})$, and
- an authenticated encryption scheme $(\mathsf{AE.Encrypt}, \mathsf{AE.Decrypt})$ with keyspace $\mathbb{F}$ (some finite field), and
- a security parameter $\lambda \in \mathbb{N}$.

The message space is $\{0,1\}^*$. We use a hash function $\mathsf{Hash}\colon \{0,1\}^\lambda \times \mathcal{P} \to [N]^n$ (e.g., built from SHA-256) which we model as a random oracle [53]. We use t-out-of-n Shamir secret sharing [453]; we denote the sharing and reconstruction algorithms over a finite field $\mathbb{F}$ by $(\mathsf{Shamir.Share}_\mathbb{F}, \mathsf{Shamir.Reconst}_\mathbb{F})$.

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$.

- On input security parameter λ, expressed in unary, run the key-generation routine for the underlying public-key encryption scheme: $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.

$\mathsf{Encrypt}(\mathsf{pk}_1, \dots, \mathsf{pk}_N, \mathsf{pin}, \mathsf{msg}) \to (\mathsf{salt}, \mathsf{ct})$.

- Sample a random salt $\mathsf{salt} \xleftarrow{R} \{0,1\}^\lambda$ and compute $(i_1, \dots, i_n) \leftarrow \mathsf{Hash}(\mathsf{salt}, \mathsf{pin})$.
- Sample a transport key $k \xleftarrow{R} \mathbb{F}$ and split it using t-out-of-n-Shamir secret sharing [453]: $(k_1, \dots, k_n) \leftarrow \mathsf{Shamir.Share}_\mathbb{F}(k)$.
- Encrypt each share of the key using the underlying public-key encryption scheme. That is, for $j \in \{1, \dots, n\}$, set $C_i \leftarrow \mathsf{PKE.Encrypt}\big(\mathsf{pk}_{i_j}, k_i\big)$.
- Set $M \leftarrow \mathsf{AE.Encrypt}(k, \mathsf{msg})$, $\mathsf{ct} \leftarrow (M, C_1, \dots, C_n)$, and output $(\mathsf{salt}, \mathsf{ct})$.

$\mathsf{Select}(\mathsf{salt}, \mathsf{pin}) \to (i_1, \dots, i_n) \in [N]^n$.

- Output $\mathsf{Hash}(\mathsf{salt}, \mathsf{pin}) \in [N]^n$.

$\mathsf{Decrypt}(\mathsf{sk}, i, \mathsf{ct}) \to \sigma_i \in \mathbb{G}$.

- Parse ct as $(M, C_1, \dots, C_n)$.
- Output $\sigma \leftarrow \big(M, \mathsf{PKE.Decrypt}(\mathsf{sk}, C_i)\big)$.

$\mathsf{Reconstruct}(\sigma_1, \dots, \sigma_n) \to \mathsf{msg}$.

- For $i \in [n]$, Parse each share σ_i as a pair (M_i, k_i).
- Let $k \leftarrow \mathsf{Shamir.Reconst}_\mathbb{F}(k_1, \dots, k_n) \in \mathbb{F}$.
- Let M be the most common value in $\{M_1, \dots, M_n\}$.
- Output $\mathsf{AE.Decrypt}(k, M)$.

Figure 15: Our construction of location-hiding encryption.

5.11.5 Proof of correctness

We now prove:

Theorem 21. *Consider an instantiation of the encryption scheme $\mathcal{E}$ of Figure 15 with parameters $(N, n, \mathcal{P}, t)$, with cluster size $n = \Omega(\lambda)$, on security parameter λ, and recovery threshold $t = n/2$. Then the resulting scheme is a* correct *location-hiding encryption scheme (in the sense of Definition 13) for any fault-tolerance parameter $f_{\mathsf{live}} \leq \frac{1}{8}$.*

Proof. To prove the claim, we need to bound the probability that, out of a random size-n subset of all N keys, no more than $t = n/2$ are failed. Fix a size-$(n/2)$ subset S of the n chosen keys. The probability that the keys in S are all failed is $(f_{\mathsf{live}})^{n/2}$. We can now take a union bound over all $\binom{n}{n/2}$ choices of S to bound the final failure probability:

$$\Pr[\text{fail}] \leq \binom{n}{n/2} \cdot (f_{\mathsf{live}})^{n/2} \leq 2^n \cdot \left(\frac{1}{2^3}\right)^{n/2} = 2^{-n/2}.$$

Since we have taken $n = \Omega(\lambda)$, this probability is negligible in λ. □

5.11.6 Proof of security

We state the main theorem and then prove it here.

Theorem 22. *The location-hiding encryption scheme $\mathcal{E}$ of Figure 15 instantiated with the hashed-ElGamal public-key encryption system (Section 5.11.4) is* secure*, in the sense of Definition 15, in the random-oracle model, for recovery threshold $t = n/2$ and corruption threshold $f_{\mathsf{secret}} = 1/16$,*

More precisely, we consider an instantiation of our location-hiding encryption scheme $\mathcal{E}$ with parameters $(N, n, \mathcal{P}, t = n/2)$, where $N > e \cdot n \approx 2.71n$ and $|\mathcal{P}| < 2^{n/2}$, using hashed ElGamal encryption for the public-key encryption scheme, and using an arbitrary authenticated encryption scheme AE.

Then, let $\mathcal{A}$ be an adversary that attack $\mathcal{E}$ in the location-hiding encryptions security game with corruption threshold $f_{\mathsf{secret}} = 1/16$. Denote $\mathcal{A}$'s attack advantage as $\mathsf{LHEncAdv}[\mathcal{A}, \mathcal{E}]$*, Then if $\mathcal{A}$ makes at most Q queries to the random oracle* Hash'*, used in hashed ElGamal encryption, we construct:*

- *an efficient algorithm $\mathcal{B}_{\mathsf{CDH}}$ that breaks CDH in group $\mathbb{G}$ with advantage* $\mathsf{CDHAdv}[\mathcal{B}_{\mathsf{CDH}}, \mathbb{G}]$ *and*
- *an efficient algorithm $\mathcal{B}_{\mathsf{AE}}$ that breaks the authenticated encryption scheme* AE *with advantage* $\mathsf{AEAdv}[\mathcal{B}_{\mathsf{AE}}, \mathsf{AE}]$

such that

$$\mathsf{LHEncAdv}[\mathcal{A}, \mathcal{E}] \;\leq\; 2^{-N/4} + N \cdot Q \cdot \mathsf{CDHAdv}[\mathcal{B}_{\mathsf{CDH}}, \mathbb{G}] + \frac{3N}{n|\mathcal{P}|} + \mathsf{AEAdv}[\mathcal{B}_{\mathsf{AE}}, \mathsf{AE}].$$

Game 0. We define Game 0 by instantiating Experiment 16 our location-hiding encryption scheme, instantiated in turn with hashed-ElGamal encryption (Section 5.11.4). Game 0 is parameterized by an adversary $\mathcal{A}$, a group $\mathbb{G}$ of prime order p with generator g, a universe size $N \in \mathbb{N}$, a cluster size $n \in \mathbb{N}$, a PIN-space $\mathcal{P}$, a recovery threshold t, a security parameter λ, hash functions $\mathsf{Hash} : \{0,1\}^\lambda \times \mathcal{P} \to [N]^n$ and $\mathsf{Hash}' : \mathbb{G} \to \mathbb{F}$ (which we model as random oracles where the challenger plays the role of the random oracle), an authenticated encryption scheme $(\mathsf{AE.Encrypt}, \mathsf{AE.Decrypt})$ with keyspace $\mathbb{F}$ and message space $\{0,1\}^*$, a security parameter $\lambda \in \mathbb{N}$, and a bit $\beta \in \{0,1\}$.

- The challenger runs

$$\mathsf{sk}_i \xleftarrow{\mathrm{R}} \mathbb{Z}_p,\ \mathsf{pk}_i \leftarrow g^{\mathsf{sk}_i} \text{ for } i \in \{1,\dots,N\}$$

and sends $(\mathsf{pk}_1,\dots,\mathsf{pk}_N)$ to the adversary.

- The adversary chooses two messages $\mathsf{msg}_0, \mathsf{msg}_1 \in \mathcal{M}$ and sends these to the challenger.
- The challenger then computes the ciphertext using our location-hiding encryption scheme instantiated with hashed-ElGamal. That is, the challenger computes

$$\begin{aligned} \mathsf{salt} &\xleftarrow{\mathrm{R}} \{0,1\}^\lambda \\ \mathsf{pin} &\xleftarrow{\mathrm{R}} \mathcal{P} \\ (i_1,\dots,i_n) &\leftarrow \mathsf{Hash}(\mathsf{salt},\mathsf{pin}) \\ k &\xleftarrow{\mathrm{R}} \mathbb{F} \\ (k_1,\dots,k_n) &\leftarrow \mathsf{Shamir.Share}_{\mathbb{F}}(k). \end{aligned}$$

Here, $\mathsf{Shamir.Share}_{\mathbb{F}}$ denotes t-out-of-n Shamir secret sharing over the field $\mathbb{F}$. The challenger then encrypts the shares $(k_1,\dots,k_n)$ of the transport key k using hashed ElGamal encryption. For $j \in \{1,\dots,n\}$, the challenger computes:

$$\begin{aligned} &r_j \xleftarrow{\mathrm{R}} \mathbb{Z}_p \\ &\kappa_j \leftarrow \mathsf{Hash}'((\mathsf{pk}_{i_j})^{r_j}) \\ &C_j \leftarrow (g^{r_j}, \kappa_j \oplus k_j). \end{aligned}$$

Finally, the challenger encrypts the message msg with the transport key k and outputs the ciphertext:

$$\begin{aligned} &M \leftarrow \mathsf{AE.Encrypt}(k, \mathsf{msg}_\beta) \\ &\mathsf{ct} \leftarrow (M, C_1, \dots, C_n) \end{aligned}$$

and sends $(\mathsf{salt}, \mathsf{ct})$ to the adversary.

- The adversary may make adaptive $f_{\mathsf{secret}} \cdot N$ *corruption queries* and may perform computation between its queries. At each query:
  - the adversary sends to the challenger an index $i \in [N]$ and
  - the challenger sends to the adversary sk_i.
- Finally, the adversary outputs a bit $\beta' \in \{0,1\}$.

The output of the experiment is the bit β'.

Notice that if the number of HSMs N grows much larger than the size of the PIN space $|\mathcal{P}|$, the bound of Theorem 22 on the adversary's advantage becomes vacuous. (In fact, this limitation is inherent—see Remark 17 in Section 5.11.) To support extremely large data deployments, it would be possible to shard users into data centers of moderate size (e.g., $N \approx 50,000$).

The main technical challenge in proving Theorem 22 comes from the fact that the adversary may adaptively compromise a subset of the secret keys. This is very similar to the issues that arise when proving a cryptosystem secure against "selective opening attacks" [52, 162]

Proof of Theorem 22 (Security). We construct the following series of games and show that the difference between the adversary's advantage from one game to the next is small. For $i \in \{0, \dots, 4\}$, let W_i the event that the adversary wins in Game i, where we define the winning condition for each game below.

Game 0. Game 0 proceeds according to the location-hiding security game where the challenger plays the role of the random oracle. For location-hiding encryption scheme $\mathcal{E}$ and adversary $\mathcal{A}$ in Game 0, by definition,

$$\Pr[W_0] = \mathsf{LHEncAdv}[\mathcal{A}, \mathcal{E}] \ . \tag{5.1}$$

Game 1. Game 1 proceeds in the same way as Game 0, except that for the adversary to win in Game 1, we also require that a certain "bad event" does not take place. This bad event is that many different PINs hash to the same set of HSMs. Formally, the bad event is that there exists a set S of $f_{\mathsf{live}} \cdot N = N/16$ HSMs and a set of $\frac{3}{n} \cdot N$ PINs $\{\mathsf{pin}_1, \mathsf{pin}_2, \dots\}$ such that, for each $i \in [N/n]$, it holds that: $|\mathsf{Hash}(\mathsf{salt}, \mathsf{pin}_i) \cap S| \geq t = n/2$. Using Definition 19, we can simply write the probability of this bad event as $\mathsf{Cover}_{N,n,\Phi}(\frac{1}{16}, \frac{3}{n})$.

By Lemma 20, we have that

$$|\Pr[W_0] - \Pr[W_1]| \leq 2^{-N/40} \ . \tag{5.2}$$

Game 2. Game 2 proceeds in the same way as Game 1 except that for the adversary to win, we require that

- the adversary wins in Game 1, and
- the adversary never makes a random-oracle query for $(\mathsf{pk}_i)^{r_i}$ where $i \in (i_1, \dots, i_n)$ unless the adversary issued a corruption query for i first.

Then by Lemma 23,

$$|\Pr[W_1] - \Pr[W_2]| < N \cdot Q \cdot \mathsf{CDHAdv}[\mathcal{B}_{\mathsf{CDH}}, \mathbb{G}] \ . \tag{5.3}$$

Game 3. Game 3 proceeds as in Game 2 except that for the adversary to win, we require that

- the adversary wins in Game 2, and
- the adversary makes fewer than $t = n/2$ corruption queries to elements in $I = (i_1, \dots, i_n)$.

Then, by Lemma 24,

$$|\Pr[W_2] - \Pr[W_3]| \leq \frac{3N}{n|\mathcal{P}|} \tag{5.4}$$

Game 4. In Game 4, we modify the behavior of the challenger. Rather than encrypting msg with k, instead the challenger samples an additional key $k' \xleftarrow{R} \mathbb{F}$ for an authenticated encryption scheme and uses k' to encrypt msg.

Then, by Lemma 25,

$$|\Pr[W_3] - \Pr[W_4]| = 0. \tag{5.5}$$

Final reduction. In Lemma 26, we show that

$$|\Pr[W_4]| = \mathsf{AEAdv}[\mathcal{B}_{\mathsf{AE}}, \mathsf{AE}] \ . \tag{5.6}$$

Putting it together.

We can write $\Pr[W_0]$ as

$$\begin{aligned} \Pr[W_0] \leq & |\Pr[W_0] - \Pr[W_1]| \\ & + |\Pr[W_1] - \Pr[W_2]| \\ & + |\Pr[W_2] - \Pr[W_3]| \\ & + |\Pr[W_3] - \Pr[W_4]| + \Pr[W_4]. \end{aligned}$$

Then, by (5.1)-(5.6), we can bound $\mathsf{LHEncAdv}$ as follows:

$$\begin{aligned} \mathsf{LHEncAdv}[A, \mathcal{E}] \leq & 2^{-N/40} \\ & + N \cdot Q \cdot \mathsf{CDHAdv}[\mathcal{B}_{\mathsf{CDH}}, \mathbb{G}] \\ & + \frac{3N}{n|\mathcal{P}|} \\ & + \mathsf{AEAdv}[\mathcal{B}_{\mathsf{AE}}, \mathsf{AE}] \ . \end{aligned}$$

□

Lemma 23. *Let W_1 be the event that the adversary wins in Game 1 and W_2 be the event that the adversary wins in Game 2. In particular, for every adversary $\mathcal{A}$ in Game 1, we construct a CDH adversary $\mathcal{B}_{\mathsf{CDH}}$ in group $\mathbb{G}$ with advantage $\mathsf{CDHAdv}[\mathcal{B}_{\mathsf{CDH}}, \mathbb{G}]$ that runs in time linear in the runtime of $\mathcal{A}$ and makes Q Hash'-oracle queries such that*

$$|\Pr[W_1] - \Pr[W_2]| \leq N \cdot Q \cdot \mathsf{CDHAdv}[\mathcal{B}_{\mathsf{CDH}}, \mathbb{G}] \ .$$

Proof. Let F be the event that the adversary makes a Hash'-oracle query at the point $\mathsf{pk}_i^{r_i}$ *before* making a corruption query for key i. Then, by the definition of Games 1 and 2, and the Difference Lemma [456], we have

$$|\Pr[W_1] - \Pr[W_2]| \leq \Pr[F].$$

So, to prove the lemma we need only bound $\Pr[F]$.

Given an adversary $\mathcal{A}$ in Game 1, we will construct a CDH adversary $\mathcal{B}_{\mathsf{CDH}}$ such that $\Pr[F] \leq N \cdot Q \cdot \mathsf{CDHAdv}[\mathcal{B}_{\mathsf{CDH}}, \mathbb{G}]$. We construct $\mathcal{B}_{\mathsf{CDH}}$ as follows:

- The algorithm $\mathcal{B}_{\mathsf{CDH}}$ receives a challenge tuple $(g, g^r, g^x) \in \mathbb{G}^3$ from the CDH challenger.
- The algorithm $\mathcal{B}_{\mathsf{CDH}}$ plays the role of the Game-1 challenger. The algorithm $\mathcal{B}_{\mathsf{CDH}}$ deviates from the normal behavior of the Game-1 challenger in two ways:
    1. The challenger chooses a random value $i^* \xleftarrow{\mathrm{R}} [N]$. For the i^*th keypair, algorithm $\mathcal{B}_{\mathsf{CDH}}$ sets $\mathsf{pk}_i \leftarrow g^x$, where g^x is the value from the CDH challenger. (Algorithm $\mathcal{B}_{\mathsf{CDH}}$ generates all other keypairs as the challenger in Game 1 does.)
    2. When constructing the final ciphertext, algorithm $\mathcal{B}_{\mathsf{CDH}}$ sets the encryption nonce in the i^*th ciphertext to be the value g^r, where g^r is the value from the CDH challenger. For the value $\mathsf{Hash}'((\mathsf{pk}_{i^*})^r)$ needed to construct the ciphertext, the algorithm $\mathcal{B}_{\mathsf{CDH}}$ just chooses a random element in $\mathbb{F}$.
- The algorithm $\mathcal{B}_{\mathsf{CDH}}$ then proceeds in the same way as the Game 1 challenger with the following modification:
    - If $\mathcal{A}$ makes a corruption query for i^*, $\mathcal{B}_{\mathsf{CDH}}$ outputs $\perp$.
- If $\mathcal{B}_{\mathsf{CDH}}$ did not output $\perp$, then $\mathcal{B}_{\mathsf{CDH}}$ randomly chooses one of the points at which $\mathcal{A}$ made an Hash'-oracle query and returns the queried point to the CDH challenger.

We now compute algorithm $\mathcal{B}_{\mathsf{CDH}}$'s CDH advantage. Whenever event F occurs, the algorithm $\mathcal{A}$ makes a random-oracle query to a point $(\mathsf{pk}_i)^{r_i}$, for some $i \in [N]$, before issuing a corruption query at i. Notice that $\mathcal{B}_{\mathsf{CDH}}$ succeeds whenever:

1. event F occurs,
2. $i = i^*$, and
3. the algorithm $\mathcal{B}_{\mathsf{CDH}}$ guesses the correct random-oracle query to output.

These three events are independent. Furthermore, their probabilities are:

1. $\Pr[F]$ – to be computed later,
2. $\Pr[i = i^*] = 1/N$, and
3. $\Pr[\text{guesses correct r.o. query}] = 1/Q$.

Therefore,

$$\mathsf{CDHAdv}[\mathcal{B}_{\mathsf{CDH}}, \mathbb{G}] \geq \Pr[F] \cdot \left(\frac{1}{N}\right) \cdot \left(\frac{1}{Q}\right),$$

which proves the lemma. □

Lemma 24. *Let W_2 be the event that the adversary wins in Game 2 and W_3 be the event that the adversary wins in Game 3. Then*

$$|\Pr[W_2] - \Pr[W_3]| \leq \frac{3N}{n|\mathcal{P}|} .$$

Proof. Let F be the event that the adversary $\mathcal{A}$ wins in Game 2 but not in Game 3. By construction of the games, we have

$$|\Pr[W_2] - \Pr[W_3]| \leq \Pr[F],$$

so our task is to bound $\Pr[F]$.

To analyze $\Pr[F]$, we consider a modified game between $\mathcal{A}$ and its challenger in Games 2 and 3. Here, we modify the challenger to halt the execution of $\mathcal{A}$ as soon as event F occurs. This modification cannot increase $\Pr[F]$.

However, in the modified game, $\mathcal{A}$'s view is *independent of the uncorrupted elements of I* until event F occurs. This is so because the only values that the adversary sees that depend on the set I are the κ_j values. Since these are computed as the output of $\mathsf{Hash}'((\mathsf{pk}_i)^{r_j})$, until the adversary queries the random oracle Hash' at the point $(\mathsf{pk}_i)^{r_j}$, these κ values are also independent of I.

Since we have already argued (Game 2) that the challenger never queries the random oracle at a point $(\mathsf{pk}_i)^{r_j}$ without having corrupted i, the ciphertext values that the adversary sees are independent of the uncorrupted elements of I.

Therefore, the answers to the corruption queries give the adversary no information on the uncorrupted elements of I. Then $\Pr[F]$ is just the probability that the adversary makes $N/16$ *non-adaptive* corruption queries and is able to cause event F to occur.

By the winning condition of Game 1, for any set of $N/16$ corrupted HSMs S, there are at most $3N/n$ PINs pin such that $|\mathsf{Hash}(\mathsf{salt}, \mathsf{pin}) \cap S| > n/2$. Therefore, the probability that F occurs is $\Pr[F] \leq (3N/n)/|\mathcal{P}| = 3N/(n|\mathcal{P}|)$. □

Lemma 25. *Let W_3 be the event that the adversary wins in Game 3 and W_4 be event that the adversary wins in Game 4. Then*

$$|\Pr[W_3] - \Pr[W_4]| = 0 .$$

Proof. We use the security of Shamir secret sharing to prove this lemma. For each of the indexes that the adversary does not corrupt, we can replace $\kappa_i \oplus k_i$ with a random element in $\mathbb{F}$ in both games. We know that in both games, the adversary corrupts fewer than $n/2$ of the keys in I, and so the adversary learns fewer than $n/2$ of the shares $k_1, \ldots, k_n$ of the transport key. Therefore, we can replace the transport key k with $k' \xleftarrow{R} \mathbb{F}$, completing the proof. □

Lemma 26. *Let W_4 be event that the adversary wins in Game 4. Then given an adversary $\mathcal{A}$ in Game 4, we construct an AE adversary $\mathcal{B}_{\mathsf{AE}}$ for AE scheme AE with advantage $\mathsf{AEAdv}[\mathcal{B}_{\mathsf{AE}}, \mathsf{AE}]$ that runs in time linear in the runtime of $\mathcal{A}$ such that*

$$\Pr[W_4] = \mathsf{AEAdv}[\mathcal{B}_{\mathsf{AE}}, \mathsf{AE}] .$$

Proof. Given an adversary $\mathcal{A}$ in Game 4, we will construct an adversary $\mathcal{B}_{\mathsf{AE}}$ attacking the authenticated encryption scheme AE such that the advantage of $\mathcal{B}_{\mathsf{AE}}$ is identical to that of $\mathcal{A}$. We construct $\mathcal{B}_{\mathsf{AE}}$ as follows:

- The algorithm $\mathcal{B}_{\mathsf{AE}}$ computes $(\mathsf{pk}_1, \dots, \mathsf{pk}_N)$ in the same way as the Game 4 challenger and sends them to $\mathcal{A}$.
- When $\mathcal{A}$ returns the messages $\mathsf{msg}_0, \mathsf{msg}_1 \in \mathcal{M}$, $\mathcal{B}_{\mathsf{AE}}$ forwards the messages to the AE challenger and receives the ciphertext c^* from the AE challenger.
- The algorithm $\mathcal{B}_{\mathsf{AE}}$ then computes the ciphertext in the same way as the Game 4 challenger except that instead of computing the ciphertext as $(\mathsf{AE.Encrypt}(k, \mathsf{msg}), C_1, \dots, C_n)$, $\mathcal{B}_{\mathsf{AE}}$ sends $\mathcal{A}$ $(c^*, C_1, \dots, C_n)$.
- The algorithm $\mathcal{B}_{\mathsf{AE}}$ responds to queries in the same way as the Game 4 challenger.
- When $\mathcal{A}$ outputs bit $\beta' \in \{0, 1\}$, $\mathcal{B}_{\mathsf{AE}}$ forwards the response to the AE challenger.

Because in Game 4, we sample k' independently from the rest of the messages, $\mathcal{A}$ cannot distinguish between interaction with $\mathcal{B}_{\mathsf{AE}}$ and the Game 4 challenger. The advantage of $\mathcal{A}$ is exactly $\mathsf{AEAdv}[\mathcal{B}_{\mathsf{AE}}, \mathsf{AE}]$, completing the proof. □

5.12 Conclusion

SafetyPin is an encrypted backup system that (a) requires its users to only remember a short PIN, (b) defeats brute-force PIN-guessing attacks using hardware protections, and (c) provides strong protection against hardware compromise. SafetyPin demonstrates that it is possible to reap the benefits of hardware security protections without turning these hardware devices into single points of security failure.

Chapter 6

Larch: An authentication logging system without a single point of security failure

6.1 Introduction

Account security is a perennial weak link in computer systems. Even well-engineered systems with few bugs become vulnerable once human users are involved. With poorly engineered or configured systems, account compromise is often the first of several cascading failures. In general, 82% of data breaches involve a human element, with the most common methods including use of stolen credentials (40%) and phishing (20%) [502].

When users and administrators identify stolen credentials, it is challenging to determine the extent of the damage. Not knowing what an attacker accessed can lead to either inadequate or overly extensive recovery. LastPass suffered a breach in November 2022 because they didn't fully recover from a compromise the previous August [438]. Conversely, Okta feared 366 organizations might have been accessed when an attacker gained remote desktop access at one of their vendors. It took a three-month investigation to determine that, in fact, only two organizations, not 366, had really been victims of the breach [172].

Single sign-on schemes, such as OpenID [443] and "Sign in with Google," can keep an authentication log and thereby determine the extent of a credential compromise. However, these centralized systems represent a security and privacy risk: they give a third party access to all of a user's accounts and to a trace of their authentication activity.

An ideal solution would give the benefits of universal authentication logging without the security and privacy drawbacks of single-sign-on systems. For security, the logging service shouldn't be able to authenticate on behalf of a user. For privacy, the logging service should learn no information about a user's authentication history: the log service should not even learn if the user is authenticating to the same web service twice or to two separate web services.

In this chapter, we propose larch ("login archive"), an accountable authentication framework with strong security and privacy properties. Authentication takes place between a user and a service, which we call the *relying party*. In larch, we add a third party: a user-chosen larch log service. The

larch log service provides the user with a complete, comprehensive history of her authentication activity, which helps users detect and recover from compromises. Once an account is registered with larch, even an attacker who controls the user's client cannot authenticate to the account without the larch log service storing a record that allows the user to recover the time and relying-party name.

The key challenge in larch is allowing the log service to maintain a complete authentication history *without becoming a single point of security or privacy failure*. A malicious larch log service cannot access users' accounts and learns no information about users' authentication histories. Only users can decrypt their own log records.

Larch works with any relying party that supports one of three standard user authentication schemes: FIDO2 [179] (popularized by Yubikeys and Passkeys [14]), TOTP [369] (popularized by Google Authenticator), and password-based login. FIDO2 is the most secure but least widely deployed of the three options.

A larch deployment consists of two components: a browser add-on, which manages the user's authentication secrets, and one or more larch log services, which store authentication logs on behalf of a set of users. At a high level, larch provides four operations. (1) Upon deciding to use larch, a user performs a one-time *enrollment* with a log service. (2) For each account to use with larch, the user runs *registration*. To relying parties, registration looks like adding a FIDO2 security key, adding an authenticator app, or setting a password. (3) The user then performs *authentication* with larch as necessary to access registered accounts. Finally, (4) at any point the user can *audit* login activity by downloading and decrypting the complete history of authentication events to all accounts. The client can use auditing for intrusion detection or to evaluate the extent of the damage after a client has been compromised.

All authentication mechanisms require generating an authentication credential based on some secret. In FIDO2, the secret is a signature key and the credential is a digital signature; the signed payload depends on the name of the relying party and a fresh challenge, preventing both phishing and credential reuse. With TOTP, the secret is an HMAC key and the credential an HMAC of the current time, which prevents credential reuse in the future. With passwords, the credential is simply the password, which has the disadvantage that it can be reused once a malicious client obtains it.

Larch splits the authentication secret between the client and log service so that both parties must participate in authentication. We introduce split-secret authentication protocols for FIDO2, TOTP, and password-based login. At the end of each protocol, the log service holds an encrypted authentication log record and the client holds a credential. Larch ensures that if the client obtains a valid credential, the log service also obtains a well-formed log record, even if the client is compromised and behaves maliciously. At the same time, the log service learns no information about the relying parties that the user authenticates to.

We design larch to achieve the following (informal) security and privacy goals:

- *Log enforcement against a malicious client:* An attacker that compromises a client cannot authenticate to an account that the client created before compromise without the log obtaining a well-formed, encrypted log record.
- *Client privacy and security against a malicious log:* A malicious log service cannot authenticate to the user's accounts or learn any information about the relying parties to which the user

has authenticated, including whether two authentications are for the same account or different accounts.

- *Client privacy against a malicious relying party:* Colluding malicious relying parties cannot link a user across accounts.

Larch's FIDO2 protocol uses zero-knowledge proofs [208] to convince the log that an encrypted authentication log record generated by the client is well-formed relative to the digest of a FIDO2 payload. If it is, the client and log service sign the digest with a new, lightweight two-party ECDSA signing protocol tailored to our setting. For TOTP, larch executes an authentication circuit using an existing garbled-circuit-based multiparty computation protocol [511, 525]. For password-based login, the client privately swaps a ciphertext encrypting the relying party's identity for the log's share of the corresponding password using a discrete-log-based protocol [218].

In the event that a user's device is compromised, a user can revoke access to all accounts—even accounts she may have forgotten about—by interacting only with the log service. At the same time, involving the log service in every authentication could pose a reliability risk (just as relying on OpenID does). We show how to split trust across multiple log service providers to strengthen availability guarantees, making larch strictly better than OpenID for all three of security, privacy, and availability.

We expect users to perform many password-based authentications, some FIDO2 authentications, and a comparatively small number of TOTP authentications. Given a client with four cores and a log server with eight cores, an authentication with larch takes 150ms for FIDO2, 91ms for TOTP, and 74ms for passwords (excluding preprocessing, which takes 1.23s for TOTP). One authentication requires 1.73MiB of communication for FIDO2, 65.2MiB for TOTP, and 3.25KiB for passwords. TOTP communication costs are comparatively high because we use garbled circuits [511]; however, all but 202KiB of the communication can be moved into a preprocessing step.

Larch shows that it is possible to achieve privacy-preserving authentication logging that is backwards compatible with existing standards. Moreover, larch provides new paths for FIDO2 adoption, as larch users can authenticate using FIDO2 without dedicated hardware tokens, which could motivate more relying parties to deploy FIDO2. Users who do own hardware tokens can use them to authenticate to the larch log service, providing strong security guarantees for relying parties that do not yet support FIDO2 (albeit without the anti-phishing protection). We also suggest small changes to the FIDO standard that would substantially reduce the overheads of larch while providing the same security and privacy properties.

6.2 Design overview

We now give an overview of larch.

6.2.1 Entities

A larch deployment involves the following entities:

Users. We envision a deployment with millions of users, each of which has hundreds of accounts at different online services—shopping websites, financial institutions, news sites, and so on. Each user has an account at a larch log service, secured by a strong, unique password and optionally (but ideally) strong second-factor authentication such as a FIDO2 hardware security key. (In Section 6.6, we describe how a user can create accounts with multiple log services in order to protect against faulty logs.) A user also has a set of devices (e.g. laptop, phone, tablet) running larch client software and storing larch secrets, including cryptographic keys and passwords.

Relying parties. A relying party is any website that a user authenticates to (e.g., a shopping website or bank). Larch is compatible with any relying party that supports authentication via FIDO2 (U2F) [179,506], time-based one-time passwords (TOTP) [369], or standard passwords. The strength of larch's security guarantees depends on the strength of the underlying authentication method.

Log service. Whenever the user authenticates to a relying party, the client must communicate with the log service. We envision a major service provider (e.g. Google or Apple) deploying this service on behalf of their customers. The log service:

- keeps an encrypted record of the user's authentication history, but
- learns no information about which relying party the user authenticates to.

At any time, a client can fetch this authentication record from the log service and decrypt it to see the user's authentication history. That is, if an attacker compromises one of Alice's devices and authenticates to `github.com` as Alice, the attacker will leave an indelible trace of this authentication in the larch log. At the same time, to protect Alice's privacy, the log service learns no information about which relying parties Alice has authenticated to. A production log service should consist of multiple, georeplicated servers to ensure high availability.

6.2.2 Protocol flow

Background. We use two-out-of-two *additive secret sharing* [453]: to secret-share a value $x \in \{0,\ldots,p-1\}$, choose random values $x_1, x_2 \in \{0,\ldots,p-1\}$ such that $x_1 + x_2 = x \bmod p$. Neither x_1 nor x_2 individually reveals any information about x. We also use a cryptographic *commitment scheme*: to commit to a value $x \in \{0,1\}^*$, choose a random value $r \in \{0,1\}^{256}$ (the commitment *opening*) and output the hash of $(x\|r)$ using a cryptographic hash function such as SHA-256. For computationally bounded parties, the commitment reveals no information about x, but makes it impractical to convince another party that the commitment opens to a value $x' \neq x$.

The client's interaction with the log service consists of four operations.

Step 1: Enrollment with a log service. To use larch, a user must first *enroll* with a larch log service by creating an account. In addition to configuring traditional account authentication (i.e., setting a password and optionally registering FIDO2 keys), the user's client generates a secret *archive key* for each authentication method supported. For FIDO2 and TOTP, the archive key is a symmetric encryption key, and the client sends the log service a commitment to this key. For passwords, the archive key is an ElGamal private encryption key, so the client sends the log service the corresponding public key. The client subsequently encrypts log records using these archive keys,

while the log service verifies these log records are well-formed using the corresponding commitment or public key.

Step 2: Registration with relying parties. After the user has enrolled with a log service, she can create accounts at relying parties (e.g., `github.com`) using larch-protected credentials. We call this process *registration*. Registration works differently depending on which authentication mechanism the relying party uses: FIDO2 public-key authentication, TOTP codes, or standard passwords. All generally follow the same pattern where at the conclusion of the registration protocol:

- the log service holds an encryption of the relying party's identity under a key that only the client knows,
- the log service and client jointly hold the account's authentication secret using two-out-of-two *secret sharing* [453],
- the relying party is unaware of larch and holds the usual information necessary to verify account access: an ECDSA public key (for FIDO2), an HMAC secret key (for TOTP), or a password hash (for password-based login), and
- the log service learns nothing about the identity of the relying party.

By splitting the user's authentication secret between the client and the log, we ensure that the log service participates in all of the user's authentication attempts, which allows the log service to guarantee that every authentication attempt is correctly logged.

The underlying authentication mechanisms (FIDO2, TOTP, and password-based login) only provide security for a given relying party if the user's device was uncompromised at the time of registration; larch provides the same guarantees.

Step 3: Authentication to a relying party. Registering with a relying party lets the user later authenticate to that relying party (Figure 1). At the conclusion of an authentication operation, larch must ensure that:

- authentication succeeds at the relying party,
- the log service holds a record of the authentication attempt that *includes the name of the relying party*, encrypted under the archive key known only to the client, and
- the log service learns *no information* about the identity of the relying party involved.

The technical challenge here is guaranteeing that a compromised client cannot successfully authenticate to a relying party without creating a valid log record. In particular, the log service must verify that the log record contains a valid encryption of the relying party's name under the archive key *without* learning anything about the relying party's identity.

To achieve these goals, we design *split-secret authentication protocols* that allow the client and log to use their split authentication secrets to jointly produce an authentication credential. Our split-secret authentication protocols are essentially special-purpose two-party computation protocols [526]. In a two-party computation, each party holds a secret input, and the protocol allows the parties to jointly compute a function on their inputs while keeping each party's input secret from the other. Our split-secret authentication protocols follow a general pattern, although the specifics depend on the underlying authentication mechanism in use (FIDO2, TOTP, or password-based login):

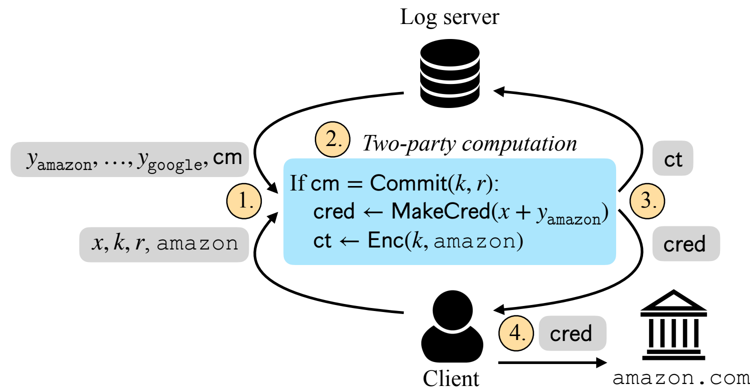


Figure 1: The client and log service run split-secret authentication where the client obtains the credential for `amazon.com` and the log service obtains an encryption of `amazon.com` under the client's key. The client's inputs are its share x of the authentication secret, the archive key k, a random nonce r, and the string `amazon.com`. The log's inputs are its shares $y_{\mathtt{amazon}}, \ldots, y_{\mathtt{google}}$ of all the client's authentication secrets and the commitment cm to the archive key generated at enrollment. The MakeCred function takes extra inputs for FIDO2 and TOTP.

- The client algorithm takes as input the identity of the relying party, the client's share of the corresponding authentication secret, the archive key, and the opening for the log service's commitment to the archive key.
- The log algorithm takes as input its shares of authentication secrets and the client's commitment to the archive key (which it received at enrollment).
- The client algorithm outputs an authentication credential: a signature (for FIDO2), an HMAC code (for TOTP), or a password (for password-based login).
- The log algorithm outputs an encryption of the relying party identifier under the archive key.

In this way, the client and log service jointly generate authentication credentials while guaranteeing that every successful authentication is correctly logged. The client and log do not learn any information beyond the outputs of the computation. We use this general pattern to construct split-secret authentication protocols for FIDO2 (Section 6.3), TOTP (Section 6.4), and password-based login (Section 6.5).

Step 4: Auditing with the log. Finally, at any time, the user can ask the log service for its collection of log entries encrypted under the archive key. A user could do this when she suspects that an attacker has compromised her credentials. The user's client could also perform this auditing in the background and notify the user if it ever detects anomalous behavior. The client uses the encryption key it generated during enrollment to decrypt log entries.

6.2.3 System goals

We now describe the security goals of larch (Figure 2).

Goal 1: Log enforcement against a malicious client. Say that an honest client enrolls with an honest

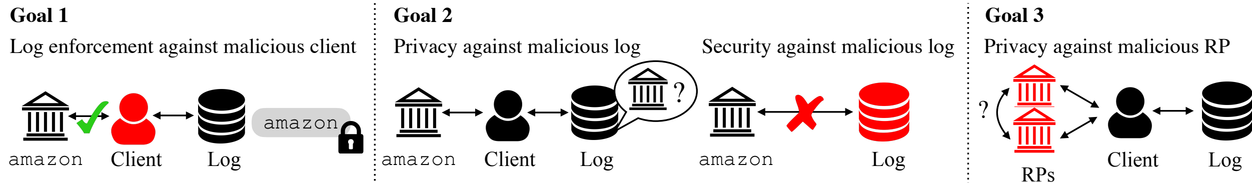


Figure 2: Larch security goals.

log service and then registers with a set of relying parties. Later on, an attacker compromises the client's secrets (e.g., by compromising one of the user's devices and causing it to behave maliciously). Every successful authentication attempt that the attacker makes using credentials managed by larch will appear in the client's authentication log stored at the larch log service. Furthermore, the honest client can decrypt these log entries using its secret key.

Goal 2: Client privacy and security against a malicious log. Even if the log service deviates arbitrarily from the prescribed protocol, it learns no information about (a) the client's authentication secrets (meaning that the log service cannot authenticate on behalf of the client) or (b) which relying parties a client has interacted with.

Goal 3: Client privacy against a malicious relying party. A set of colluding malicious relying parties learn no information about which registered accounts belong to the same client. That is, relying parties cannot link a client across multiple relying parties using information they learn during registration or authentication.

To be usable in practice, larch should additionally achieve the following functionality goal:

Goal 4: No changes to the relying party. Relying parties that support FIDO2 (U2F), TOTP, or password authentication do not need to be aware of larch. Clients can unilaterally register authentication credentials such that all future authentications are logged in larch.

6.2.4 Non-goals and extensions

Availability against a compromised log service. Larch does not provide availability if the log service refuses to provide service. We discuss defenses against availability attacks in Section 6.6.

Privacy against colluding log and relying party. If the log service colludes with a relying party, they can always use timing information to map log entries to authentication requests. Therefore, larch makes no effort to obscure the relationship between private messages seen by the two parties and only guarantees privacy when the relying party and log service do not collude.

Limitations of underlying authentication schemes. Larch provides security guarantees that match the security of the underlying authentication schemes. FIDO2 provides the strongest security, followed by TOTP, and then followed by passwords. For TOTP and password-based login, larch provides no protection against *credential breaches*: if an attacker steals users' authentication secrets (MAC keys or passwords) from the relying party, the attacker can use those secrets to authenticate without those authentications appearing in the log. FIDO2 defends against credential breaches because the relying party only ever sees the client's public key.

Larch does protect against *device compromise* for all three authentication mechanisms: even if an attacker gains control of a user's device, generating any of the user's larch-protected credentials requires communicating with the log service and results in an archived log record. If the user discovers the device break-in later on, she can recover from the log a list of authentications and take steps to remediate the effects of compromise (contacting the affected relying parties, etc.).

An attacker who compromises an account can often disable two-factor authentication or add its own credentials to a compromised account. Therefore, only an attacker's first successful access to a given relying party is guaranteed to be archived in larch. That said, many relying parties send out notifications, require step-up authentication, or revoke access to logged in clients on credential updates, all of which could complicate an attack or alert legitimate users to a problem. Hence, it is valuable to ensure that all accesses with the original account credentials are logged. Larch can make this guarantee for FIDO2, where every authentication requires a unique two-party signature. It does not provide this guarantee with passwords, as the attacker learns the password as part of the authentication process: only the attacker's first authentication to a given relying party will be logged. With TOTP, each generated code produces a larch log record. Some relying parties implement a TOTP replay cache, in which case one code allows one login. Other relying parties allow a single TOTP code to be used for arbitrarily many authentications in a short time period (generally about a minute).

Fortunately, when recovering from compromise, a user is most interested in learning whether an attacker has accessed an account zero times or more than zero times. For larch-generated credentials, users will always be able to learn this information from the larch log. However, if users import passwords that are not unique into larch, this guarantee does not hold. By default, the larch client software generates a unique random password for every relying party, but it also allows user to import existing legacy passwords, which might not be unique. In the event of password reuse, the attacker can generate a single log record to obtain the password and then use it to authenticate to all affected relying parties.

6.3 Logging for FIDO2

6.3.1 Background

FIDO2 protocol. The FIDO2 protocol [179, 506] allows a client to authenticate using cryptographic keys stored on a device (e.g., a Yubikey hardware token or a Google passkey). To register with a relying party (e.g., `github.com`), the client generates an ECDSA keypair, stores the secret key, and sends the public key to the relying party. When the client subsequently wants to authenticate to relying party `github.com`, Github's server sends the client a random challenge. The client then signs the hash of the string `github.com` and the Github-chosen challenge using the secret key the client generated for `github.com` at registration. If the signature is valid, the Github server authorizes the client. Because the message signed by the client is bound to the name `github.com`, FIDO2 provides a strong defense against phishing attacks. The FIDO2 protocol supports passwordless, second-factor, and multi-factor authentication.

Zero-knowledge arguments. Informally, zero-knowledge arguments allow a prover to convince a verifier that a statement is true without revealing *why* the statement is true [208]. More precisely, we consider non-interactive zero-knowledge argument systems [66, 178] in the random-oracle model [53]. Both the prover and verifier hold the description of a computation C and a public input x. The prover's goal is to produce a proof π that convinces the verifier that there exists a witness w that causes $C(x, w) = 1$, without revealing the witness w to the verifier. We require the standard notions of completeness, soundness, and zero knowledge [66, 208]. Throughout the chapter, we will refer to this type of argument system as a "zero-knowledge proof."

We use the ZKBoo protocol [106, 200, 268] for proving statements about computations expressed as Boolean circuits. Our system could also be instantiated with succinct non-interactive arguments of knowledge, which would decrease proof size and verification time, but at the cost of increasing proving time and requiring large parameters generated via a separate setup algorithm [65, 196, 217, 400].

Threshold signatures. A two-party threshold signature scheme [154, 155] is a set of protocols that allow two parties to jointly generate a single public key along with two shares of the corresponding secret key and then jointly sign messages using their secret key shares such that the signature verifies under the joint public key. Informally, no malicious party should be able to subvert the protocols to extract another party's share of the secret key or forge a signature on a message other than the honest party's message. We would ideally instantiate our system using BLS multisignatures [67]. Unfortunately, the predominant signing algorithms for FIDO2 are ECDSA and RSASSA [4, 12, 368]. For backwards-compatibility, we present a construction for two-party ECDSA signing with preprocessing tailored to our setting in Section 6.3.3.

6.3.2 Split-secret authentication

We now describe our split-secret authentication protocol for FIDO2 where the authentication secret is split between the larch client software and the log service. The key challenge is achieving log enforcement and log privacy simultaneously: every successful authentication should result in a valid log entry encrypting the identity of the relying party, but the log should not learn the identity of the relying party.

We use threshold signing to ensure that both the client and log participate in every successful authentication. A natural way to use threshold signing would be to have the client and log each generate a new threshold signing keypair at every registration. Unfortunately, if the log service used a different key share for each relying party, it would know which authentication requests correspond to the same relying party, violating Goal 2 (privacy against a malicious log). Instead, we have the log use the *same signing-key share for all relying parties*. The client still uses a different signing-key share per party, ensuring the public keys are unlinkable across relying parties. To authenticate to a relying party with identifier id and challenge chal, the client computes a digest $\mathsf{digest} = \mathsf{Hash}(\mathsf{id}, \mathsf{chal})$ that hides id. The client and log then jointly sign digest.

We also need to ensure that the log service obtains a correct record of every authentication. In particular, the log should only participate in threshold signing if it obtains a valid encryption ct of the relying-party identifier id [466].

To be valid, a ciphertext ct must (1) decrypt to id under the archive key k established for that client, and (2) be correctly related to the digest digest that the log will sign (i.e., $\mathsf{Dec}(k, \mathsf{ct}) = \mathsf{id}$ and $\mathsf{digest} = \mathsf{Hash}(\mathsf{id}, \mathsf{chal})$). To allow the log service to check that the client is using the right archive key without learning the key, we use a commitment scheme. During enrollment, the client generates a commitment cm to the archive key k using random nonce r and sends cm to the log service. During authentication, the client uses a zero-knowledge proof to prove to the log that it knows a key k, randomness r, relying-party identifier id, and authentication challenge chal such that ciphertext ct, digest digest, and commitment cm from enrollment meet the following conditions:

(a) $\mathsf{cm} = \mathsf{Commit}(k, r)$,

(b) $\mathsf{id} = \mathsf{Dec}(k, \mathsf{ct})$, and

(c) $\mathsf{digest} = \mathsf{Hash}(\mathsf{id}, \mathsf{chal})$.

The public inputs are the ciphertext ct, digest digest, and commitment cm (known to the client and log); the witness is the archive key k (known only to the client), commitment opening r, relying-party identifier id, and challenge chal.

Final protocol. We now outline our final protocol.

Enrollment. During enrollment, the client samples a symmetric encryption key k as the archive key and commits to it with some random nonce r. The client sends the commitment cm to the log, and the log generates a signing-key share for the user. The log sends the client the public key corresponding to its signing-key share to allow the client to derive future keypairs for relying parties.

Registration. At registration, the client generates a new signing-key share for that relying party. The client then aggregates the log's public key with its new signing-key share and sends the resulting public key to the relying party. No interaction with the log service is required.

Authentication. To authenticate to id with challenge chal, the client computes $\mathsf{digest} \leftarrow \mathsf{Hash}(\mathsf{id}, \mathsf{chal})$ and $\mathsf{ct} \leftarrow \mathsf{Enc}(k, \mathsf{id})$. The client then generates a zero-knowledge proof π that it knows an archive key k, commitment nonce r, relying-party identifier id, and authentication challenge chal such that digest and ct are correctly related relative to the commitment cm that the client generated at enrollment. The client sends digest, ct, and π to the log service. The log service checks the proof and, if it verifies, runs its part of the threshold signing protocol. The log service stores ct and returns its signature share to the client. The log service also stores the current time and client IP address with ct, allowing the user to obtain additional metadata by auditing. Finally, the client completes the threshold signature and sends it to the relying party.

Auditing. To audit the log, the client requests the list of ciphertexts and metadata from the log service and decrypts all of the relying-party identifiers.

6.3.3 Two-party ECDSA with preprocessing

Section 6.3.2 shows how to implement larch for any two-of-two threshold signing scheme that cryptographically hashes input messages. However, FIDO2 compatibility forces us to use ECDSA, which is more cumbersome than BLS to threshold. We present a concretely efficient protocol for ECDSA signing between the client and log.

There is a large body of prior work on multi-party ECDSA signing [18, 90, 92, 131, 132, 160, 197, 198, 220, 325]. However, existing protocols are orders of magnitude more costly than the one we present here [90, 92, 197, 198, 325]. The efficiency gain for us comes from the fact that we may assume that the client is *honest at enrollment time and only later compromised*. In contrast, standard schemes for two-party ECDSA signing must protect against the compromise of either party at any time. Prior protocols provide this stronger security property at a computational and communication cost. In our setting, we need only ensure that an honest client can run an enrollment procedure with the log service such that if the client is later compromised, the attacker cannot subvert the signing protocol.

We leverage the client to split signing into two phases:

1. During an *offline phase*, which takes place during enrollment, the client performs some preprocessing to produce a "presignature." Security only holds if the client is honest during the offline phase.
2. During an *online phase*, which takes place during authentication, the client and log service use the presignature to perform a lightweight, message-dependent computation to produce an ECDSA signature. Security holds if either the client or log service is compromised during the online phase.

Prior work also splits two-party signing into an offline and online phase. However, prior work performs this partitioning to reduce the online time at the expense of a more costly offline phase [90, 131, 132, 521]. (The offline phase in these schemes is expensive since the protocols do not assume that both parties are honest during the offline phase.) We split the signing scheme into an offline and online phase to take advantage of the fact that we may assume that the client is honest in the offline phase and so can reduce the total computation time this way.

An additional requirement in our setting is that the log should *not* learn the public key that the signature is generated under. Because the public key is specific to a relying party, hiding the public key is necessary for ensuring that the log cannot distinguish between relying parties. The signing algorithm can take as input a relying-party-specific key share from the client and a relying-party-independent key share from the log.

Background: ECDSA. For a group $\mathbb{G}$ of prime order q with generator g, fixed in the ECDSA standard, an ECDSA secret key is of the form $\mathsf{sk} \in \mathbb{Z}_q$, where $\mathbb{Z}_q$ denotes the ring of integers modulo q. The corresponding ECDSA public key is $\mathsf{pk} = g^{\mathsf{sk}} \in \mathbb{G}$. ECDSA uses a hash function $\mathsf{Hash}\colon \{0,1\}^* \to \mathbb{Z}_q$ and a "conversion" function $f : \mathbb{G} \to \mathbb{Z}_q$. To generate an ECDSA signature on a message $m \in \{0,1\}^*$ with secret key $\mathsf{sk} \in \mathbb{Z}_q$, the signer samples a signing nonce $r \xleftarrow{R} \mathbb{Z}_q$ and computes

$$r^{-1} \cdot (\mathsf{Hash}(m) + f(g^r) \cdot \mathsf{sk}) \quad \in \mathbb{Z}_q.$$

Our construction. We now describe our construction for a two-party ECDSA signing protocol with presignatures. (See Section 6.12 for technical details.) To generate the log keypair, the log samples $x \xleftarrow{R} \mathbb{Z}_q$, sets its secret key to $x \in \mathbb{Z}_q$, and sets its public key to $X = g^x \in \mathbb{G}$. Then to generate a keypair from the log public key, the client samples $y \xleftarrow{R} \mathbb{Z}_q$ and sets the relying-party-specific public key to $\mathsf{pk} = X \cdot g^y \in \mathbb{G}$. For each public key of the form $g^{x+y} \in \mathbb{G}$, the log has one share $x \in \mathbb{Z}_q$ of the secret

key that is the same for all public keys and the client has the other share $y \in \mathbb{Z}_q$ of the secret key that is different for each public key.

We split the signature-generation process into two parts:

1. *Offline phase*: a message-independent, key-independent "presignature" algorithm that the client runs, and
2. *Online phase*: a message-dependent, key-dependent signing protocol that the log and client run jointly.

To generate the presignature in the offline phase, the client samples a signing nonce $r \xleftarrow{R} \mathbb{Z}_q$, computes $R \leftarrow g^r \in \mathbb{G}$, and splits r^{-1} into additive secret shares: $r^{-1} = r_0 + r_1 \in \mathbb{Z}_q$. The log's portion of the presignature is $(f(R), r_0) \in \mathbb{Z}_q^2$, and the client's portion is $(f(R), r_1) \in \mathbb{Z}_q^2$. Then, to produce a signature on a message in the online phase, the client and log simply perform a single secure multiplication to compute

$$r^{-1} \cdot (\mathsf{Hash}(m) + f(R) \cdot \mathsf{sk}) \in \mathbb{Z}_q$$

where $r^{-1} \in \mathbb{Z}_q$ (signing nonce) and $\mathsf{sk} \in \mathbb{Z}_q$ (signing key) are secret-shared between the client and log.

To perform this multiplication over secret-shared values, we use Beaver triples [48]. A Beaver triple is a set of one-time-use shares of values that the log and client can use to efficiently perform a two-party multiplication on secret-shared values. Traditionally, generating Beaver triples is one of the expensive portions of multiparty computation protocols (e.g., in prior work on threshold ECDSA [131]). In our setting, the client at enrollment time can generate a Beaver triple as part of the presignature. Note that the client and log can use each signing nonce and Beaver triple exactly once. That is, the client and log must use a fresh presignature to generate each signature.

Malicious security. By deviating from the protocol, neither the client nor the log should be able to learn secret information (i.e., the other party's share of the secret key or signing nonce) or produce a signature for any message apart from the one that the protocol fixes. We describe how to accomplish this using traditional tools for malicious security (e.g. information-theoretic MACs [133]) in Section 6.12.

Formalizing and proving security. We define and prove security in Section 6.11 and Section 6.12.

Implications for system design. Our preprocessing approach increases the client's work at enrollment: the client generates some number of presignatures (e.g., 10K) and sends the log's presignature shares to the log. To reduce storage burden on the log, the client can store encryptions of the log's presignature shares.

When the client is close to running out of presignatures, it can authenticate with the log, generate more presignatures, and send the log's presignature shares to the log service. If the log service does not receive an objection after some period of time, it will start using the new presignatures. An honest client periodically checks the log to see whether any unexpected presignatures (created by an attacker) appear in its log. If the client learns that a new batch of presignatures was generated that the client did not authorize, the client authenticates to the log service and objects. This approach provides security as long as an honest client can detect client compromise and object to any adversarially generated presignatures during the objection period. (If a user is concerned about

recovering authentication logs from a long period of undetected compromise, the client can set a very long objection period, or generate enough presignatures at enrollment that, with high probability, it will not need to generate more.)

If the client runs out of presignatures and the log service rejects the client's presignatures, the client and the log can temporarily use a more expensive signing protocol that does not require presignatures [160, 198, 325, 521]. The client could run out of presignatures and be forced to use the slow multisignature protocol in the following cases:

1. The attacker compromised the user's credentials with the log service, allowing the attacker to object to the new presignatures. In this case, the attacker could change the user's credentials and permanently lock the user out of her account.
2. The honest client was close to running out of presignatures, generated new presignatures, and then ran out of presignatures while waiting for a possible objection. This scenario only occurs when the honest client makes an unexpectedly large number of authentications in a short period of time. The client only needs to pay the cost of the slow multisignature protocol for a short period of time.

An attacker that has compromised the log service can also deny service, as we discussed in Section 6.2.4.

Benefits of future support for Schnorr-based signing. The FIDO2 standard recommends support for EdDSA, which, if widely supported in the future, could simplify the two-party signing protocol and avoid preprocessing altogether. Adapting a Schnorr-based threshold signing protocol [182, 351, 379] to the setting where only the client knows the message and public key could potentially improve performance.

6.4 Logging for time-based one-time passwords

We now show how larch can support time-based one-time passwords (TOTP).

6.4.1 Background: TOTP

TOTP is a popular form of second-factor authentication that authenticator apps (Authy, Google Authenticator, and others [369]) implement. When a client registers for TOTP with a relying party, the relying party sends the client a secret cryptographic key. Then, to authenticate, the client and the relying party both compute a MAC on the current time using the secret key from registration. The client sends the resulting MAC tag to the server. If the client's submitted tag matches the one that the server computes, the relying party authorizes the client. TOTP uses a hash-based MAC (HMAC).

6.4.2 Split-secret authentication for TOTP

At a high level, in our split-secret authentication protocol for TOTP, both the client and log service have as private input additive secret shares of the TOTP secret key. At the conclusion of the

split-secret authentication, the client holds a TOTP code and the log service holds a ciphertext. We now give the details of our protocol.

Enrollment. At enrollment, just as with FIDO2, the client generates and stores a long-term symmetric-encryption archive key k and random nonce r. Then, the client sends the commitment $\mathsf{cm} = \mathsf{Commit}(k, r)$ to the log service.

Registration. To register a client, a relying party generates and sends the client a secret MAC key $\mathsf{k_{id}}$ for TOTP. The client samples a random identifier id for the relying party and then splits the TOTP secret key $\mathsf{k_{id}}$ into additive secret shares $\mathsf{klog_{id}}$ and $\mathsf{kclient_{id}}$. The client sends $(\mathsf{id}, \mathsf{klog_{id}})$ to the log service and locally stores $(\mathsf{id}, \mathsf{kclient_{id}})$ alongside a name identifying the relying party (e.g., `user@amazon.com`).

Authentication. In order to authenticate to the relying party id at time t, the client needs to compute $\mathsf{HMAC}(\mathsf{k_{id}}, t)$ with the help of the log service. Let n be the number of relying parties with which the client has registered. To authenticate, the client and log service run a secure two-party computation where:

- The client's input is its long-term symmetric archive key k and commitment opening r from enrollment, the relying-party identifier id, and the client's share of the TOTP key $\mathsf{kclient_{id}}$.
- The log service's input is the commitment cm from enrollment, the list of relying-party identifiers that the client has registered with $(\mathsf{id}_1, \ldots, \mathsf{id}_n)$, and the log service's TOTP key shares $(\mathsf{klog_{id_1}}, \ldots, \mathsf{klog_{id_n}})$—one per relying party.
- The client outputs the TOTP code $\mathsf{HMAC}(\mathsf{k_{id}}, t)$.
- The log outputs an encrypted log record: an encryption of the relying-party identifier id under the archive key k.

We execute this two-party computation using an off-the-shelf garbled-circuit-based multiparty computation protocol. Garbled circuits allow two parties to jointly execute any Boolean circuit on private inputs, where neither party learns information about the other's input beyond what they can infer from the circuit's output [525]. We use the protocol from Wang et al. [511], which provides malicious security, meaning that the protocol remains secure even if one corrupted party deviates arbitrarily from the protocol. As long as either the client or the log service is honest, the log service does not learn any information about the client's authentication secrets, and the client learn no information about the TOTP secret, apart from the single TOTP code that the protocol outputs. Because we use an off-the-shelf garbled-circuit protocol, the communication overhead is much higher than in the special-purpose protocols we design for FIDO2 and passwords (Section 6.8). TOTP is challenging to design a special-purpose protocol for because the authentication credential must be generated via the SHA hash function which, unlike the authentication credentials for FIDO2 and passwords, does not have structure we can exploit. Clients can ask the log service to delete registrations for unused accounts to speed up the two-party computation.

Auditing. To audit the log, the client simply requests the list of ciphertexts from the log service. The client decrypts each ciphertext with its archive key k and then, using its mapping of id values to relying party names, outputs the resulting list of relying party names.

6.5 Logging for passwords

We now describe how larch can support passwords.

6.5.1 Protocol overview

We construct a split-secret authentication protocol that takes place between the client and the log service. In particular, we show how the client can compute the password to authenticate to a relying party in such a way that (a) the log service does not learn the relying party's identity and (b) the client's authentication attempt is logged. At the start of the authentication protocol run:

- the client holds a secret key, the log service's public key, and the identity id^* of the relying party it wants to authenticate to, and
- the log service holds its own secret key, the client's public key, and a list of relying-party identities $(\mathsf{id}_1, \dots, \mathsf{id}_n)$ at which the client has registered.

At the end of the authentication protocol run:

- the client holds a password derived as a pseudorandom function of the client's secret, the log's secret, and the relying party identity id^*, and
- the log service holds a ciphertext encrypting the relying party's identity id^* under the client's public key.

Limitations inherent to passwords. As we discussed in Section 6.2.4, larch for passwords does not protect against credential breaches, but does defend against device compromise.

6.5.2 Split-secret authentication for passwords

The larch scheme for password-based authentication uses a cyclic group $\mathbb{G}$ of prime order q with a fixed generator $g \in \mathbb{G}$. Our implementation uses the NIST P-256 elliptic-curve group.

When using password-based authentication in larch, the client and log service after registration each hold a secret share of the password for each relying party. In particular, the password for a relying party with identity $\mathsf{id} \in \{0,1\}^*$ is the string $\mathsf{pw}_{\mathsf{id}} = k_{\mathsf{id}} \cdot \mathsf{Hash}(\mathsf{id})^k \in \mathbb{G}$, where:

- $k_{\mathsf{id}} \in \mathbb{Z}_q$ is a per-relying-party secret share held by the client,
- $\mathsf{Hash}\colon \{0,1\}^* \to \mathbb{G}$ is a hash function, and
- $k \in \mathbb{Z}_q$ is a per-client secret key held by the log service.

Thus, computing $\mathsf{pw}_{\mathsf{id}}$ requires both the client's per-site key k_{id} and the log's secret key k.

The technical challenge is to construct a protocol that allows the client to compute the password $\mathsf{pw}_{\mathsf{id}}$ while (a) hiding id from the log service and (b) ensuring that the log service completes the interaction holding an encryption of id under the client's public key.

Protocol. We describe the protocol steps:

Enrollment. The client samples an ElGamal secret key $x \in \mathbb{Z}_q$ as the archive key and sends the corresponding public key $X = g^x \in \mathbb{G}$ to the log service. The log service samples a Diffie-Hellman secret key $k \in \mathbb{Z}_q$ and sends its public key $K = g^k \in \mathbb{G}$ to the client.

Registration. The client samples a per-relying-party random identifier $\mathsf{id} \xleftarrow{R} \{0,1\}^{128}$, saves id locally alongside the name of the relying party (e.g., `user@amazon.com`), and sends id to the log service. The log service saves the string $\mathsf{Hash}(\mathsf{id})$ and replies with $\mathsf{Hash}(\mathsf{id})^k \in \mathbb{G}$. To generate a new strong password $\mathsf{pw}_{\mathsf{id}}$ (the recommended use), the client samples and saves a random key share $k_{\mathsf{id}} \xleftarrow{R} \mathbb{G}$ and sets $\mathsf{pw}_{\mathsf{id}} \leftarrow k_{\mathsf{id}} \cdot \mathsf{Hash}(\mathsf{id})^k \in \mathbb{G}$. To import a legacy password $\mathsf{pw}_{\mathsf{id}}$ (less secure), the client computes and stores $k_{\mathsf{id}} \leftarrow \mathsf{pw}_{\mathsf{id}} \cdot \left(\mathsf{Hash}(\mathsf{id})^k\right)^{-1} \in \mathbb{G}$. The client then deletes $\mathsf{Hash}(\mathsf{id})^k$ and $\mathsf{pw}_{\mathsf{id}}$. Note that the log server can discard id, which it only uses to avoid providing h^k for arbitrary h. When the client samples id and k_{id} randomly in the recommended usage, the password $\mathsf{pw}_{\mathsf{id}}$ for each relying party is random and distinct.

Authentication. During authentication, the client must recompute the password $\mathsf{pw}_{\mathsf{id}}$. To do so, the client first sends the log service an encryption of $\mathsf{Hash}(\mathsf{id})$ under the public ElGamal archive key g^x: the client samples $r \xleftarrow{R} \mathbb{Z}_q^*$ and computes the ciphertext $(c_1, c_2) = (g^r, \mathsf{Hash}(\mathsf{id}) \cdot g^{xr}) \in \mathbb{G}^2$. In addition, the client sends a zero-knowledge proof to the log service attesting to the fact that (c_1, c_2) is an encryption under the client's public key X of $\mathsf{Hash}(\mathsf{id})$ for $\mathsf{id} \in \{\mathsf{id}_1, \mathsf{id}_2, \dots, \mathsf{id}_n\}$—the set of relying-party identifiers that the client sent to the log service during each of its registrations so far. The client executes this proof using the technique from Groth and Kohlweiss [218]. The proof size is $O(\log n)$ and the prover and verifier time are both $O(n)$. (See Section 6.13 for implementation details.)

The log service saves the ciphertext as a log entry, checks the zero-knowledge proof, and returns the value $h = c_2^k = \mathsf{Hash}(\mathsf{id})^k \cdot g^{xrk} \in \mathbb{G}$ to the client. The client can then compute

$$\mathsf{pw}_{\mathsf{id}} = k_{\mathsf{id}} \cdot h \cdot K^{-xr} = k_{\mathsf{id}} \cdot \mathsf{Hash}(\mathsf{id})^k \quad \in \mathbb{G}.$$

Crucially, the client deletes $\mathsf{pw}_{\mathsf{id}}$ after authentication to ensure that future authentications must again interact with the log service.

Auditing. To audit the log, the client downloads the ElGamal ciphertexts and can decrypt each ciphertext to recover a list of hashed identities: $(\mathsf{Hash}(\mathsf{id}_1), \mathsf{Hash}(\mathsf{id}_2), \dots)$. The client uses its stored mapping of ids to relying-party identifiers to recover the plaintext names of the relying parties in the log.

6.6 Protecting against log misbehavior

The larch log service must participate in each of the user's authentication attempts. If the log service goes offline, the user will not be able to authenticate to any of her larch-enabled relying parties. In a real-world deployment, the log service could consist of multiple servers replicated using standard state-machine replication techniques to tolerate benign failures [306, 384]. However, users might also worry about intentional denial-of-service attacks on the part of the log.

To defend against availability attacks, a user can split trust across multiple logs. At enrollment time, the user can enroll with n logs. Then at registration, the user can set a threshold t of logs that must participate in authentication. Thus, the user can authenticate to her accounts so long as t logs are online, and she can audit activity so long as $n - t + 1$ logs are available. We need $n - t + 1$ logs

to be available for auditing in order to guarantee that at least one of the t logs that participated in authentication is online. To ensure that colluding logs cannot authenticate on behalf of a client, the user's client can run $n+1$ logical parties, and $n+t+1$ parties can generate an authentication credential. In the setting with multiple log services, we need to adapt our two-party protocols to threshold multi-party protocols. Although we present our techniques for two parties (the client and a single log), our techniques generalize to multiple parties in a straightforward way.

For FIDO2 and passwords, the client now sends a zero-knowledge proof to each of the n logs. In the password case, the client can then retrieve (t,n) Shamir shares of the password [453], and in the FIDO2 case, the client can run any existing multi-party threshold signing protocol that does not take the public key as input [131, 457]. For TOTP, the client and the n logs can execute the same circuit using any malicious-secure threshold multi-party computation protocol [55].

Note that for relying parties that support FIDO2, users can optionally register a backup hardware FIDO2 device to allow them to bypass the log. In this case, the user can authenticate either via larch or via her backup FIDO2 key. While registering a backup hardware device protects against availability attacks, if an attacker obtains this hardware device, they can authenticate as the user without interacting with the log.

6.7 Implementation

We implemented larch for FIDO2, TOTP, and passwords with a single log service. We use C/C++ with gRPC and OpenSSL with the P256 curve (required by the FIDO2 standard). We wrote approximately 5,700 lines of C/C++ and 50 lines of Javascript (excluding tests and benchmarks). Our implementation is available at `https://github.com/edauterman/larch`.

For our FIDO2 implementation, we implemented a ZKBoo [200] library for arbitrary Boolean circuits. Our ZKBoo implementation (with optimizations from ZKB++ [106]) uses emp-toolkit to support arbitrary Boolean circuits in Bristol Fashion [510]. To support the parallel repetitions required for soundness error $< 2^{-80}$, we use SIMD instructions with a bitwidth of 32 and run 5 threads in parallel. For the proof circuit, we use AES in counter mode for encryption and SHA-256 for commitments (SHA-256 is necessary for backwards compatability with FIDO2). We built a log service and client that invoke the ZKBoo library, as well as a Chrome browser extension that interfaces with our client application and is compatible with existing FIDO2 relying parties. We built our browser extension on top of an existing extension [297].

Our TOTP implementation uses a maliciously secure garbled-circuit construction [511] implemented in emp-toolkit [510]. We generated our circuit using the CBMC-GC compiler [184] with ChaCha20 for encryption and SHA-256 for commitments.

For our passwords implementation, we implemented Groth and Kohlweiss's proof system [218].

Our implementation uses a single log server for the log service, does not encrypt communication between the client and the log service, and does not require the client to authenticate with the log service. A real-world deployment would use multiple servers for replication, use TLS between the client and the log service, and authenticate the client before performing any operations.

Optimizations. We use pseudorandom generators (PRGs) to compress presignatures: the log stores 6 elements in $\mathbb{Z}_q$ and the client stores 1 element. Also, instead of running an authenticated encryption scheme (e.g. AES-GCM) inside the circuit for FIDO2 or TOTP, we run an encryption scheme without authentication (e.g. AES in counter mode) inside the circuit and then sign the ciphertext (client has the signing key, log has the verification key). The log can check the integrity of the ciphertext by verifying the signature, which is much faster than checking in a zero-knowledge proof or computing the ciphertext tag jointly in a two-party computation.

6.8 Evaluation

In this section, we evaluate the cost of larch to end users and the cost of running a larch log service.

Experiment setup. We run our benchmarks on Amazon AWS EC2 instances. Unless otherwise specified, we run the log service on a c5.4xlarge instance with 8 cores and 32GiB of memory and, for latency benchmarks, the client on a c5.2xlarge instance with 4 cores and 16GiB of memory, comparable to a commodity laptop (2 hyperthreads per core throughout). We configure the network connection between the client and log service to have a 20ms RTT and a bandwidth of 100 Mbps.

6.8.1 End-user cost

We show larch authentication latency and communication costs for FIDO2, TOTP, and passwords.

FIDO2.

Latency. The client for our FIDO2 scheme can complete authentication in 303ms with a single CPU core, or 117ms when using eight cores (Figure 3). Loading a webpage often takes a few seconds because of network latency, so the client cost of larch authentication is minor by comparison. The client's running time during authentication is independent of the number of relying parties. The heaviest part of the client's computation is proving to the log service that its encrypted log entry is well formed.

At enrollment, the client must generate many "presignatures," which it later uses to run our authentication protocol with the log. Generating 10,000 presignatures for 10,000 future FIDO2 authentications takes 885ms. When the client runs out of presignatures, it generates new presignatures it can use after a waiting period (see Section 6.3.3).

Communication. During enrollment, the client must send the log 1.8MiB worth of presignatures. Thereafter, each authentication attempt requires 1.73MiB worth of communication: the bulk of this consists of the client's zero-knowledge proof of correctness, and 352B of it comes from the signature protocol. By using a different zero-knowledge proof system, we could reduce communication cost at the expense of increasing client computation cost.

Comparison to existing two-party ECDSA. For comparison, a state-of-the-art two-party ECDSA protocol [521] that does not require presignatures from the client and uses Paillier requires 226ms of computation at signing time (the authors' measurements exclude network latency, which we estimate would add 60ms) and 6.3KiB of per-signature communication. Using a variant of the

same protocol based on oblivious transfer [521] makes it possible to reduce computation to 2.8ms (again, excluding network latency, which we estimate would add 60ms) at the cost of increasing per-signature communication to 90.9KiB. In contrast, our signing protocol only requires 0.5KiB per-signature communication (including the log presignature and the signing messages) and takes 61ms time at signing, almost all of which is due to network latency and can be run in parallel with proving and verifying as the computational overhead is minimal (roughly 1ms).

TOTP.

Latency. In Figure 3 (right), we show how TOTP authentication latency increases with the number of relying parties the user registered with. Because we implement TOTP authentication using garbled circuits [511], we can split authentication into two phases: an "offline", input-independent phase and an "online", input-dependent phase (the log service and client communicate in both phases). Both phases are performed once per authentication. However, the offline phase can be performed in advance of when the user needs to authenticate to their account, and so it does not affect the latency that the user experiences. For 20 relying parties, the online time is 91ms and the offline time is 1.23s. For 100 relying parties, the online time is 120ms and the offline time is 1.39s.

Communication. Communication costs for our TOTP authentication scheme are large: for 20 relying parties, the total communication cost is 65MiB, and for 100 relying parties, the total communication is 93MiB. The online communication costs are much smaller: for 20 relying parties, the online communication is 202KiB and for 100 relying parties, the online communication is 908KiB. We envision clients running the offline phase in the background while they have good connectivity. While these communication costs are much higher than those associated with FIDO2 or passwords, we expect users to authenticate with TOTP less frequently because TOTP is only used for second-factor authentication.

Passwords.

Latency. In Figure 3 (center), we show how password authentication latency increases with the number of registered relying parties. With 16 relying parties, authentication takes 28ms, and with 512 relying parties, it takes 245ms: the authentication time grows linearly with the number of relying parties. The proof system we use requires padding the number of relying parties to the nearest power of two, meaning that registering at additional relying parties does not affect the latency or communication until the number of relying parties reaches the next power of two.

Communication. In Figure 5, we show how communication increases logarithmically with the number of relying parties. This behavior is due to the fact that proof size is logarithmic in the number of relying parties. With 16 relying parties, the communication is 1.47KiB, and with 512 relying parties, it is 4.14KiB.

6.8.2 Cost to deploy a larch service

If successful, larch can become much simpler and more efficient with a little support from future FIDO specifications (see Section 6.9). Nonetheless, we show larch is already practical by analyzing the cost of deploying a larch service today (Table 6). We expect a larch log service to perform many password-based authentications, some FIDO2 authentications, and a comparatively small number of

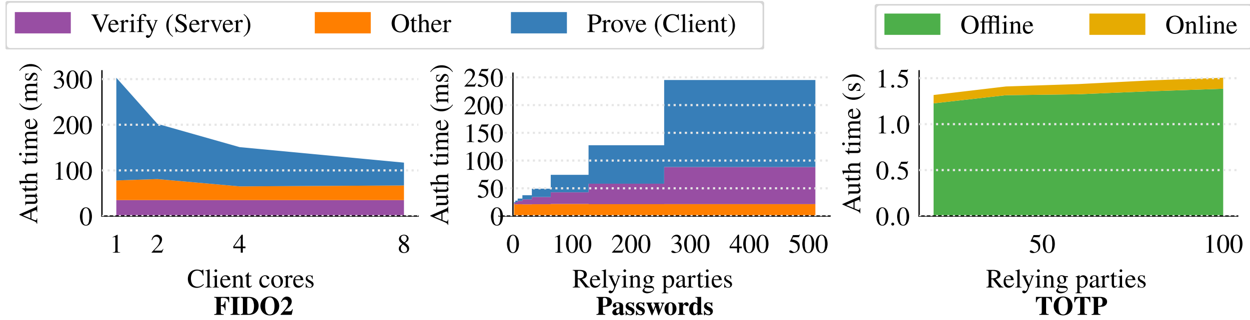


Figure 3: On the left, larch FIDO2 latency decreases as the number of client cores increases (latency is independent of the number of relying parties). In the center, larch password latency grows with the number of relying parties, with the majority of the time spent on client proof generation. On the right, larch TOTP latency grows with the number of relying parties, with the majority of the time spent in an input-independent "offline" phase as opposed to the input-dependent "online" phase (both phases require network communication).

TOTP authentications. This is because the majority of relying parties only support passwords, and relying parties typically require second-factor authentication only from time to time.

Throughout this section, we consider password-based authentication with 128 relying parties (based on the fact that the average user has roughly 100 passwords [439]) and TOTP-based authentication with 20 relying parties (based on the fact that Yubikey's maximum number of TOTP registrations is 32 [13]). The authentication overhead of FIDO2 in larch is independent of the number of relying parties the user has registered with.

Storage. For each of the three protocols, the log service must store authentication records (timestamp, ciphertext, and signature). Authentication records are 104B for FIDO2, 88B for TOTP, and 138B for passwords (differences are due to ciphertext size). The FIDO2 protocol additionally requires the client to generate presignatures for the log, each of which is 192B. For 10K presignatures, the log service must store 1.83MiB. In Figure 4 (left), we show how per-client log storage actually decreases as presignatures are consumed and replaced by authentication records. To minimize storage costs, the log service can encrypt its presignatures and store them at the client. The log service then simply needs to keep a counter to prevent presignature re-use.

Throughput. In Table 6, we show the number of auths/s a single log service core can support assuming 128 passwords and 20 TOTP accounts. We achieve the highest throughput for passwords (47.62 auths/cores/s), which are the most common authentication mechanism. For FIDO2, which can be used as either a first or second authentication factor and is supported by fewer relying parties than passwords, we achieve 6.18 auths/core/s. Finally, for TOTP, which is only used as a second factor, we achieve 0.73 auths/core/s.

Our FIDO2 protocol can be instantiated with any NIZK proof system to achieve a different tradeoff between authentication latency and log service throughput. For example, we instantiate our system with ZKBoo, but could also use Groth16 [217] to reduce communication and verifier time (increasing log throughput). We measure the performance of Groth16 on our larch FIDO2 circuit on the BN-128 curve using ZoKrates [539] with libsnark [304] with a single core (we only measure

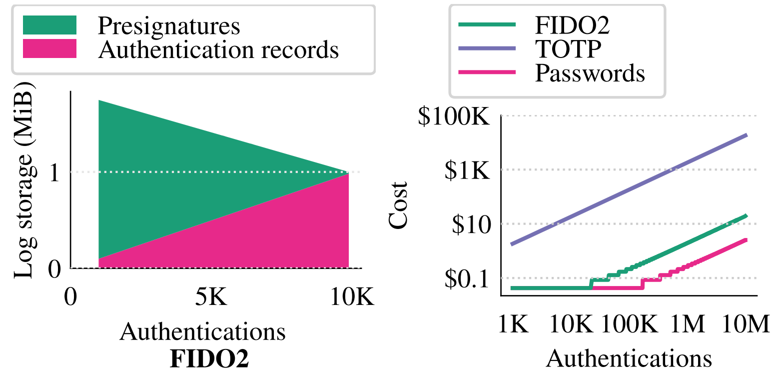


Figure 4: On the left, per-client storage overhead at the log decreases as presignatures are replaced with authentication records (client enrolls with 10K presignatures). On the right, minimum cost of supporting more authentications with passwords, (128 relying parties), FIDO2, and TOTP (20 relying parties). Both axes use a logarithmic scale.

the overhead of SHA-256, which dominates circuit cost, to provide a performance lower bound). While the verifier time is much lower (8ms) and the proof is much smaller (4.26KiB), (1) the trusted setup requires the client to store 19.86MiB and the log service to store 9.2MiB per client, and (2) the proving time is 4.07s, meaning that authentication latency is much higher.

Cost. We now quantify the cost of running a larch log service. The cost of one core on a c5 instance is $0.0425-$0.085/hour depending on instance size [2]. Data transfer to AWS instances is free, and data transfer from AWS instances costs $0.05-$0.09/GB depending on the amount of data transferred per month [2]. In Table 6, we show the cost of supporting 10M authentications for each authentication method with larch.

Supporting 10M authentications requires 450 log core hours for FIDO2, 3,832 log core hours for TOTP, and 59 log core hours for passwords. Compute for 10M authentications costs $19.13-$38.25 for FIDO2, $162.86-$325.72 for TOTP, and $2.51-$5.02 for passwords. Communication for 10M authentications costs $0.10-$0.19 for FIDO2, $17,923-$32,262 for TOTP, and $0.015-$0.027 for passwords. The high cost for TOTP is due to the large amount of communication required at authentication: the log service must send the client 36.8MiB for every authentication. In both the FIDO2 and password protocols, the vast majority of the communication overhead is due to the proof sent from the client to the log service, which incurs no monetary cost. We show how cost increases with the number of authentications for each of the the authentication methods in Figure 4 (right).

TOTP is substantially more expensive than FIDO2 or passwords. However, we expect a relatively small fraction of authentication requests to be for TOTP.

6.9 Discussion

Deployment strategy. Because larch supports passwords, TOTP, and FIDO2, people can use it with the vast majority of web services. In addition, larch offers users many of the benefits of FIDO2

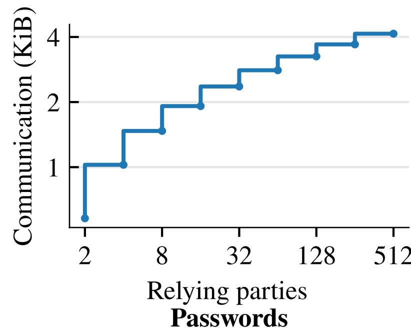


Figure 5: Communication for larch with passwords increases logarithmically with the number of relying parties (both axes use a logarithmic scale).

	FIDO2	TOTP	Password
Online auth time	150 ms	91 ms	74 ms
Total auth time	150 ms	1.32 s	74 ms
Online auth comm.	1.73 MiB	201 KiB	3.25 KiB
Total auth comm.	1.73 MiB	65 MiB	3.25 KiB
Auth record	104 B	88 B	138 B
Log presignature	192 B	∅	∅
Log auths/core/s	6.18	0.73	47.62
10M auths min cost	$19.19	$18,086	$2.48
10M auths max cost	$38.37	$32,588	$4.96

Table 6: Costs for larch with FIDO2, TOTP (20 relying parties), and passwords (128 relying parties). We take the cost of one core on a c5 instance to be $0.0425-$0.085/hour (depending on instance size) and data transfer out of AWS to cost $0.05-$0.09/GB (depending on amount of data transferred) [2]. For comparison, the Argon2 password hash function should take 0.5s using 2 cores.

without a dedicated hardware security token, particularly FIDO2's protection against phishing. The flexibility for users to choose log services can foster an ecosystem of new security products, such as log services that request login confirmation via a mobile phone app, apps that monitor the log to notify users of anomalous behavior, or enterprise security products that monitor access to arbitrary third-party services that a company could contract with.

FIDO improvements. Larch can benefit from enhancements we hope to see considered for future versions of the FIDO specification. One simple improvement would be to support BLS signatures, which are easier to threshold and so eliminate larch's need for presignatures [67].

Future versions of FIDO could also directly support secure client-side logging by allowing the relying party to compute the encrypted log record itself. The relying party could then ensure that the log service receives the correct encrypted log record by checking for the log record in the signing

payload. Specifically, the signature payload could have the form:

$$\mathsf{Hash}(\text{log-record-ciphertext}, \mathsf{Hash}(\text{remaining-FIDO-data}))\ .$$

The log server can then take the outer hash preimage as input without needing to verify anything else about the log record.

We want to allow the relying party to generate the encrypted log record without making it possible to link users across relying parties. Instead of giving the relying party the user's public key directly at registration, which would link a user's identity across relying parties, we instead give the relying party a key-private, re-randomizable encryption of the relying party's identifier (we can achieve this using ElGamal encryption). At authentication, the relying party can re-randomize the ciphertext to generate the encrypted log record.

We also hope that future FIDO revisions standardize and promote authentication metadata as part of the challenge and hypothetical log record field. For users with multiple accounts at one relying party, it would be useful to include account names as well as relying party names in signed payloads. It would furthermore improve security to allow distinct types of authentication log records for different security-sensitive operations such as authorizing payments and changing or removing 2FA on an account. An app monitoring a user's log can then immediately notify the user of such operations.

Multiple devices. Clients need to authenticate to their accounts across multiple devices, which requires synchronizing a small amount of dynamic, secret state across devices. Cross-device state could be stored encrypted at the log, or could be disseminated through existing profile synchronization mechanisms in browsers. There is a danger of the synchronization mechanism maliciously convincing two devices to use the same presignature. Therefore, presignatures should be partitioned between devices in advance, and devices should employ techniques such as fork consistency [353] to detect and deter any rollback attacks. Existing tools can help a user recover if she loses all of her devices [137, 296, 337, 507].

Enforcing client-specific policies. We can extend larch in a straightforward way to allow the log to enforce more complex policies on authentications. The client could submit a policy at enrollment time, and the log service could then enforce this policy for subsequent authentications. If the policy decision is based on public information, the log service can apply the policy directly (e.g., rate-limiting, sending push notifications to a client's mobile device). Other policies could be based on private information. For example, if we used larch for cryptocurrency wallets, the log could enforce a policy such as "deny transactions sending more than $10K to addresses that are not on the allowlist." For policies based on private information, the client could send the log service a commitment to the policy at enrollment, and the log service could then enforce the policy by running a two-party computation or checking a zero-knowledge proof.

Revocation and migration. If a user loses her device or wants to migrate her authentication secrets from an old device to a new device, she needs a way to easily and remotely invalidate the secrets on the old device. Larch allows her to do this. To migrate credentials to a new device, the client and log simply re-share the authentication secrets. To invalidate the secrets on the old device, the client asks the log to delete the old secret shares (client must authenticate with the log first).

Account recovery. In the event that a client loses all of her devices, she needs some way to recover her larch account. To ensure that she can later recover her account, the client can encrypt her larch client state under a key derived from her password and store the ciphertext with the larch service. The security of the backup is only as good as the security of the client's password. Alternatively, the client could choose a random key to encrypt her client state and then back up this key using her password and secure hardware in order to defend against password-guessing attacks [137].

Limitations. If an attacker compromises the client's account with the log, the attacker can access the client's entire authentication history. To mitigate this damage, the log could delete old authentication records (e.g., records older than one week) or re-encrypt them under a key that the user keeps offline.

6.10 Related work

We now describe related work since the time of publication of the original paper [141] (Section 6.10.1) and subsequent (or concurrent) related work (Section 6.10.2).

6.10.1 Related work at the time of publication

Privacy-preserving single sign-on. Prior work has explored how to protect user privacy in single sign-on systems. BrowserID [176] (previously implemented in Mozilla Persona and Firefox Accounts), SPRESSO [177], EL PASSO [535], UnlimitID [265], UPPRESSO [235], PseudoID [156], and Hammann et al. [243] all aim to hide user login patterns from the single sign-on server. Several of these systems [156, 243, 265, 535] are compatible with existing single sign-on protocols for incremental deployment. However, with the exception of UnlimitID [265], these systems do not protect user accounts from a malicious attacker that compromises the single sign-on server. None of these systems privately log the identity of the relying party.

Separately, Privacy Pass allows a user to obtain anonymous tokens for completing CAPTCHAs, which she can then spend at different relying parties without allowing them to link her across sites [143]. Like larch, Privacy Pass does not link users across accounts, but unlike larch, Privacy Pass does not provide a mechanism for logging authentications.

Threshold signing. Our two-party ECDSA with preprocessing protocol builds on prior work on threshold ECDSA. MacKenzie and Reiter proposed the first threshold ECDSA protocol for a dishonest majority specific to the two-party setting [340]. Genarro et al. [198] and Lindell [325] subsequently improved on this protocol. Doerner et al. show how to achieve two-party threshold ECDSA without additional assumptions [160]. Another line of work supports threshold ECDSA using generic multi-party computation over finite fields [131, 457]. A number of works show how to split ECDSA signature generation into online and offline phases [18, 90, 92, 132, 191, 219, 220, 521]; in many, the offline phase is signing-key-specific, allowing for a non-interactive online signing phase, whereas we need an offline phase that is signing-key-independent. Abram et al. show how to reduce the bandwidth of the offline phase via pseudorandom correlation generators [18]. Aumasson et al. provide a survey of prior work on threshold ECDSA [37]. Arora et al. show how to split trust across a group of FIDO authenticators to enable account recovery using a new group signature scheme [33].

Proving properties of encrypted data. Larch's split-secret authentication protocol for FIDO2 and passwords relies on proving properties of encrypted data, which is also explored in prior work. Verifiable encryption was first proposed by Stadler [466], and Camenisch and Damgard introduced it as a well-defined primitive [87]. Subsequent work has designed verifiable encryption schemes for limited classes of relations (e.g. discrete logarithms) [35,88,338,380,523]. Takahashi and Zaverucha introduced a generic compiler for MPC-in-the-head-based verifiable encryption [477]. Lee et al. [314] contribute a SNARK-based verifiable encryption scheme that decouples the encryption function from the circuit by using a commit-and-prove SNARK [89]. This approach does not work for us for FIDO2 authentication because the ciphertext must be connected to a SHA-256 digest.

Grubbs et al. introduce zero-knowledge middleboxes, which enforce properties on encrypted data using SNARKs [221]. Wang et al. show how to build blind certificate authorities, enabling a certificate authority to validate an identity and generate a certificate for it without learning the identity [509]. DECO allows users to prove that a piece of data accessed via TLS came from a particular website and, optionally, prove statements about the data in zero-knowledge [532].

Transparency logs. Like larch, transparency logs detect attacks rather than prevent them, and they achieve this by maintaining a log recording sensitive actions [9,28,107,137,257,309,355]. However, transparency logs traditionally maintain public, global state. For example, the certificate transparency log records what certificates were issued and by whom in order to track when certificates were issued incorrectly [309]. In contrast, the larch log service maintains encrypted, per-user state about individual users' authentication history.

6.10.2 Subsequent related work

Hanzlik, Loss, and Wagner formalize the notion of privacy for WebAuthn (part of FIDO2) and propose a solution for FIDO2 revocation that is inspired by BIP32 from cryptocurrency wallets [245]. FIDO-AC shows a mechanism for integrating FIDO2 with anonymous credentials in order to allow relying parties to verify client attributes as needed [527]. Zombie reduces the costs associated with zero-knowledge middleboxes, which requires techniques for reducing the cost of proving that encrypted data complies with middlebox policies [531].

6.11 ECDSA with additive key derivation and presignatures

We now define security for a variant of ECDSA where (1) the adversary can choose "tweaks" to the signing key, and (2) the adversary can request presignatures that are later used to generate presignatures. In the rest of this section, we will show that the advantage of the adversary in this modified version of ECDSA is negligible. In Section 6.12, we will argue that a two-party protocol that achieves the ideal functionality of this modified version of ECDSA is secure.

Throughout all algorithms implicitly run in time polynomial in the security parameter.

We define a security experiment for ECDSA with preprocessing and additive key derivation in Experiment 1 in Figure 7. The security experiment models the fact that the client's and log's secret key shares are not authenticated, and so the adversary can query for signatures under a signing key

with adversarially chosen "tweaks". However, the final signature must verify under a fixed set of tweaks (in our setting, these correspond to public keys that the client generated at registration). The presignature queries allow us to capture the client preprocessing that we take advantage of.

Theorem 27. *Let* $\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N]$ *denote the adversary* $\mathcal{A}$*'s advantage in Experiment 1 with group* $\mathbb{G}$ *of prime order* q*,* ℓ Gen *queries, and* N *total* PreSign *and* Sign *queries. Then*

$$\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N] \leq O(N\cdot((\ell+N)/q+1/\sqrt{q}))\ .$$

Proof. In order to prove the above theorem, we use an additional experiment, Experiment 2 in Figure 8. Let $\mathsf{GSECDSAAdv}[\mathcal{B},\mathbb{G},N,\mathcal{E}]$ denote the advantage of adversary $\mathcal{B}$ in Experiment 2 with a set of tweaks $\mathcal{E}$ of size ℓ. By Lemma 28,

$$\mathsf{GSECDSAAdv}[\mathcal{A},\mathbb{G},N,\mathcal{E}] \leq O(N\cdot((|\mathcal{E}|+N)/q+1/\sqrt{q}))\ ,$$

and by Lemma 29,

$$\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N] \leq \mathsf{GSECDSAAdv}[\mathcal{B},\mathbb{G},N,\mathcal{E}]\ ,$$

and because $|\mathcal{E}|=\ell$,

$$\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N] \leq O(N\cdot((\ell+N)/q+1/\sqrt{q}))\ .$$

□

For the intermediate security experiment, we use the security game from Groth and Shoup, which we include for reference as Experiment 2 in Figure 8. The intermediate security experiment allows us to leverage Groth and Shoup's security analysis for ECDSA with additive key derivation and presignatures [220]. They define a security game that is similar to but slightly different from Experiment 1 that we define to match our setting. Experiment 2 models the variant of additive key derivation where the signing tweak is not constrained to lie in the set of tweaks that the adversary must produce a forgery for (only the forging tweak is constrained; see Note 1 in Section 6 of Groth and Shoup [220]).

At a high level, the differences between Experiment 1 and Experiment 2 are:

- In Experiment 1, the adversary makes Gen queries to receive public keys corresponding to the set of tweaks, whereas in Experiment 2, the adversary simply receives the set of tweaks directly at the beginning of the experiment (these are an experiment parameter).
- Signing queries in Experiment 1 take as input a share of the tweak rather than the entire signing tweak.
- The Experiment 1 challenger enforces an order on Gen, PreSign, and Sign queries. The Experiment 2 challenger does not enforce an order.

Lemma 28. *Let* $\mathsf{GSECDSAAdv}[\mathcal{A},\mathbb{G},N,\mathcal{E}]$ *denote adversary* $\mathcal{A}$*'s advantage in Experiment 2 with group* $\mathbb{G}$ *of prime order* q*, number of presignature and signing queries* $N\in\mathbb{N}$*, and a set of tweaks* $\mathcal{E}$

of size polynomial in the security parameter. Then if Hash *is collision resistant and* $\mathbb{G}$ *is a random oracle,*

$$\mathsf{GSECDSAAdv}[\mathcal{A},\mathbb{G},N,\mathcal{E}] \leq O(N \cdot ((|\mathcal{E}|+N)/q + 1/\sqrt{q}))\ .$$

We refer the reader to Groth and Shoup's Theorem 4 for the analysis proving Lemma 28 in the generic group model. In our setting, the number of public keys and therefore size of $\mathcal{E}$ is polynomial.

All that remains is to prove that the the adversary in the modified Experiment 1 does not have a greater advantage than the adversary in the original Experiment 2.

Lemma 29. *Let* $\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N]$ *denote the adversary* $\mathcal{A}$*'s advantage in Experiment 1 with group* $\mathbb{G}$ *and* $\ell, N \in \mathbb{N}$*. Let* $\mathsf{GSECDSAAdv}[\mathcal{B},\mathbb{G},N,\mathcal{E}]$ *denote the adversary* $\mathcal{B}$*'s advantage in Experiment 2 with a set of tweaks* $\mathcal{E}$ *of size* ℓ*. Then given an adversary* $\mathcal{A}$ *in Experiment 1, we construct an adversary* $\mathcal{B}$ *for Experiment 2 that runs in time linear in* $\mathcal{A}$ *such that for all groups* $\mathbb{G}$*,* $N \in \mathbb{N}$*, and randomly sampled set of tweaks* $\mathcal{E}$ *of size* ℓ*,*

$$\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N] \leq \mathsf{GSECDSAAdv}[\mathcal{B},\mathbb{G},N,\mathcal{E}]\ .$$

Proof. We prove the above theorem by using an adversary $\mathcal{A}$ in Experiment 1 to construct an adversary $\mathcal{B}$ in Experiment 2. We then show that the adversary $\mathcal{B}$ has an advantage greater than or equal to the adversary $\mathcal{A}$.

We construct $\mathcal{B}$ in the following way:

- Rather than sending $\mathcal{A}$ the set of tweaks $\mathcal{E}$ immediately, $\mathcal{B}$ keeps the set of tweaks to use to respond to Gen queries. On the ith invocation of Gen, $\mathcal{B}$ sends $D \cdot g^{\omega_i}$ where $\omega_i \in \mathcal{E}$.
- $\mathcal{B}$ simply forwards presignature requests from $\mathcal{A}$ to the Experiment 2 challenger and sends the responses back to $\mathcal{A}$.
- $\mathcal{B}$ takes signing requests from $\mathcal{A}$ with an index j and a tweak ω. $\mathcal{B}$ then computes $\omega_j + \omega = \omega'$ where ω_j was the value returned by the jth call to Gen (if $\mathcal{A}$ has not made j calls to Gen, $\mathcal{B}$ outputs $\perp$). $\mathcal{B}$ then forwards the signing request with ω' to the Experiment 2 challenger.
- $\mathcal{B}$ additionally enforces that all presignature queries must be made before signing queries and that presignatures must be used in order. If $\mathcal{A}$ sends queries that do not meet these requirements, $\mathcal{B}$ outputs $\perp$.

The adversary $\mathcal{A}$ cannot distinguish between interactions with $\mathcal{B}$ and the Experiment 1 challenger, and so the advantage of $\mathcal{A}$ is less than or equal to that of $\mathcal{B}$, completing the proof.

□

Zero-knowledge proof of preimage In larch, the log takes as input a hash of the message rather than the message itself. It is important for security that the log has a zero-knowledge proof of the preimage of the signing digest, as ECDSA with presignatures is completely insecure if the signing oracle signs arbitrary digests directly instead of messages [220]. Because the log checks a

Experiment 1: ECDSA with presignatures and additive key derivation. The experiment is parameterized by a number of Gen queries $\ell \in \mathbb{N}$, a number of PreSign and Sign queries $N \in \mathbb{N}$, a group $\mathbb{G}$ of prime order q with generator g, message space $\mathcal{M}$, a hash function $\mathsf{Hash} : \mathcal{M} \to \mathbb{Z}_q$, a conversion function $f : \mathbb{G} \to \mathbb{Z}_q$.

- The challenger initializes state $\mathsf{initdone} = 0$, $\mathsf{presigdone} = 0$.
- The adversary can make ℓ Gen queries and N PreSign and Sign queries.
- $\mathsf{Gen}() \to \mathsf{pk}$:
  - If $\mathsf{initdone} = 0$, $k = 1$; otherwise $k \leftarrow k + 1$.
  - Sample $\mathsf{sk}_k \xleftarrow{R} \mathbb{Z}_q$.
  - Set $\mathsf{ctr}_{\mathsf{presig}}, \mathsf{ctr}_{\mathsf{auth}} \leftarrow 0$.
  - Set $\mathsf{initdone} \leftarrow 1$, $\mathsf{presigdone} \leftarrow 0$.
  - Output g^{sk_k}.
- $\mathsf{PreSign}() \to R$:
  - If $\mathsf{initdone} = 0$ or $\mathsf{presigdone} = 1$, output $\perp$.
  - Sample $r_{\mathsf{ctr}_{\mathsf{presig}}} \xleftarrow{R} \mathbb{Z}_q^*$.
  - Output $g^{r_{\mathsf{ctr}_{\mathsf{presig}}}}$ and set $\mathsf{ctr}_{\mathsf{presig}} \leftarrow \mathsf{ctr}_{\mathsf{presig}} + 1$.
- $\mathsf{Sign}(m, \omega, j) \to \sigma$:
  - If $\mathsf{initdone} = 0$, $\mathsf{ctr}_{\mathsf{presig}} < \mathsf{ctr}_{\mathsf{auth}}$, or $j > k$ or $j < 1$, output $\perp$.
  - Let $R \leftarrow g^{r_{\mathsf{ctr}_{\mathsf{auth}}}}$
  - Let $s \leftarrow r_{\mathsf{ctr}_{\mathsf{auth}}}^{-1} \cdot (\mathsf{Hash}(m) + (\mathsf{sk}_j + \omega) \cdot f(R))$.
  - Set $\mathsf{presigdone} \leftarrow 1$, $\mathsf{ctr}_{\mathsf{auth}} \leftarrow \mathsf{ctr}_{\mathsf{auth}} + 1$.
  - Output $(s, f(R))$.

The output of the experiment is "1" if:

- the signature σ^* on m^* verifies under pk,
- m^* was not an input to Sign, and
- pk was an output of Gen.

The output is"0" otherwise.

Figure 7: Our experiment for security of ECDSA with additive key derivation and presignatures.

Experiment 2: Groth-Shoup ECDSA with presignatures and additive key derivation [220]. We recall the security experiment from Groth and Shoup [220] in Figure 4 for ECDSA with presignatures with the modifications described for additive key derivation.

The experiment is parameterized by the number of total queries the adversary can make $N \in \mathbb{N}$, a group $\mathbb{G}$ of prime order q with generator g, message space $\mathcal{M}$, a hash function $\mathsf{Hash} : \mathcal{M} \to \mathbb{Z}_q$, a conversion function $f : \mathbb{G} \to \mathbb{Z}_q$, and a set of tweaks $\mathcal{E} \subseteq \mathbb{Z}_q$.

- The challenger initializes state:
  - $k \leftarrow 0$, $K \leftarrow \emptyset$
  - $d \xleftarrow{R} \mathbb{Z}_q$, $D \leftarrow g^d \in \mathbb{G}$.
- The adversary can make N total queries (presign or sign).
- Presignature query:
  - $k \leftarrow k+1$, $r_k \xleftarrow{R} \mathbb{Z}_q^*$
  - $R_k \leftarrow g^{r_k} \in \mathbb{G}$
  - $t_k \leftarrow f(R_k) \in \mathbb{Z}_q$. If $t_k = 0$, output $\perp$
  - Return R_k
- Signing request for message m with presignature $k \in K$ and tweak $\omega \in \mathbb{Z}_q$:
  - $K \leftarrow K \setminus \{k\}$
  - $h_k \leftarrow \mathsf{Hash}(m) \in \mathbb{Z}_q$
  - If $h_k + t_k d + t_k \omega = 0$, output $\perp$
  - $s_k \leftarrow r_k^{-1}(h_k + t_k d + t_k \omega) \in \mathbb{Z}_q$
  - Return (s_k, t_k, R_k)
- After making N queries, the adversary must output $(m^*, \sigma^*, \omega^*)$.

The output of the experiment is "1" if:

- the signature σ^* on m^* verifies under $D \cdot g^{\omega^*}$,
- m^* was not an input to a previous signing query, and
- $\omega^* \in \mathcal{E}$.

The output is "0" otherwise.

Figure 8: Security experiment for ECDSA with additive key derivation and presignatures from Groth and Shoup [220].

zero-knowledge proof certifying that the digest preimage is correctly encrypted before signing, it will not sign arbitrary purported hashes generated by a malicious client (the party submitting the hash must know the preimage for the proof to verify).

6.12 Two-party ECDSA with preprocessing

In this section, we describe the construction of our two-party ECDSA with preprocessing protocol and argue its security. In Section 6.12.1, we describe the syntax and construction of our protocol. In Section 6.12.2, we explain a sub-protocol and argue that it is secure. Finally in Section 6.12.3, we prove that our overall protocol is secure by showing that the protocol achieves the ideal functionality captured by the challenger in Experiment 1 from Section 6.11.

6.12.1 Syntax and construction

For our purposes, a two-party ECDSA signature scheme consists of the following algorithms:

- $\mathsf{LogKeyGen}() \to (\mathsf{sk}_0, \mathsf{pk}_0)$: Generate a log secret key $\mathsf{sk}_0 \in \mathbb{Z}_q$ and corresponding public key $\mathsf{pk}_0 \in \mathbb{G}$. The log runs this routine at enrollment.
- $\mathsf{PreSign}() \to (\mathsf{presig}_0, \mathsf{presig}_1)$: Generate presignature $(\mathsf{presig}_0, \mathsf{presig}_1)$ where each presignature should be used to sign exactly once. The client runs this routine many times at enrollment to generate a batch of presignatures.
- $\mathsf{ClientKeyGen}(\mathsf{pk}_0) \to (\mathsf{sk}_1, \mathsf{pk})$: Given the log public key $\mathsf{pk}_0 \in \mathbb{G}$, output a secret key share $\mathsf{sk}_1 \in \mathbb{Z}_q$ and corresponding public key $\mathsf{pk} \in \mathbb{G}$. The client runs this routine during registration with each relying party.

We additionally define the following signing protocol:

- Π_{Sign}: Both parties take as input the message $m \in \mathcal{M}$, the log takes as input log secret key sk_0 and presignature presig_0, and the client takes as input a secret-key share sk_1 and presignature presig_1. The joint output is a signature σ on message m or $\perp$ (if either misbehaved).

The signing protocol outputs ECDSA signatures that verify under pk output by $\mathsf{ClientKeyGen}$, and so the signature-verification algorithm is exactly as in ECDSA.

We include the constructions for the above algorithms and signing protocols in Figure 9. The construction for the Π_{HalfMul} is in Section 6.12.2 and the opening protocol Π_{Open} is the same opening protocol used in SPDZ [133].

6.12.2 Malicious security with half-authenticated secure multiplication

As part of our signing protocol, we use a half-authenticated secure multiplication sub-protocol. We describe the protocol in Figure 10.

We need to ensure that by deviating from the protocol, neither the client nor the log can learn secret information (i.e. the other party's share of the secret key or signing nonce) or produce a signature for a different message. To use tools for malicious security (e.g. information-theoretic MACs [133]) in a black-box way, we need authenticated shares of the signing nonce and the secret key. We can easily generate authenticated shares of the signing nonce as part of the presignature, but generating authenticated shares of the secret key poses several problems: (1) presignatures are generated at enrollment (before the client has secret-key shares), and (2) we don't want which presignature the client uses to leak which relying party the client is authenticating to.

The ECDSA signature scheme for message space $\mathcal{M}$ uses a group $\mathbb{G}$ of prime order q and is parameterized by a hash function $\mathsf{Hash} : \mathcal{M} \to \mathbb{Z}_q$ and a conversion function $f : \mathbb{G} \to \mathbb{Z}_q$. An ECDSA keypair is a pair $(y, g^y) \in \mathbb{Z}_q \times \mathbb{G}$ for $y \xleftarrow{R} \mathbb{Z}_q$. We use a secure multiplication protocol Π_{HalfMul} (Figure 10 in Section 6.12.2) and a secure opening protocol Π_{Open} that returns the result or $\perp$ ("output" step in Figure 1 of SPDZ [133]).

$\underline{\mathsf{LogKeyGen}() \to (\mathsf{sk}_0, \mathsf{pk}_0)}$:

- Sample $x \xleftarrow{R} \mathbb{Z}_q$ and output (x, g^x).

$\underline{\mathsf{PreSign}() \to (\mathsf{presig}_0, \mathsf{presig}_1)}$:

- Sample $r \xleftarrow{R} \mathbb{Z}_q^*$ and compute $R \leftarrow f(g^r)$.
- Sample $\alpha \xleftarrow{R} \mathbb{Z}_q$ and compute $\hat{r} \leftarrow \alpha \cdot r^{-1}$.
- Split r^{-1} into secret shares r_0, r_1; $\hat{r}$ into $\hat{r_0}, \hat{r_1}$; α into α_0, α_1;
- Output $(R, r_0, \hat{r_0}, \alpha_0)$, $(R, r_1, \hat{r_1}, \alpha_1)$.

$\underline{\mathsf{ClientKeyGen}(\mathsf{pk}_0) \to (\mathsf{sk}_1, \mathsf{pk})}$:

- Sample $y \xleftarrow{R} \mathbb{Z}_q$.
- Output $(y, \mathsf{pk}_0 \cdot g^y)$.

$\underline{\Pi_{\mathsf{Sign}}}$:

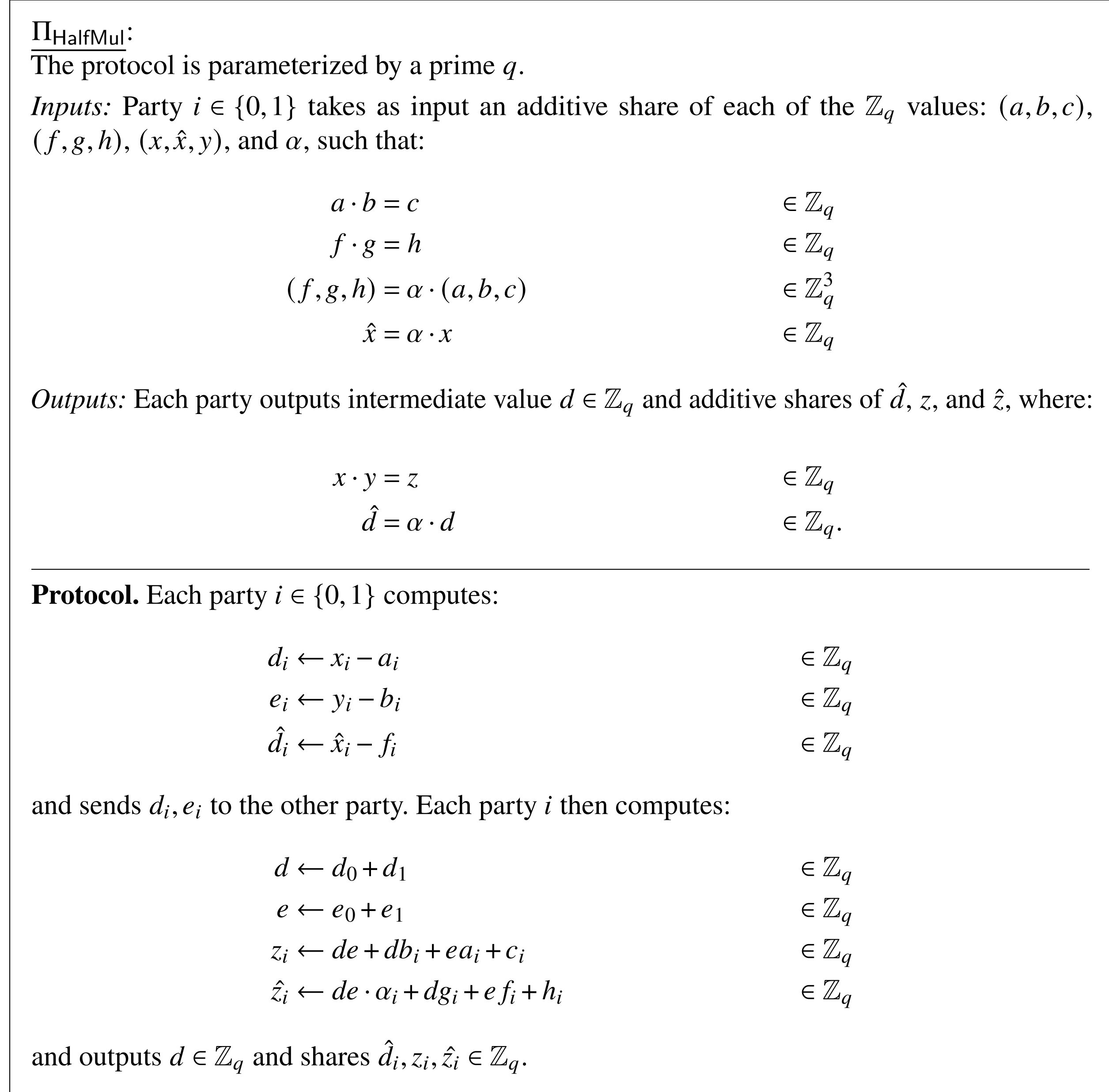
We refer to the log as party 0 and the client as party 1. The input of party $i \in \{0, 1\}$ is $(m, \mathsf{sk}_i, \mathsf{presig}_i)$, and the output is a signature σ on m or $\perp$.

For each party $i \in \{0, 1\}$:

- Party i parses presig_i as $(R, r_i, \hat{r_i}, \alpha_i)$.
- Given input $(r_i, \hat{r_i}, \mathsf{sk}_i, \alpha_i)$ for party i, run Π_{HalfMul} to compute shares $(x_i, \hat{x_i}, v_i, \hat{v_i})$ where $\hat{v_i}$ authenticates any intermediate values.
- Party i computes $s_i \leftarrow r_i \cdot \mathsf{Hash}(m) + x_i \cdot R$.
- Party i computes $\hat{s_i} \leftarrow \hat{r_i} \cdot \mathsf{Hash}(m) + \hat{x_i} \cdot R$.
- Parties run Π_{Open} with party i input $\alpha_i, s_i, \hat{s_i}, v_i, \hat{v_i}$ to get s or $\perp$ (if returns $\perp$, output $\perp$).
- Output (R, s).

Figure 9: Two-party ECDSA signing protocol with preprocessing.

Ideal functionality. At a very high level, the ideal functionality takes as input additive shares of x, y and outputs shares of $x \cdot y$. In order to perform the multiplication, we use Beaver triples. To authenticate x, we use information-theoretic MAC tags (because there is no MAC tag for y, each party can adjust its share by an arbitrary additive shift without detection). More precisely then, the ideal functionality takes as inputs additive shares of (a, b, c), (f, g, h), $(x, \hat{x}, y)$, and α such that $a \cdot b = c$, $f \cdot g = h$, $(f, g, h) = \alpha \cdot (a, b, c)$, and $\hat{x} = \alpha \cdot x$. Each party outputs intermediate value d and additive shares of $\hat{d}, z, \hat{z}$ where $x \cdot y = z$ and $\hat{d} = \alpha \cdot d$.



Π_{HalfMul}:

The protocol is parameterized by a prime q.

Inputs: Party $i \in \{0,1\}$ takes as input an additive share of each of the $\mathbb{Z}_q$ values: (a,b,c), (f,g,h), $(x,\hat{x},y)$, and α, such that:

$$\begin{aligned} a \cdot b &= c && \in \mathbb{Z}_q \\ f \cdot g &= h && \in \mathbb{Z}_q \\ (f,g,h) &= \alpha \cdot (a,b,c) && \in \mathbb{Z}_q^3 \\ \hat{x} &= \alpha \cdot x && \in \mathbb{Z}_q \end{aligned}$$

Outputs: Each party outputs intermediate value $d \in \mathbb{Z}_q$ and additive shares of $\hat{d}$, z, and $\hat{z}$, where:

$$\begin{aligned} x \cdot y &= z && \in \mathbb{Z}_q \\ \hat{d} &= \alpha \cdot d && \in \mathbb{Z}_q. \end{aligned}$$

Protocol. Each party $i \in \{0,1\}$ computes:

$$\begin{aligned} d_i &\leftarrow x_i - a_i && \in \mathbb{Z}_q \\ e_i &\leftarrow y_i - b_i && \in \mathbb{Z}_q \\ \hat{d}_i &\leftarrow \hat{x}_i - f_i && \in \mathbb{Z}_q \end{aligned}$$

and sends d_i, e_i to the other party. Each party i then computes:

$$\begin{aligned} d &\leftarrow d_0 + d_1 && \in \mathbb{Z}_q \\ e &\leftarrow e_0 + e_1 && \in \mathbb{Z}_q \\ z_i &\leftarrow de + db_i + ea_i + c_i && \in \mathbb{Z}_q \\ \hat{z}_i &\leftarrow de \cdot \alpha_i + dg_i + ef_i + h_i && \in \mathbb{Z}_q \end{aligned}$$

and outputs $d \in \mathbb{Z}_q$ and shares $\hat{d}_i, z_i, \hat{z}_i \in \mathbb{Z}_q$.

Figure 10: Π_{HalfMul} protocol.

Protocol. Our protocol uses information-theoretic MACs for only one of the inputs (the signing nonce). We call this protocol Π_{HalfMul}. Our construction uses authenticated Beaver triples and follows naturally from the SPDZ protocol [133]. We also use a secure opening protocol Π_{Open} for checking MAC tags, which we can instantiate using the SPDZ protocol directly [133]. It is safe to not authenticate one of the key shares due to the fact that the signature scheme is secure if the adversary can request signatures for arbitrary "tweaks" of the secret key (Section 6.11).

We present a slightly modified version of the SPDZ protocol [133] for multiplication on authenticated secret-shared inputs in Figure 10. The only difference is that, in our protocol, only one of the inputs is authenticated. This requirement means that we only authenticate one of the intermediate values in the Beaver triple multiplication. We allow the attacker to add arbitrary shifts to the unauthenticated input, but the attacker cannot shift the authenticated input without detection (Claim 30).

The protocol Π_{HalfMul} allows us to model authenticating the signing nonce in ECDSA signing. The signing key is unrestricted (we discuss why this is secure in Section 6.11).

We first show the security of our Π_{HalfMul} protocol for secure multiplication where only one of the inputs is authenticated, which is very similar to the multiplication protocol in SPDZ [133]. This allows us to ensure that both parties use the correct signing nonce from the presignature.

Claim 30. *Let $x, y, \alpha, \hat{x} \in \mathbb{Z}_q$ be inputs to Π_{HalfMul} secret-shared across the parties where $\hat{x} = \alpha x$. Then, the probability that an adversary that has statically corrupted one of the parties can cause the protocol Π_{HalfMul} to output shares of $z, \hat{z}, \hat{d}$ where $\hat{z} = \alpha \cdot z$ and $z = (x + \Delta)y$ for some $\Delta \neq 0$ is $1/q$.*

Proof. Let the input Beaver triple be $(a, b, c) \in \mathbb{Z}_q$ such that $a \cdot b = c$ where to multiply values x, y, we use intermediate values $d = x - a$ and $e = y - b$, where $\hat{d}$ is the MAC tag for d.

To avoid detection, the adversary needs to ensure that $\hat{d} = \alpha \cdot d$, so the adversary needs to find some $\Delta_1, \Delta_2 \in \mathbb{Z}_q$ such that

$$\alpha(x + \Delta_1 - a) = \hat{x} + \Delta_2 - \alpha \cdot a \quad \in \mathbb{Z}_q$$

which we can reduce to

$$\alpha(x + \Delta_1) = \hat{x} + \Delta_2 \quad \in \mathbb{Z}_q$$

The probability of the adversary choosing $\Delta_1, \Delta_2 \in \mathbb{Z}_q$ that satisfies this equation is the probability of guessing α, or $1/q$. The value e does not depend on x. Therefore, since the remainder of the operations are additions and multiplications by public values, the attacker can only shift the final output by Δ_3, Δ_4 and needs to ensure the following:

$$\alpha(xy + \Delta_3) = \alpha xy + \Delta_4 \quad \in \mathbb{Z}_q$$

The probability of finding such Δ_3, Δ_4 is the probability of guessing α, which is $1/q$. □

6.12.3 Security proof for our construction

Recall our construction of our two-party signing scheme in Figure 9.

As the construction in Figure 9 contains separate algorithms for key generation and generating presignatures, we define Π_{Gen} and Π_{PreSign} in terms of the algorithms in Figure 9 below:

- Π_{Gen}:
  - If pk_0 is not initialized, the log runs $(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{LogKeyGen}()$ and sends pk_0 to the client.
  - If k is not initialized, set to 1; otherwise $k \leftarrow k + 1$.

  - The client runs $(\mathsf{sk}_{1,k}, \mathsf{pk}_k) \leftarrow \mathsf{ClientKeyGen}(\mathsf{pk}_0)$

- Π_{PreSign}:
  - Client runs $(\mathsf{presig}_0, \mathsf{presig}_1) \leftarrow \mathsf{PreSign}()$ and sends presig_0 to the log.

Additive key derivation models the fact that the secret key shares are unauthenticated private inputs to Π_{HalfMul}, and so the adversary can run the signing protocol with any secret key share as input. However, to produce a forgery, the adversary must generate a signature that verifies under a small, fixed set of public keys (corresponding to the public keys generated at registration before compromise).

We define the ideal functionality $\mathcal{F}_{\mathsf{ECDSA}}$ as simply the routines the challenger runs to respond to Gen,PreSign, and Sign queries in Experiment 1.

We prove security using the ideal functionality $\mathcal{F}_{\mathsf{Open}}$ for opening values and checking MAC tags from SPDZ [133]. At a high level, $\mathcal{F}_{\mathsf{Open}}$ takes as input the party's output and intermediate shares and MAC tags for output and intermediate shares and outputs the combined output or abort if the MAC tags are not correct. The corresponding simulator $\mathsf{Sim}_{\mathsf{Open}}$ takes as input one party's shares of the output and intermediate values and their corresponding MAC tags, as well as the combined output value.

Theorem 31. *The two-party ECDSA signing protocol* Π *securely realizes (with abort)* $\mathcal{F}_{\mathsf{ECDSA}}$ *in the* $\mathcal{F}_{\mathsf{Open}}$*-hybrid model in the presence of a single statically corrupted malicious party (if the client is the compromised party, it can only be compromised after presigning is complete). Specifically, let* $\mathsf{View}_{\Pi}^{\textbf{Real}}$ *denote the adversary* $\mathcal{A}$*'s view in the real world. Then there exists a probabilistic polynomial-time algorithm* Sim *where* $\mathsf{View}_{\mathsf{Sim}}^{\textbf{Ideal}}$ *denotes the view simulated by* Sim *given the outputs of* $\mathcal{F}_{\mathsf{ECDSA}}$ *where* $\mathcal{A}$ *can adaptively choose which procedures to run and the corresponding inputs such that*

$$\mathsf{View}_{\Pi}^{\textbf{Real}} \approx \mathsf{View}_{\mathsf{Sim}}^{\textbf{Ideal}} .$$

Proof. Our goal is to construct a simulator where the simulator takes as input the outputs of the ideal functionality $\mathcal{F}$ (as well at the public input message for signing). The adversary $\mathcal{A}$ should then not be able to distinguish between the real world (interaction with the protocol) and the ideal world (interaction with the simulator where the simulator is given the compromised party's inputs and the outputs of $\mathcal{F}_{\mathsf{ECDSA}}$).

Let i be the index of the compromised party ($i = 0$ for compromised client, $i = 1$ for compromised log). The simulator always generates the presignatures and outputs the presignature share to the adversary, in order to model the fact that we only provide security if the client is malicious at signing time. We construct the simulator as follows:

- $\mathsf{Gen}(\mathsf{pk})$:
  - If $i = 0$:
    - ∗ If sk_0 is not initialized, sample $\mathsf{sk}_0 \xleftarrow{R} \mathbb{Z}_q$ and send $\mathsf{pk}_0 \leftarrow g^{\mathsf{sk}_0}$ to $\mathcal{A}$.
    - ∗ Otherwise, send nothing to $\mathcal{A}$.
  - Otherwise if $i = 1$:

    * If pk_0 is not initialized, receive pk_0 from $\mathcal{A}$.
    * Output pk.
    * Set $\mathsf{initdone} \leftarrow 1$, $\mathsf{presigdone} \leftarrow 0$.
  - Set $\mathsf{ctr}_{\mathsf{presig}}, \mathsf{ctr}_{\mathsf{auth}} \leftarrow 0$.
  - Set $\mathsf{initdone} \leftarrow 1$.

- $\mathsf{PreSign}(R)$:
  - If $\mathsf{initdone} = 0$ or $\mathsf{presigdone} = 1$, output $\bot$.
  - Sample $\alpha_0, \alpha_1 \xleftarrow{R} \mathbb{Z}_q$, $r_0, r_1 \xleftarrow{R} \mathbb{Z}_q^*$.
  - Set $\alpha^{(\mathsf{ctr}_{\mathsf{presig}})} \leftarrow \alpha_0 + \alpha_1, r^{(\mathsf{ctr}_{\mathsf{presig}})} \leftarrow r_0 + r_1$.
  - Sample $\hat{r_0}, \hat{r_1}$ such that $\alpha^{(\mathsf{ctr}_{\mathsf{presig}})} \cdot r^{(\mathsf{ctr}_{\mathsf{presig}})} = \hat{r_0} + \hat{r_1}$.
  - Sample shares of $(a,b,c),(f,g,h) \in \mathbb{Z}_q^3$ such that $a \cdot b = c$, $f \cdot g = h$, $(f,g,h) = \alpha^{(\mathsf{ctr}_{\mathsf{presig}})}(a,b,c)$, $\hat{x} = \alpha \cdot x$.
  - Let $T_j^{\mathsf{ctr}_{\mathsf{presig}}} = (a_j, b_j, c_j, f_j, g_j, h_j)$ for $j \in \{1,2\}$.
  - Let $\mathsf{presig}_0^{(\mathsf{ctr}_{\mathsf{presig}})} = (R, r_0, \hat{r_0}, \alpha_0, T_0^{(\mathsf{ctr}_{\mathsf{presig}})})$ and $\mathsf{presig}_1^{(\mathsf{ctr}_{\mathsf{presig}})} = (R, r_1, \hat{r_1}, \alpha_1, T_1^{(\mathsf{ctr}_{\mathsf{presig}})})$.
  - Send $\mathsf{presig}_{1-i}^{(\mathsf{ctr}_{\mathsf{presig}})}$ to $\mathcal{A}$.
  - Set $\mathsf{ctr}_{\mathsf{presig}} \leftarrow \mathsf{ctr}_{\mathsf{presig}} + 1$.

- $\mathsf{Sign}(m, \sigma)$:
  - If $\mathsf{initdone} = 0$ or $\mathsf{ctr}_{\mathsf{presig}} < \mathsf{ctr}_{\mathsf{auth}}$, output $\bot$.
  - Parse σ as $(s, _)$.
  - Parse $\mathsf{presig}_i^{(\mathsf{ctr}_{\mathsf{auth}})}$ as $(R, r_i, \hat{r_i}, \alpha_i, T_i)$.
  - Parse $T_i^{(\mathsf{ctr}_{\mathsf{auth}})}$ as $(a_i, b_i, c_i, f_i, g_i, h_i)$.
  - Let $x \leftarrow \frac{s - r^{(\mathsf{ctr}_{\mathsf{auth}})} \cdot \mathsf{Hash}(m)}{R}$
  - Let $\mathsf{sk}_i \leftarrow x / r_i$
  - Let $\hat{d_i} = \hat{r_i} - f_i$.
  - Send $d_i = r_i - a_i$ and $e_i = \mathsf{sk}_i - b_i$ to $\mathcal{A}$.
  - Receive d_{1-i} and e_{1-i} from $\mathcal{A}$ and compute $d = d_0 + d_1$ and $e = e_0 + e_1$.
  - Let $x_i \leftarrow de + db_i + ea_i + c_i$
  - Let $\hat{x_i} \leftarrow de\alpha_i + dg_i + ef_i + h_i$
  - Compute $s' \leftarrow r^{(\mathsf{ctr}_{\mathsf{presig}})} \cdot \mathsf{Hash}(m) + x \cdot R$
  - Compute $s_i \leftarrow r_i \cdot \mathsf{Hash}(m) + x_i \cdot R + s - s'$
  - Compute $\hat{s_i} \leftarrow \hat{r_i} \cdot \mathsf{Hash}(m) + \hat{x_i} \cdot R + \alpha^{(\mathsf{ctr}_{\mathsf{auth}})}(s - s')$
  - Run $\mathsf{Sim}_{\mathsf{Open}}$ on $(\alpha_i, s_i, \hat{s_i}, d, \hat{d_i}, s)$; if $\mathsf{Sim}_{\mathsf{Open}}$ aborts, also abort.
  - Set $\mathsf{presigdone} \leftarrow 1$ and $\mathsf{ctr}_{\mathsf{auth}} \leftarrow \mathsf{ctr}_{\mathsf{auth}} + 1$.

We now prove that the view generated by Sim in the ideal world is indistinguishable from the real world.

We start with the real world (Figure 9). We then replace calls to Π_{Open} with $\mathsf{Sim}_{\mathsf{Open}}$. Because we are in the $\mathcal{F}_{\mathsf{Open}}$-hybrid model, the adversary cannot distinguish between these.

Every other message sent to $\mathcal{A}$ is either (1) a value that is random or information theoretically indistinguishable from random, or (2) a value generated by $\mathcal{F}_{\mathsf{ECDSA}}$ (R in $\mathsf{PreSign}$).

The last step is to show that if $\mathcal{A}$ deviates from the protocol when sending d_{1-i} to the simulator, the simulator can detect this and abort. By Claim 30 and the guarantees of $\mathcal{F}_{\mathsf{Open}}$, the adversary cannot send an incorrect value for d_{1-i} without detection except with probability $1/q$. The adversary *can* send any value for e_{1-i}; this is equivalent to the adversary being allowed to choose any signing tweak ω, which the attacker can do in Experiment 1.

Therefore, $\mathcal{A}$ cannot distinguish between the real world and the ideal world except with probability $1/q$, completing the proof. □

6.13 Protocol details for password-based logging

6.13.1 Zero-knowledge proofs for discrete log relations

The protocol of Section 6.5 requires the client to prove to the log service that the ElGamal decryption of a ciphertext decrypts to one of n values in a set. To do so, the client uses a zero-knowledge proof of discrete-log relations, whose syntax and construction we describe here. The proof system uses a cyclic group $\mathbb{G}$ of prime order q.

The proof system consists of two algorithms:

- $\mathsf{DLProof.Prove}(\mathsf{idx}, x, h, \mathsf{cm}_1, \ldots, \mathsf{cm}_n) \to \pi$:
  Output a proof π asserting that $\mathsf{cm}_{\mathsf{idx}} = h^x \in \mathbb{G}$ for $\mathsf{idx} \in [n]$
- $\mathsf{DLProof.Verify}(\pi, h, \mathsf{cm}_1, \ldots, \mathsf{cm}_n) \to \{0,1\}$:
  Check the prover's claim that it knows some $x \in \mathbb{Z}_q$ where $\mathsf{cm}_{\mathsf{idx}} = h^x \in \mathbb{G}$ for $\mathsf{idx} \in [n]$.

We require the standard notions of *completeness*, *soundness* (against computationally bounded provers), and *zero knowledge* [66, 208] (in the random-oracle model [53]). We instantiate DLProof using proof techniques from Groth and Kohlweiss [218].

6.13.2 Protocol for passwords

We now describe the syntax of our $\mathsf{Larch}_{\mathsf{PW}}$ scheme.

Step #1: Enrollment with log service. At enrollment, the client and log generate cryptographic keys and exchange public keys.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{ClientGen}() \to (x, X)$: The client outputs a secret key $x \in \mathbb{Z}_q$ and a public key $X \in \mathbb{G}$.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{LogGen}() \to (k, K)$: The log service outputs a secret key $k \in \mathbb{Z}_q$ and a public key $K \in \mathbb{G}$.

Step #2: Registration with relying party. Once the client has enrolled with a log service, it can register with a relying party by interacting with the log service.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{ClientRegister}() \to (\mathsf{id}, k_{\mathsf{id}})$: The client outputs an identifier $\mathsf{id} \in \{0,1\}^\lambda$ and a key $k_{\mathsf{id}} \in \mathbb{G}$.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{LogRegister}(k, \mathsf{id}) \to y$: Given the log's secret key k and id produced by ClientRegister, the log outputs $y \in \mathbb{G}$.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{FinishRegister}(k_{\mathsf{id}}, y) \to \mathsf{pw}_{\mathsf{id}}$: Given the key k_{id} generated by ClientRegister and the value y generated by LogRegister, the client outputs the password $\mathsf{pw}_{\mathsf{id}} \in \mathbb{G}$.

Step #3: Authentication with relying party. After registration, the client and log service perform authentication together.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{ClientAuth}(\mathsf{idx}, x, \mathsf{id}_1, \dots, \mathsf{id}_n) \to (r, \mathsf{ct}, \pi_1, \pi_2)$: Given an index $\mathsf{idx} \in \{1, \dots, n\}$, the client's secret key $x \in \mathbb{Z}_q$, and identifier values $\mathsf{id}_1, \dots, \mathsf{id}_n$ output by ClientRegister, the client outputs $r \in \mathbb{Z}_q^*$, an ElGamal ciphertext $\mathsf{ct} \in \mathbb{G}^2$, and proofs π_1 and π_2.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{LogAuth}(\mathsf{ct}, \pi_1, \pi_2, \mathsf{id}_1, \dots, \mathsf{id}_n) \to y$: Given a ciphertext $\mathsf{ct} \in \mathbb{G}^2$, proofs π_1 and π_2, and identifiers $\mathsf{id}_1, \dots, \mathsf{id}_n$ output by ClientRegister, the log service outputs $y \in \mathbb{G}$.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{FinishAuth}(x, K, r, k_{\mathsf{id}}, y) \to \mathsf{pw}_{\mathsf{id}}$: Given the client's secret key $x \in \mathbb{Z}_q$, the log's public key $K \in \mathbb{G}$, the nonce $r \in \mathbb{Z}_q^*$ generated by ClientAuth, the key $k_{\mathsf{id}} \in \mathbb{G}$ generated by ClientRegister, and the value $y \in \mathbb{G}$ from LogAuth, output the password $\mathsf{pw}_{\mathsf{id}} \in \mathbb{G}$.

Step #4: Auditing with log service. Given a ciphertext, the client runs ElGamal decryption to recover the corresponding $\mathsf{Hash}(\mathsf{id})$ value.

We give a detailed description of the larch password-based authentication protocol in Figure 11.

6.14 Conclusion

Larch is an authentication manager that logs every successful authentication to any of a user's accounts on a third-party log service. It guarantees log integrity without trusting clients. It furthermore guarantees account security and privacy without trusting the log service. Larch works with any existing service supporting FIDO2, TOTP, or password-based login. Our evaluation shows the implementation is practical and cost-effective.

Larch password-based authentication scheme. The protocol is parameterized by: a cyclic group $\mathbb{G}$ of prime order q with generator $g \in \mathbb{G}$, a hash function $\mathsf{Hash}\colon \{0,1\}^* \to \mathbb{G}$, and a zero-knowledge discrete-log proof system DLProof with syntax as in Section 6.13.1.

$\underline{\mathsf{Larch_{PW}.ClientGen}() \to (x, X)}$

- Sample $x \xleftarrow{R} \mathbb{Z}_q$.
- Output (x, g^x).

$\underline{\mathsf{Larch_{PW}.LogGen}() \to (k, K)}$:

- Sample $k \xleftarrow{R} \mathbb{Z}_q$.
- Output (k, g^k).

$\underline{\mathsf{Larch_{PW}.ClientRegister}() \to (\mathsf{id}, k_{\mathsf{id}})}$:

- Sample $\mathsf{id} \xleftarrow{R} \{0,1\}^\lambda$.
- Sample $k_{\mathsf{id}} \xleftarrow{R} \mathbb{G}$.
- Output $(\mathsf{id}, k_{\mathsf{id}})$.

$\underline{\mathsf{Larch_{PW}.LogRegister}(k, \mathsf{id}) \to y}$:

- Output $\mathsf{Hash}(\mathsf{id})^k$.

$\underline{\mathsf{Larch_{PW}.FinishRegister}(k_{\mathsf{id}}, y) \to \mathsf{pw_{id}}}$:

- Output $k_{\mathsf{id}} \cdot y$.

$\underline{\mathsf{Larch_{PW}.ClientAuth}(\mathsf{idx}, x, \mathsf{id}_1, \dots, \mathsf{id}_n) \to (r, \mathsf{ct}, \pi_1, \pi_2)}$:

- Sample $r \xleftarrow{R} \mathbb{Z}_q^*$
- Compute $c_1 = g^r, c_2 = \mathsf{Hash}(\mathsf{id_{idx}}) \cdot g^{xr}$.
- Let $h_i = c_2 / \mathsf{Hash}(\mathsf{id}_i)$ for $i \in \{1, \dots, n\}$.
- Let $\pi_1 \leftarrow \mathsf{DLProof.Prove}(\mathsf{idx}, r, X, h_1, \dots, h_n)$.
- Let $\pi_2 \leftarrow \mathsf{DLProof.Prove}(\mathsf{idx}, x, c_1, h_1, \dots, h_n)$.
- Let $\mathsf{ct} = (c_1, c_2)$.
- Output $(r, \mathsf{ct}, \pi_1, \pi_2)$.

$\underline{\mathsf{Larch_{PW}.LogAuth}(\mathsf{ct}, \pi_1, \pi_2, \mathsf{id}_1, \dots, \mathsf{id}_n) \to y}$:

- Parse ct as (c_1, c_2).
- Let $h_i = c_2 / \mathsf{Hash}(\mathsf{id}_i)$ for $i \in \{1, \dots, n\}$.
- Let $b_1 \leftarrow \mathsf{DLProof.Verify}(\pi_1, X, h_1, \dots, h_n)$
- Let $b_2 \leftarrow \mathsf{DLProof.Verify}(\pi_2, c_1, h_1, \dots, h_n)$
- If $b_1 \neq 1$ or $b_2 \neq 1$, output $\perp$
- Output c_2^k

$\underline{\mathsf{Larch_{PW}.FinishAuth}(x, K, r, k_{\mathsf{id}}, y) \to \mathsf{pw_{id}}}$:

- Output $k_{\mathsf{id}} \cdot y \cdot K^{-xr}$.

Figure 11: The details of the larch protocol for password-based authentication.

Part III

Conclusion

In this dissertation, we contributed five systems that provide strong security guarantees even if some of the individual components are compromised. We showed how to hide queries to private outsourced data for keyword search in DORY (Chapter 2), time-series analytics queries in Waldo (Chapter 3), and object-store queries in Snoopy (Chapter 4). We also explored how to secure user accounts by protecting user backups from compromised hardware security modules in SafetyPin (Chapter 5) and hiding authentication logs from a logging service in larch (Chapter 6).

These systems all rely on a co-design of systems techniques and cryptography. While general-purpose cryptographic techniques can achieve the properties we want in theory [205, 525], we need to consider the specific system model and requirements of the application in order to minimize overheads. To reduce friction, systems should respect the constraints of existing software systems, the limitations of hardware, and the expectations of users. This is critical for easing the path to adoption.

Bibliography

[1] Amazon EC2 instance network bandwidth. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html, Accessed August 8, 2021.

[2] Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/, Accessed December 7, 2022.

[3] cryptoTools. https://github.com/ladnir/cryptoTools/tree/master, Accessed August, 2021.

[4] FIDO Device Onboarding Conformance Server. https://github.com/fido-alliance/iot-fdo-conformance-tools, Accessed June 26, 2024.

[5] libPSI. https://github.com/osu-crypto/libPSI.

[6] Multi-Protocol SPDZ: Versatile framework for multi-party computation. https://github.com/data61/MP-SPDZ, Accessed July 30, 2021.

[7] SEAL-ORAM. https://github.com/InitialDLab/SEAL-ORAM.

[8] Snoopy repository. https://github.com/ucbrise/snoopy.

[9] Trillian. https://github.com/google/trillian.

[10] CVE-2015-5464. Available from MITRE, CVE-ID CVE-2015-5464., February 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5464.

[11] Titan in depth: Security in plaintext. Google, August 2017. https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext.

[12] Server requirements and transport binding profile, 2018. https://fidoalliance.org/specs/fido-v2.0-rd-20180702/fido-server-v2.0-rd-20180702.html, Accessed June 26, 2024.

[13] How many accounts can I register my YubiKey with?, 2020. https://support.yubico.com/hc/en-us/articles/360013790319-How-many-accounts-can-I-register-my-YubiKey-with-FIDO2.

[14] Passkeys. Google, 2022. https://developers.google.com/identity/passkeys.

[15] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *SOSP*, 39(5):59–74, 2005.

[16] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Efficient synchronous byzantine consensus. *arXiv preprint arXiv:1704.02397*, 2017.

[17] Ittai Abraham, Christopher W Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing ORAM with PIR. In *PKC*, pages 91–120. Springer, 2017.

[18] Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *Security & Privacy*. IEEE, 2022.

[19] Spencer Ackerman. Lavabit email service abruptly shut down citing government interference, 2013. https://www.theguardian.com/technology/2013/aug/08/lavabit-email-shut-down-edward-snowden.

[20] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. *IACR Cryptology ePrint Archive*, 2021.

[21] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *NDSS*, 2018.

[22] Ishtiyaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Coeus: A system for oblivious document ranking and retrieval. In *SOSP*, pages 672–690, 2021.

[23] Amazon S3 pricing. https://aws.amazon.com/s3/pricing/, Accessed 12 February 2020.

[24] Amazon Timestream. https://aws.amazon.com/timestream/, Accessed August, 2021.

[25] Ghous Amjad, Seny Kamara, and Tarik Moataz. Forward and backward private searchable encryption with SGX. In *Proceedings of the 12th European Workshop on Systems Security*, pages 1–6, 2019.

[26] Anchorage. https://anchorage.com/, Accessed May 25, 2020.

[27] Michael P Andersen and David E Culler. Btrdb: Optimizing storage system design for timeseries processing. In *USENIX FAST*, pages 39–52, 2016.

[28] 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.

[29] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, 2016.

[30] Apache Druid. `https://druid.apache.org/`, Accessed August 2021.

[31] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS*, pages 805–817, 2016.

[32] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[33] Sunpreet S Arora, Saikrishna Badrinarayanan, Srinivasan Raghuraman, Maliheh Shirvanian, Kim Wagner, and Gaven Watson. Avoiding lock outs: Proactive FIDO account recovery using managerless group signatures. *IACR Cryptology ePrint Archive*, 2022.

[34] Muhammad Rizwan Asghar, Giovanni Russello, Bruno Crispo, and Mihaela Ion. Supporting complex queries and access policies for multi-user encrypted databases. In *Workshop on Cloud computing security workshop*, pages 77–88. ACM, 2013.

[35] Giuseppe Ateniese. Verifiable encryption of digital signatures and applications. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):1–20, 2004.

[36] Attestation Service for Intel SGX. `https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf`.

[37] Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ECDSA threshold signing. *IACR Cryptology ePrint Archive*, 2020.

[38] Amazon AWS. Amazon EC2 On-Demand Pricing. `https://aws.amazon.com/ec2/pricing/on-demand/`.

[39] Michael Backes, Christian Cachin, and Alina Oprea. Secure key-updating for lazy revocation. In *ESORICS*, pages 327–346. Springer, 2006.

[40] Michael Backes, Amir Herzberg, Aniket Kate, and Ivan Pryvalov. Anonymous ram. In *ESORICS*, pages 344–362. Springer, 2016.

[41] Kassem Bagher, Shujie Cui, Xingliang Yuan, Carsten Rudolph, and Xun Yi. TimeClave: Oblivious in-enclave time series processing system. In *International Conference on Information and Communications Security*, pages 719–737. Springer, 2023.

[42] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: Securing LSM-based key-value stores using shielded execution. In *FAST*, pages 173–190, 2019.

[43] Sumeet Bajaj and Radu Sion. Ficklebase: Looking into the future to erase the past. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 86–97. IEEE, 2013.

[44] Karim Banawan and Sennur Ulukus. The capacity of private information retrieval from byzantine and colluding databases. *IEEE Transactions on Information Theory*, 65(2):1206–1219, 2018.

[45] Feng Bao, Robert H Deng, Xuhua Ding, and Yanjiang Yang. Private query on encrypted data in multi-user settings. In *Information Security Practice and Experience*, pages 71–85. Springer, 2008.

[46] Kenneth E Batcher. Sorting networks and their applications. In *AFIPS*, 1968.

[47] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. SMCQL: secure querying for federated databases. *VLDB*, 2017.

[48] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[49] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. In *CHES*. Springer, 2013.

[50] Amos Beimel and Yoav Stahl. Robust information-theoretic private information retrieval. In *International Conference on Security in Communication Networks*, pages 326–341. Springer, 2002.

[51] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *ASIACRYPT*, 2001.

[52] Mihir Bellare, Dennis Hofheinz, and Scott Yilek. Possibility and impossibility results for encryption and commitment secure under selective opening. In *EUROCRYPT*, pages 1–35. Springer, 2009.

[53] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73, 1993.

[54] Steven Michael Bellovin and William R Cheswick. Privacy-enhanced searches using encrypted bloom filters. *IACR Cryptology ePrint Archive*, 2007.

[55] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10. ACM, 1988.

[56] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, pages 169–188. Springer, 2011.

[57] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a blockchain keep a secret? *TCC*, 2020.

[58] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[59] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: the heavily loaded case. In *STOC*, pages 745–754, 2000.

[60] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[61] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. HE3DB: An efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption. In *CCS*, pages 2930–2944, 2023.

[62] Alexander Bienstock, Yevgeniy Dodis, and Kevin Yeo. Forward secret encrypted RAM: Lower bounds and applications. *IACR Cryptology ePrint Archive*, 2021, 2021.

[63] 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.

[64] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *USENIX Security*, pages 1213–1227, 2018.

[65] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349, 2012.

[66] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *ACM STOC*. 1988.

[67] Dan Boneh, Manu Drijvers, and Gregory Neven. BLS multi-signatures with public-key aggregation. `https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html`, Accessed May 23, 2020, 03 2018.

[68] Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, pages 213–229, 2001.

[69] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT*, pages 416–432. Springer, 2003.

[70] Dan Boneh and Richard J Lipton. A revocable backup system. In *USENIX Security*, pages 91–96, 1996.

[71] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. 2020.

[72] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):1–51, 2014.

[73] Christoph Bösch, Andreas Peter, Bram Leenders, Hoon Wei Lim, Qiang Tang, Huaxiong Wang, Pieter Hartel, and Willem Jonker. Distributed searchable symmetric encryption. In *PST*, pages 330–337. IEEE, 2014.

[74] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In *MICRO*. IEEE/ACM, 2019.

[75] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *EUROCRYPT (2)*, pages 871–900, 2021.

[76] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *TCC*. IACR, 2016.

[77] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, pages 337–367, 2015.

[78] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS*, pages 1292–1303, 2016.

[79] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *TCC*, pages 341–371, 2019.

[80] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *DSN*. IEEE, 2017.

[81] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX WOOT*, 2017.

[82] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed oram. In *SCN*, pages 215–232, 2020.

[83] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. Timecrypt: Encrypted data stream processing at scale with cryptographic access control. In *NSDI*, pages 835–850. USENIX, 2020.

[84] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. Zeph: Cryptographic enforcement of end-to-end data privacy. In *OSDI*, pages 387–404. USENIX, 2021.

[85] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. Bios chronomancy: Fixing the core root of trust for measurement. In *CCS*, pages 25–36. ACM, 2013.

[86] Chengjun Cai, Yichen Zang, Cong Wang, Xiaohua Jia, and Qian Wang. Vizard: A metadata-hiding data analytic system with end-to-end policy controls. In *CCS*, pages 441–454, 2022.

[87] Jan Camenisch and Ivan Damgård. Verifiable encryption, group encryption, and their applications to separable group signatures and signature sharing schemes. In *ASIACRYPT*, pages 331–345. Springer, 2000.

[88] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO*, pages 126–144. Springer, 2003.

[89] Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *CCS*, pages 2075–2092, 2019.

[90] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *CCS*, pages 1769–1787, 2020.

[91] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT*, pages 255–271, 2003.

[92] Ran Canetti, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA. *IACR Cryptology ePrint Archive*, 2020.

[93] Ran Canetti, Srinivasan Raghuraman, Silas Richelson, and Vinod Vaikuntanathan. Chosen-ciphertext secure fully homomorphic encryption. In *PKC*, pages 213–240. Springer, 2017.

[94] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679. ACM, 2015.

[95] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. In *NDSS*, volume 14, pages 23–26. Citeseer, 2014.

[96] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[97] Anrin Chakraborti and Radu Sion. ConcurORAM: High-throughput stateless parallel multi-client ORAM. In *NDSS*, 2019.

[98] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. Graphos: Towards oblivious graph processing. *IACR Cryptology ePrint Archive*, 2024.

[99] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. Dynamic searchable encryption with optimal search in the presence of deletions. In *USENIX Security*, pages 2425–2442, 2022.

[100] Javad Ghareh Chamani, Yun Wang, Dimitrios Papadopoulos, Mingyang Zhang, and Rasool Jalili. Multi-user dynamic searchable symmetric encryption with corrupted participants. *IEEE Transactions on Dependable and Secure Computing*, 20(1):114–130, 2021.

[101] T-H Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel RAM. In *ASIACRYPT*. IACR, 2017.

[102] 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*. Springer, IACR, 2017.

[103] T-H Hubert Chan and Elaine Shi. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In *TCC*. IACR, 2017.

[104] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ASIACRYPT*, pages 442–455. Springer, 2005.

[105] Melissa Chase, Hannah Davis, Esha Ghosh, and Kim Laine. Acsesor: A new framework for auditable custodial secret storage and recovery. *IACR Cryptology ePrint Archive*, 2022.

[106] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *CCS*, pages 1825–1842, 2017.

[107] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *CCS*, pages 1639–1656, 2019.

[108] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: improved efficiency and generic constructions. In *TCC*. IACR, 2016.

[109] 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*, pages 142–157. IEEE, 2019.

[110] Weikeng Chen and Raluca Ada Popa. Metal: A metadata-hiding file sharing system. In *NDSS*, 2020.

[111] 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.

[112] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, pages 41–50. IEEE, 1995.

[113] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–982, 1998.

[114] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204, 2007.

[115] Google Cloud. Network bandwidth. `https://cloud.google.com/compute/docs/network-bandwidth`.

[116] Aloni Cohen, Justin Holmgren, Ryo Nishimaki, Vinod Vaikuntanathan, and Daniel Wichs. Watermarking cryptographic capabilities. *SIAM*, 47(6):2157–2202, 2018.

[117] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J Wu, and Bryan Ford. Authenticated private information retrieval. In *USENIX Security*, pages 3835–3851, 2023.

[118] Graeme Connell. Technology deep dive: Building a faster ORAM layer for enclaves, 2022. `https://signal.org/blog/building-faster-oram/`, Accessed June 27, 2024.

[119] 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 (to appear)*, 2024.

[120] 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.

[121] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, pages 259–282, 2017.

[122] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *Security & Privacy*, pages 321–338. IEEE, 2015.

[123] 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.

[124] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[125] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for byzantine fault tolerance. In *OSDI*, pages 177–190, 2006.

[126] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO (2)*, pages 769–798, 2018.

[127] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO*, pages 13–25, 1998.

[128] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, pages 727–743, 2018.

[129] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[130] Curv. `https://www.curv.co/`, Accessed May 25, 2020.

[131] Anders Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In *European Symposium on Research in Computer Security*, pages 654–673. Springer, 2020.

[132] Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Ostergård. Fast threshold ECDSA with honest majority. In *SCN*, pages 382–400. Springer, 2020.

[133] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[134] Marc Damie, Florian Hahn, and Andreas Peter. A highly accurate query-recovery attack against searchable encryption using non-indexed documents. In *USENIX Security*, pages 143–160, 2021.

[135] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Santiago Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proceedings of the first ACM workshop on Smart energy grid security*, pages 75–80, 2013.

[136] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, 2014.

[137] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. SafetyPin: Encrypted backups with Human-Memorable secrets. In *OSDI*, pages 1121–1138, 2020.

[138] Emma Dauterman, Vivian Fang, Natacha Crooks, and Raluca Ada Popa. Reflections on trusting distributed trust. In *HotNets*, pages 38–45, 2022.

[139] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, pages 655–671, 2021.

[140] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An Encrypted Search System with Distributed Trust. In *USENIX OSDI*, pages 1101–1119, 2020.

[141] Emma Dauterman, Danny Lin, Henry Corrigan-Gibbs, and David Mazières. Accountable authentication with privacy protection: The larch system for universal login. In *OSDI*, pages 81–98, 2023.

[142] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *Security & Privacy*, pages 2450–2468. IEEE, 2022.

[143] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3):164–180, 2018.

[144] Gareth T Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the whatsapp end-to-end encrypted backup protocol. In *Annual International Cryptology Conference*, pages 330–361. Springer, 2023.

[145] Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. *Software: Practice and Experience*, 33(5):397–421, 2003.

[146] Leo de Castro and Keewoo Lee. VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers. In *USENIX Security (to appear)*, 2024.

[147] Roberta De Viti, Isaac Sheff, Noemi Glaeser, Baltasar Dinis, Rodrigo Rodrigues, Jonathan Katz, Bobby Bhattacharjee, Anwar Hithnawi, Deepak Garg, and Peter Druschel. CoVault: A secure analytics platform. *arXiv preprint arXiv:2208.03784*, 2022.

[148] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[149] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *CRYPTO*, 2018.

[150] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, 2020.

[151] Ioannis Demertzis and Charalampos Papamanthou. Fast searchable encryption with tunable locality. In *SIGMOD*, 2017.

[152] David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In *EUROCRYPT*, pages 425–455. Springer, 2018.

[153] David Derler, Stephan Krenn, Thomas Lorünser, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. Revisiting proxy re-encryption: forward secrecy, improved security, and applications. In *IACR International Workshop on Public Key Cryptography*, pages 219–250. Springer, 2018.

[154] Yvo Desmedt. Society and group oriented cryptography: A new concept. In *EUROCRYPT*, pages 120–127. Springer, 1987.

[155] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO*, pages 307–315, 1989.

[156] Arkajit Dey and Stephen Weis. PseudoID: Enhancing privacy in federated login. In *HotPETS workshop*, 2010.

[157] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to forget a secret. In *STOC*, pages 500–509, 1999.

[158] Marian Dietz and Stefano Tessaro. Fully malicious authenticated PIR. *IACR Cryptology ePrint Archive*, 2023.

[159] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. Resource-efficient byzantine fault tolerance. *IEEE transactions on computers*, 65(9):2807–2819, 2015.

[160] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *IEEE Security & Privacy*, pages 980–997. IEEE, 2018.

[161] Jack Doerner and Abhi Shelat. Scaling ORAM for secure computation. In *CCS*, pages 523–535. ACM, 2017.

[162] Cynthia Dwork, Moni Naor, Omer Reingold, and Larry Stockmeyer. Magic functions. In *Foundations of Computer Science*, pages 523–534. IEEE, 1999.

[163] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[164] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use Your Brain! Arithmetic 3PC for Any Modulus with Active Security. In *ITC*, volume 163 of *LIPIcs*, pages 5:1–5:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[165] Tariq Elahi, George Danezis, and Ian Goldberg. Privex: Private collection of traffic statistics for anonymous communication networks. In *ACM CCS*, pages 1068–1079, 2014.

[166] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[167] 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.

[168] Saba Eskandarian and Matei Zaharia. ObliDB: oblivious query processing for secure databases. *VLDB*, 13(2):169–183, 2019.

[169] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[170] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. In *European symposium on research in computer security*, pages 123–145. Springer, 2015.

[171] Andrés Fábrega, Carolina Ortega Pérez, Armin Namavari, Ben Nassi, Rachit Agarwal, and Thomas Ristenpart. Injection attacks against end-to-end encrypted applications. In *Security & Privacy*, pages 82–82. IEEE Computer Society, 2024.

[172] Corin Faife. Okta ends lapsus$ hack investigation, says breach lasted just 25 minutes. *The Verge*, April 2022. `https://www.theverge.com/2022/4/20/23034360/okta-lapsus-hack-investigation-breach-25-minutes`.

[173] Muhammad Faisal, Jerry Zhang, John Liagouris, Vasiliki Kalavri, and Mayank Varia. TVA: A multi-party computation system for secure and expressive time series analytics. In *USENIX Security*, pages 5395–5412, 2023.

[174] Brett Hemenway Falk, Steve Lu, and Rafail Ostrovsky. Durasift: A robust, decentralized, encrypted database supporting private searches with complex policy controls. In *WPES*, pages 26–36, 2019.

[175] Serge Fehr, Dennis Hofheinz, Eike Kiltz, and Hoeteck Wee. Encryption schemes secure against chosen-ciphertext selective opening attacks. In *EUROCRYPT*, pages 381–402. Springer, 2010.

[176] Daniel Fett, Ralf Küsters, and Guido Schmitz. Analyzing the BrowserID SSO system with primary identity providers using an expressive model of the web. In *European Symposium on Research in Computer Security*, pages 43–65. Springer, 2015.

[177] Daniel Fett, Ralf Küsters, and Guido Schmitz. Spresso: A secure, privacy-respecting single sign-on system for the web. In *CCS*, pages 1358–1369, 2015.

[178] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *EUROCRYPT*, pages 186–194. Springer, 1986.

[179] FIDO Alliance. FIDO alliance specifications: Overview. https://fidoalliance.org/specifications/, Accessed May 20, 2022.

[180] 5 advantages of a cloud-based EHR. https://www.carecloud.com/continuum/5-advantages-of-a-cloud-based-ehr-for-large-practices/.

[181] 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*, pages 215–222. IEEE, 2015.

[182] B. Ford, N. Gailly, L. Gasser, and P. Jovanovic. Collective Edwards-Curve Digital Signature Algorithm. Internet-Draft, June 2017.

[183] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. In *OSDI*, pages 233–248, 2006.

[184] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In *Compiler Construction: 23rd International Conference*, volume 8409, page 244, 2014.

[185] Ragnar Freij-Hollanti, Oliver W Gnilke, Camilla Hollanti, and David A Karpuk. Private information retrieval from coded databases with colluding servers. *SIAM Journal on Applied Algebra and Geometry*, 1(1):647–664, 2017.

[186] Kevin Edward Fu. *Group sharing and random access in cryptographic storage file systems*. PhD thesis, Massachusetts Institute of Technology, 1999.

[187] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and secure index with SGX. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 386–408. Springer, 2017.

[188] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. Sok: Cryptographically protected database search. In *Security & Privacy*, pages 172–191. IEEE, 2017.

[189] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT (2)*, pages 225–255, 2017.

[190] Jean-Baptiste Bédrune Gabriel Campana. Everybody be Cool, This is a Robbery! Blackhat, 2019. https://www.blackhat.com/us-19/briefings/schedule/#everybody-be-cool-this-is-a-robbery-16233.

[191] Adam Gągol and Damian Straszak. Threshold ecdsa for decentralized asset custody. Technical report, IACR Cryptology ePrint Archive, 2020.

[192] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *CRYPTO*, pages 563–592. Springer, 2016.

[193] Sergiu Gatlan. Cloudflare hacked using auth tokens stolen in okta attack, February 1 2024. `https://www.bleepingcomputer.com/news/security/cloudflare-hacked-using-auth-tokens-stolen-in-okta-attack/`, Accessed June 27, 2024.

[194] Shirley Gaw and Edward W Felten. Password management strategies for online accounts. In *SOUPS*, pages 44–55. ACM, 2006.

[195] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. Vanish: Increasing Data Privacy with Self-Destructing Data. In *USENIX Security*, 2009.

[196] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, pages 626–645. Springer, 2013.

[197] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *CCS*, pages 1179–1194, 2018.

[198] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *ACNS*, pages 156–174. Springer, 2016.

[199] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. Private queries in location based services: anonymizers are not necessary. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 121–132, 2008.

[200] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. In *USENIX Security*, pages 1069–1083, 2016.

[201] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, pages 640–658. Springer, 2014.

[202] Eu-Jin Goh et al. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[203] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *NDSS*, volume 3, pages 131–145, 2003.

[204] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987.

[205] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC*, pages 218–229. ACM, 1987.

[206] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[207] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[208] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[209] Dan Goodin. Millions of high-security crypto keys crippled by newly discovered flaw, 2017. `https://arstechnica.com/information-technology/2017/10/crypto-failure-cripples-millions-of-high-security-keys-750k-estonian-ids`.

[210] Michael T Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, 2011.

[211] Google. Key transparency design doc. `https://github.com/google/keytransparency/blob/master/docs/design_new.md`.

[212] S Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server ORAM. In *ASIACRYPT*, pages 141–157. Springer, 2018.

[213] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *European Workshop on Systems Security*, pages 1–6, 2017.

[214] Matthew Green. Is Apple's Cloud Key Vault a crypto backdoor?, 2016. `https://blog.cryptographyengineering.com/2016/08/13/is-apples-cloud-key-vault-crypto/`.

[215] Matthew D Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *Security & Privacy*. IEEE, 2015.

[216] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Roger Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *SRDS*, pages 189–198. IEEE, 2006.

[217] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326. Springer, 2016.

[218] Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 253–280. Springer, 2015.

[219] Jens Groth and Victor Shoup. Design and analysis of a distributed ECDSA signing service. *IACR Cryptology ePrint Archive*, 2022.

[220] Jens Groth and Victor Shoup. On the security of ECDSA with additive key derivation and presignatures. In *EUROCRYPT*, 2022.

[221] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. In *USENIX Security*, 2022.

[222] 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 Symposium*, 2020.

[223] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *ACM CCS*, pages 315–331, 2018.

[224] 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*, pages 1067–1083. IEEE, 2019.

[225] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *Security & Privacy*. IEEE, 2019.

[226] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *HotOS*, pages 162–168, 2017.

[227] 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.

[228] 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 *Security & Privacy*. IEEE, 2018.

[229] Tianyao Gu, Yilei Wang, Bingnan Chen, Afonso Tinoco, Elaine Shi, and Ke Yi. Efficient oblivious sorting and shuffling for hardware enclaves. *IACR Cryptology ePrint Archive*, 2023.

[230] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *ACM CCS*, pages 361–378, 2019.

[231] Zichen Gui, Kenneth G Paterson, and Sikhar Patranabis. Rethinking searchable symmetric encryption. In *Security & Privacy*, pages 1401–1418. IEEE, 2023.

[232] Zichen Gui, Kenneth G Paterson, Sikhar Patranabis, and Bogdan Warinschi. SWiSSSE: System-wide security for searchable symmetric encryption. *PETS*, 2024.

[233] Zichen Gui, Kenneth G Paterson, and Tianxin Tang. Security analysis of MongoDB queryable encryption. In *USENIX Security*, pages 7445–7462, 2023.

[234] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 2017.

[235] Chengqian Guo, Jingqiang Lin, Quanwei Cai, Fengjun Li, Qiongxiao Wang, Jiwu Jing, Bin Zhao, and Wei Wang. UPPRESSO: Untraceable and unlinkable privacy-preserving single sign-on services. *arXiv preprint arXiv:2110.10396*, 2021.

[236] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *USENIX NSDI*, pages 91–107, 2016.

[237] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[238] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *ESSoS*, pages 44–60. Springer, 2018.

[239] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[240] Subir Halder and Mauro Conti. Crypsh: A novel iot data protection scheme based on bgn cryptosystem. *IEEE Transactions on Cloud Computing*, 2021.

[241] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. Private anonymous data access. In *EUROCRYPT*, pages 244–273. Springer, 2019.

[242] Ariel Hamlin, Nabil Schear, Emily Shen, Mayank Varia, Sophia Yakoubov, and Arkady Yerukhimovich. *Cryptography for big data security*. Taylor & Francis LLC, CRC Press, 2016.

[243] Sven Hammann, Ralf Sasse, and David Basin. Privacy-preserving OpenID connect. In *ASIACCS*, pages 277GS22–289, 2020.

[244] Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim. A bad dream: Subverting trusted platform module while you are sleeping. In *USENIX Security*, pages 1229–1246, 2018.

[245] Lucjan Hanzlik, Julian Loss, and Benedikt Wagner. Token meets wallet: Formalizing privacy and revocation for fido2. In *Security & Privacy*, pages 1491–1508. IEEE, 2023.

[246] SM Haque, Matthew Wright, and Shannon Scielzo. A study of user password strategy for multiple accounts. In *Data and application security and privacy*, pages 173–176. ACM, 2013.

[247] Matúš Harvan, Samuel Kimoto, Thomas Locher, Yvonne-Anne Pignolet, and Johannes Schneider. Processing encrypted and compressed time series data. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1053–1062. IEEE, 2017.

[248] Ryan Henry. Polynomial batch codes for efficient IT-PIR. *PETS Symposium*, 2016.

[249] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *SOSP*, pages 396–416, 2023.

[250] Alexandra Henzinger, Matthew M Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In *USENIX Security*, pages 3889–3905, 2023.

[251] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.

[252] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A Yavuz. Mose: Practical multi-user oblivious storage via secure enclaves. In *CODASPY*. ACM, 2020.

[253] 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*, (1):172–191, 2019.

[254] Thang Hoang, Attila A Yavuz, F Betül Durak, and Jorge Guajardo. Oblivious dynamic searchable encryption on distributed cloud systems. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 113–130. Springer, 2018.

[255] Thang Hoang, Attila Altay Yavuz, and Jorge Guajardo. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *CCS*, pages 302–313. ACM, 2016.

[256] Dennis Hofheinz and Andy Rupp. Standard versus selective opening security: separation and equivalence results. In *TCC*, pages 591–615, 2014.

[257] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle 2: A low-latency transparency log system. In *Security & Privacy*, pages 285–303. IEEE, 2021.

[258] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. Ghostor: Toward a secure data-sharing system from decentralized trust. In *NSDI*, pages 851–877, 2020.

[259] Identity Theft Resource Center. *ITRC Annual Data Breach Report*, 2023 edition. `https://www.idtheftcenter.org/publication/2023-data-breach-report/`.

[260] InfluxDB. `https://www.influxdata.com/`, Accessed August 2021.

[261] InfluxData. Time series database explained. https://www.influxdata.com/time-series-database/, Accessed August 2021.

[262] InfluxDB. https://www.influxdata.com/.

[263] 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.

[264] Intel. Strengthen enclave trust with attestation. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/attestation-services.html, Accessed 5 February 2020.

[265] Marios Isaakidis, Harry Halpin, and George Danezis. Unlimitid: Privacy-preserving federated identity management using algebraic macs. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, pages 139–142, 2016.

[266] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *Cryptographers' Track at the RSA Conference*, pages 90–107. Springer, 2016.

[267] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, 2004.

[268] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *STOC*, pages 21–30, 2007.

[269] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, volume 20, page 12. Citeseer, 2012.

[270] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

[271] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. Orca: FSS-based Secure Training and Inference with GPUs. In *Security & Privacy*, pages 63–63. IEEE Computer Society, 2024.

[272] Peipei Jiang, Qian Wang, Jianhao Cheng, Cong Wang, Lei Xu, Xinyu Wang, Yihao Wu, Xiaoyuan Li, and Kui Ren. Boomerang: Metadata-private messaging under hardware trust. In *NSDI*, pages 877–899, 2023.

[273] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, volume 3, pages 29–42, 2003.

[274] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, pages 258–274. Springer, 2013.

[275] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *CCS*, pages 965–976. ACM, 2012.

[276] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: resource-efficient byzantine fault tolerance. In *EuroSys*, pages 295–308, 2012.

[277] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *NDSS*, 2019.

[278] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *security & Privacy*, pages 895–913. IEEE, 2016.

[279] Alan F Karr, Xiaodong Lin, Ashish P Sanil, and Jerome P Reiter. Secure regression on distributed databases. *Journal of Computational and Graphical Statistics*, 14(2):263–279, 2005.

[280] Jonathan Katz and Andrew Y Lindell. Aggregate message authentication codes. In *Cryptographers' Track at the RSA Conference*, pages 155–169. Springer, 2008.

[281] Jonathan Katz and Andrew Y Lindell. Aggregate message authentication codes. In *Cryptographers' Track at the RSA Conference*, pages 155–169, 2008.

[282] Bernhard Kauer. Oslo: Improving the security of trusted computing. In *USENIX Security*, volume 24, page 173, 2007.

[283] Darya Kaviani, Sijun Tan, Pravein Govindan Kannan, and Raluca Ada Popa. Flock: A framework for deploying on-demand distributed trust. In *OSDI (to appear)*, 2024.

[284] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.

[285] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *ACM CCS*, pages 1575–1590. ACM, 2020.

[286] Keybase. `https://keybase.io/`, Accessed May 26, 2020.

[287] Ali Khoshgozaran and Cyrus Shahabi. Private information retrieval techniques for enabling location privacy in location-based services. In *Privacy in Location-Based Applications*, pages 59–83. Springer, 2009.

[288] Aggelos Kiayias, Ozgur Oksuz, Alexander Russell, Qiang Tang, and Bing Wang. Efficient encrypted keyword search for multi-user data sharing. In *ESORICS*, pages 173–195. Springer, 2016.

[289] Beom Heyn Kim and David Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *Security & Privacy*, pages 880–896. IEEE, 2015.

[290] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *USENIX Security*, 2021.

[291] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *Security & Privacy*. IEEE, 2019.

[292] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: attacks on encrypted databases beyond the uniform query distribution. In *Security & Privacy*, pages 1223–1240. IEEE, 2020.

[293] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. In *Security & Privacy*, pages 1502–1519. IEEE, 2021.

[294] Stavros Korokithakis. Writing a full-text search engine using bloom filters, December 2013. `https://www.stavros.io/posts/bloom-filter-search-engine/`.

[295] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *SOSP*, 41(6):45–58, 2007.

[296] Ivan Krstic. Behind the scenes with iOS security, 2016. `https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf`.

[297] Krypton. kr-u2f. `https://github.com/kryptco/kr-u2f`, Accessed May 17, 2022.

[298] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian. Parametric trojans for fault-injection attacks on cryptographic hardware. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sept. 2014.

[299] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E Culler. JEDI: Many-to-many end-to-end encryption and key delegation for IoT. In *USENIX Security*, pages 1519–1536, 2019.

[300] Mu-Hsing Kuo. Opportunities and challenges of cloud computing to improve health care services. *Journal of medical Internet research*, 2011.

[301] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *PETS*, pages 175–191. Springer, 2011.

[302] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *SODA*. SIAM, 2012.

[303] Mehmet Kuzu, Mohammad Saiful Islam, and Murat Kantarcioglu. Efficient similarity search over encrypted data. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1156–1167. IEEE, 2012.

[304] SCIPR Lab. libsnark. `https://github.com/scipr-lab/libsnark`, Accessed May 30, 2022.

[305] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *Security & Privacy*, pages 297–314. IEEE, 2018.

[306] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[307] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[308] Butler Lampson and David B Lomet. A new presumed commit optimization for two phase commit. In *VLDB*, volume 93, pages 630–640, 1993.

[309] Adam Langley, Emilia Kasper, and Ben Laurie. Certificate transparency. *Internet Engineering Task Force*, 2013. `https://tools.ietf.org/html/rfc6962`.

[310] Ledger vault. `https://www.ledger.com/vault`, Accessed May 25, 2020.

[311] 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.

[312] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *EuroSys*. ACM, 2020.

[313] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[314] Jiwon Lee, Jaekyoung Choi, Jihye Kim, and Hyunok Oh. SAVER: Snark-friendly, additively-homomorphic, and verifiable encryption and decryption with rerandomization. *Cryptology ePrint Archive*, 2019.

[315] 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.

[316] 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*, pages 557–574, 2017.

[317] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. Hintless single-server private information retrieval. In *CRYPTO (to appear)*, 2024.

[318] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Authenticated index structures for aggregation queries. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):1–35, 2010.

[319] Feng Li, Jianfeng Ma, Yinbin Miao, Ximeng Liu, Jianting Ning, and Robert H Deng. A survey on searchable symmetric encryption. *ACM Computing Surveys*, 56(5):1–42, 2023.

[320] Jinyuan Li, Maxwell N Krohn, David Mazieres, and Dennis E Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, volume 4, pages 9–9, 2004.

[321] Mingyu Li, Jinhao Zhu, Tianxu Zhang, Cheng Tan, Yubin Xia, Sebastian Angel, and Haibo Chen. Bringing decentralized search to decentralized services. In *OSDI*, pages 331–347, 2021.

[322] S. Li, C. Man, and J. Watson. Delegated Distributed Mappings. Internet-Draft draft-watson-dinrg-delmap-02, Internet Engineering Task Force, April 2019. `https://tools.ietf.org/html/draft-watson-dinrg-delmap-02`.

[323] Xiang Li, Yunqian Luo, and Mingyu Gao. BULKOR: Enabling Bulk Loading for Path ORAM. In *Security & Privacy*, pages 103–103. IEEE Computer Society, 2024.

[324] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. Secrecy: Secure collaborative analytics on secret-shared data. *arXiv preprint arXiv:2102.01048*, 2021.

[325] Yehuda Lindell. Fast secure two-party ECDSA signing. In *CRYPTO*, pages 613–644. Springer, 2017.

[326] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer International Publishing, 2017.

[327] Ryan Little, Lucy Qin, and Mayank Varia. Secure account recovery for a privacy-preserving web service. *IACR Cryptology ePrint Archive*, 2024.

[328] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. GhostRider: A hardware-software system for memory trace oblivious computation. *ASPLOS*, 50(4):87–101, 2015.

[329] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[330] Dongli Liu, Wei Wang, Peng Xu, Laurence T Yang, Bo Luo, and Kaitai Liang. d-DSE: Distinct dynamic searchable encryption resisting volume leakage in encrypted databases. 2024.

[331] Jian LIU, Jingyu LI, Di WU, et al. PIRANA: Faster multi-query PIR via constant-weight codes. In *Security & Privacy*, volume 43. IEEE, 2024.

[332] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. XFT: Practical fault tolerance beyond crashes. In *OSDI*, pages 485–500, 2016.

[333] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *SOSP*, pages 80–96, 2019.

[334] 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*, pages 199–213, 2013.

[335] Tammy Lovell. Swedish healthcare advice line stored 2.7 million patient phone calls on unprotected web server, February 20 2019. `https://www.healthcareitnews.com/news/swedish-healthcare-advice-line-stored-27-million-patient-phone-calls-unprotected-web-server`.

[336] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396. Springer, 2013.

[337] Joshua Lund. Technology preview for secure value recovery, 2019. `https://signal.org/blog/secure-value-recovery/`.

[338] Vadim Lyubashevsky and Gregory Neven. One-shot verifiable encryption from lattices. In *EUROCRYPT*, pages 293–323. Springer, 2017.

[339] Ethan MacBrough. Cobalt: BFT governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

[340] Philip MacKenzie and Michael K Reiter. Two-party generation of DSA signatures. In *CRYPTO*, pages 137–154. Springer, 2001.

[341] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Privacy and access control for outsourced personal records. In *Security & Privacy*, pages 341–358. IEEE, 2015.

[342] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Maliciously secure multi-client ORAM. In *ACNS*, pages 645–664. Springer, 2017.

[343] Sujaya Maiyya, Seif Ibrahim, Caitlin Scarberry, Divyakant Agrawal, Amr El Abbadi, Huijia Lin, Stefano Tessaro, and Victor Zakhary. QuORAM: Quorum-replicated fault tolerant ORAM datastore. In *USENIX Security*, pages 3665–3682, 2022.

[344] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. *Proceedings of the ACM on Management of Data*, 1(4):1–25, 2023.

[345] Kathryn I Marko, Jill M Krapf, Andrew C Meltzer, Julia Oh, Nihar Ganju, Anjali G Martinez, Sheetal G Sheth, and Nancy D Gaba. Testing the feasibility of remote patient monitoring in prenatal care using a mobile app and connected devices: a prospective observational trial. *JMIR research protocols*, 5(4):e200, 2016.

[346] Moxie Marlinspike. The difficulty of private contact discovery, January 2014. `https://signal.org/blog/contact-discovery/`.

[347] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[348] 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.

[349] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. BITE: Bitcoin lightweight client privacy using trusted execution. In *USENIX Security*, pages 783–800, 2019.

[350] 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*, pages 1583–1600, 2017.

[351] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Designs, Codes and Cryptography*, 87(9):2139–2164, 2019.

[352] Travis Mayberry, Erik-Oliver Blass, and Guevara Noubir. Multi-User Oblivious RAM Secure Against Malicious Servers. *IACR Cryptology ePrint Archive*, 2015:121, 2015.

[353] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.

[354] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 10(1), 2013.

[355] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.

[356] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. *arXiv preprint arXiv:1508.06110*, 2015.

[357] Memtier benchmark. `https://github.com/RedisLabs/memtier_benchmark`.

[358] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server PIR via FHE composition. In *Security & Privacy*, pages 930–947. IEEE, 2022.

[359] Samir Jordan Menon and David J Wu. YPIR: High-Throughput Single-Server PIR with Silent Preprocessing. In *USENIX Security (to appear)*, 2024.

[360] Ralph Merkle. A certified digital signature. In *CRYPTO*, pages 218–238. Springer, 1989.

[361] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Security & Privacy*, pages 279–296. IEEE, 2018.

[362] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[363] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[364] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *USENIX Security*, 2020.

[365] Payman Mohassel and Peter Rindal. ABY3: A mixed protocol framework for machine learning. In *ACM CCS*, pages 35–52, 2018.

[366] Priyanka Mondal, Javad Ghareh Chamani, Ioannis Demertzis, and Dimitrios Papadopoulos. I/O-efficient dynamic searchable encryption meets forward & backward privacy. 2024.

[367] MongoDB. Time Series Data and MongoDB. `https://www.mongodb.com/blog/post/time-series-data-and-mongodb-part-1-introduction`, Accessed August 2021.

[368] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS# 1: RSA cryptography specifications version 2.2. *Internet Engineering Task Force, Request for Comments*, 8017:72, 2016.

[369] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-Based One-Time Password Algorithm. RFC 6238, May 2011.

[370] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response efficient single-server PIR. In *CCS*, pages 2292–2306, 2021.

[371] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. In *Security & Privacy*, pages 437–452. IEEE, 2023.

[372] 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 *Security and Privacy*. IEEE, 2020.

[373] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2008.

[374] Ellen Nakashima. Russian government hackers penetrated DNC, stole opposition research on Trump, June 14 2016. `https://www.washingtonpost.com/world/national-security/russian-government-hackers-penetrated-dnc-stole-opposition-research-on-trump/2016/06/14/cf006cb4-316e-11e6-8ff7-7b6c1998b7a0_story.html`.

[375] Muhammad Naveed. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. *IACR Cryptology ePrint Archive*, 2015:668, 2015.

[376] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. Dynamic searchable encryption via blind storage. In *Security & Privacy*, pages 639–654. IEEE, 2014.

[377] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of Coppersmith's attack: Practical factorization of widely used RSA moduli. In *CCS*, pages 1631–1648. ACM, 2017.

[378] Nicholas Ngai, Ioannis Demertzis, Javad Ghareh Chamani, and Dimitrios Papadopoulos. Distributed & scalable oblivious sorting and shuffling. In *Security & Privacy*, pages 153–153. IEEE Computer Society, 2024.

[379] Jonas Nick, Tim Ruffing, and Yannick Seurin. MuSig2: Simple two-round schnorr multi-signatures. In *CRYPTO*, pages 189–221. Springer, 2021.

[380] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. MuSig-DN: Schnorr multi-signatures with verifiably deterministic nonces. In *CCS*, pages 1717–1731, 2020.

[381] Hao Nie, Wei Wang, Peng Xu, Xianglong Zhang, Laurence T Yang, and Kaitai Liang. Query recovery from easy to hard: Jigsaw attack against SSE. 2024.

[382] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In *USENIX Security Symposium*, 1998.

[383] Department of health and human services. Medicare and Medicaid Programs; Policy and Regulatory Revisions in Response to the COVID-19 Public Health Emergency. `https://www.cms.gov/files/document/covid-final-ifc.pdf`.

[384] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319, 2014.

[385] Taku Onodera and Tetsuo Shibuya. Succinct oblivious ram. *arXiv preprint arXiv:1804.08285*, 2018.

[386] OpenEnclave. https://github.com/openenclave/openenclave.

[387] OpenHAB. https://www.openhab.org/, Accessed August 2021.

[388] OpenTSDB. http://opentsdb.net/, Accessed August 2021.

[389] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for SGX enclaves. In *EuroSys*. ACM, 2017.

[390] Chris Orsini, Alessandra Scafuro, and Tanner Verber. How to recover a cryptographic secret from the cloud. *IACR Cryptology ePrint Archive*, 2023.

[391] Charlie Osborne. Fortune 500 company leaked 264gb of client, payment data, June 7 2019. https://www.zdnet.com/article/veteran-fortune-500-company-leaked-264gb-in-client-payment-data/.

[392] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523. ACM, 1990.

[393] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security*, 2021.

[394] Simon Oya and Florian Kerschbaum. IHOP: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization. In *USENIX Security*, pages 2407–2424, 2022.

[395] Carly Page. Okta admits hackers accessed data on all customers during recent breach, November 29 2023. https://techcrunch.com/2023/11/29/okta-admits-hackers-accessed-data-on-all-customers-during-recent-breach/.

[396] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *USENIX OSDI*, pages 587–602, 2016.

[397] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *International conference on information and communications security*, pages 1–15. Springer, 2007.

[398] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *Security & Privacy*, pages 359–374. IEEE, 2014.

[399] Vasilis Pappas, Mariana Raykova, Binh Vo, Steven M Bellovin, and Tal Malkin. Private search in the real world. In *ACSAC*, pages 83–92, 2011.

[400] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Security & Privacy*, pages 238–252. IEEE, 2013.

[401] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *Security & Privacy*. IEEE, 2011.

[402] `https://github.com/aricrocuta/oram2pc`, Accessed April 14, 2020.

[403] Ryan Paul. Infineon DRM/encryption chip succumbs to physical attack. Ars Technica, 2010. `https://arstechnica.com/information-technology/2010/02/infineon-drmencryption-chip-succumbs-to-physical-attack/`.

[404] Radia Perlman. File system design with assured delete. In *Third IEEE International Security in Storage Workshop (SISW'05)*, pages 6–pp. IEEE, 2005.

[405] Zachary NJ Peterson, Randal C Burns, Joseph Herring, Adam Stubblefield, and Aviel D Rubin. Secure deletion for a versioning file system. In *FAST*, volume 5, 2005.

[406] 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.

[407] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. *VLDB*, 12(11):1664–1678, 2019.

[408] 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.

[409] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *NSDI*. USENIX, 2018.

[410] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. 2020.

[411] Geong Sen Poh, Ji-Jian Chin, Wei-Chuen Yau, Kim-Kwang Raymond Choo, and Moesfa Soeheila Mohamad. Searchable symmetric encryption: designs and challenges. *ACM Computing Surveys (CSUR)*, 50(3):1–37, 2017.

[412] Raluca A Popa and Nickolai Zeldovich. Multi-key searchable encryption. *IACR Cryptology ePrint Archive*, 2013:508, 2013.

[413] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[414] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *EuroSys*, pages 1–14, 2015.

[415] David Pouliot and Charles V Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *CCS*, pages 1341–1352, 2016.

[416] Adam Powers. FIDO TechNotes: The truth about attestation. `https://fidoalliance.org/fido-technotes-the-truth-about-attestation/`, Accessed May 25, 2020.

[417] PreVeil. `https://www.preveil.com/`, Accessed May 26, 2020.

[418] Prevounce. Examples of remote patient monitoring: 9 top patient applications. `https://blog.prevounce.com/examples-of-remote-patient-monitoring-9-top-patient-applications`.

[419] Prevounce. Remote Monitoring of Peak Expiratory Flow, October 2020. `https://blog.prevounce.com/remote-monitoring-of-peak-expiratory-flow`.

[420] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A secure database using SGX. In *Security & Privacy*, pages 264–278. IEEE, 2018.

[421] Prometheus. `https://prometheus.io/`, Accessed August 2021.

[422] Martin Raab and Angelika Steger. "Balls into bins"—a simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[423] Charles Rackoff and Daniel R Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO*, pages 433–444, 1991.

[424] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *Security & Privacy*. IEEE, 2021.

[425] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. *arXiv preprint arXiv:2008.00332*, 2020.

[426] MV Ramakrishna. Computing the probability of hash table/urn overflow. *Communications in Statistics-Theory and Methods*, 16(11):3343–3353, 1987.

[427] Mariana Raykova, Binh Vo, Steven M Bellovin, and Tal Malkin. Secure anonymous database search. In *Workshop on Cloud computing security*, pages 115–126, 2009.

[428] Joel Reardon, David Basin, and Srdjan Capkun. Sok: Secure data deletion. In *Security & Privacy*, pages 301–315. IEEE, 2013.

[429] Joel Reardon, Hubert Ritzdorf, David Basin, and Srdjan Capkun. Secure data deletion from persistent media. In *CCS*, pages 271–284, 2013.

[430] Redis. `https://redis.io/`.

[431] Corinne Reichert. Payroll data for 29,000 facebook employees stolen, December 13 2019. https://www.cnet.com/news/payroll-data-of-29000-facebook-employees-reportedly-stolen/.

[432] 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.

[433] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. Heda: multi-attribute unbounded aggregation over homomorphically encrypted database. *Proceedings of the VLDB Endowment*, 16(4):601–614, 2022.

[434] 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.

[435] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *FAST*, volume 2, pages 15–30, 2002.

[436] Panagiotis Rizomiliotis and Stefanos Gritzalis. ORAM based forward privacy preserving dynamic searchable symmetric encryption schemes. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, pages 65–76. ACM, 2015.

[437] Daniel S Roche, Adam Aviv, and Seung Geol Choi. A practical oblivious map data structure with secure deletion and history independence. In *Security & Privacy*, pages 178–197. IEEE, 2016.

[438] Emma Roth. LastPass' latest data breach exposed some customer information. *The Verge*, November 2022. https://www.theverge.com/2022/11/30/23486902/lastpass-hackers-customer-information-breach.

[439] Adam Rowe. Study reveals average person has 100 passwords, November 2021. https://tech.co/password-managers/how-many-passwords-average-person.

[440] Mark D Ryan. Enhanced certificate transparency and end-to-end encrypted mail. *IACR Cryptology ePrint Archive*, 2013.

[441] SafeNet Luna network hardware security modules A700 - cryptographic accelerator. https://www.insight.com/en_US/shop/product/908-000366-001-000/GEMALTO/908-000366-001-000/SafeNetLunaNetworkHardwareSecurityModulesA700-Cryptographicaccelerator-GigE-1U-rack-mountable/, Accessed February 5, 2020.

[442] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *Security & Privacy*, pages 198–217. IEEE, 2016.

[443] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID connect core 1.0 incorporating errata set 1. `https://openid.net/specs/openid-connect-core-1_0.html`, November 2014.

[444] Luca Santini, Karim Mahfouz, Valentina Schirripa, Nicola Danisi, Michelangelo Leone, Gloria Mangone, Monica Campari, Sergio Valsecchi, and Fabrizio Ammirati. Preliminary experience with a novel multisensor algorithm for heart failure monitoring: The heartlogic index. *Clinical case reports*, 6(7):1317, 2018.

[445] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. volume 2018.

[446] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks. In *CCS*, pages 3328–3342, 2023.

[447] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*. ACM, 2019.

[448] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*. Springer, 2017.

[449] Sacha Servan-Schreiber, Simon Langowski, and Srinivas Devadas. Private approximate nearest neighbor search with sublinear communication. In *Security & Privacy*, pages 911–929. IEEE, 2022.

[450] 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.

[451] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. Secure sharing of partially homomorphic encrypted iot data. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–14, 2017.

[452] Hossein Shafagh, Anwar Hithnawi, Andreas Dröscher, Simon Duquennoy, and Wen Hu. Talos: Encrypted query processing for the internet of things. In *Proceedings of the 13th ACM conference on embedded networked sensor systems*, pages 197–210, 2015.

[453] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[454] Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Encountering stronger password requirements: user attitudes and behaviors. In *SOUPS*, page 2. ACM, 2010.

[455] Elaine Shi, TH Hubert Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *Proc. NDSS*, volume 2, pages 1–17. Citeseer, 2011.

[456] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.

[457] Nigel P Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In *IMA International Conference on Cryptography and Coding*, pages 342–366. Springer, 2019.

[458] Smartcar. `https://smartcar.com/`, Accessed August 2021.

[459] Solana. Solana decentralized exchange. `https://soldex.ai/wp-content/uploads/2021/07/Soldex.ai-whitepaper-.pdf`.

[460] SolarNetwork. `https://solarnetwork.net/`, Accessed August 2021.

[461] Kevin Solmssen. Querying time-series data privately. 2022.

[462] Solokeys. Solo. `https://github.com/solokeys/solo`, Accessed February 5, 2020.

[463] Solokeys. `https://solokeys.com/`, Accessed February 5, 2020.

[464] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security & Privacy*, pages 44–55. IEEE, 2000.

[465] SpiderOak. `https://spideroak.com/`, Accessed May 26, 2020.

[466] Markus Stadler. Publicly verifiable secret sharing. In *EUROCRYPT*, pages 190–199. Springer, 1996.

[467] Statista. Number of smartphone users worldwide from 2016 to 2021. `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`.

[468] Statista. Number of smartphones sold to end users worldwide from 2007 to 2020. `https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/`, Accessed 23 May 2020.

[469] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, volume 71, pages 72–75, 2014.

[470] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *CCS*, pages 247–258. ACM, 2013.

[471] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Security & Privacy*, pages 253–267. IEEE, 2013.

[472] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. In *NDSS*, 2012.

[473] 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*, pages 299–310. ACM, 2013.

[474] Myung-kyung Suh, Chien-An Chen, Jonathan Woodbridge, Michael Kai Tu, Jung In Kim, Ani Nahapetian, Lorraine S Evangelista, and Majid Sarrafzadeh. A remote patient monitoring system for congestive heart failure. *Journal of medical systems*, 35(5):1165–1179, 2011.

[475] Hua Sun and Syed Ali Jafar. The capacity of robust private information retrieval with colluding databases. *IEEE Transactions on Information Theory*, 64(4):2361–2370, 2017.

[476] Sync. `https://www.sync.com/`, Accessed May 26, 2020.

[477] Akira Takahashi and Greg Zaverucha. Verifiable encryption from MPC-in-the-Head. *Cryptology ePrint Archive*, 2021.

[478] Roberto Tamassia. Authenticated data structures. In *European symposium on algorithms*, pages 2–5. Springer, 2003.

[479] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. CRYPTGPU: Fast privacy-preserving machine learning on the gpu. 2021.

[480] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, 2017.

[481] Qiang Tang. Nothing is for free: security in searching shared and encrypted data. *Transactions on Information Forensics and Security*, 9(11):1943–1952, 2014.

[482] Yang Tang, Patrick PC Lee, John CS Lui, and Radia Perlman. FADE: Secure overlay cloud storage with file assured deletion. In *International Conference on Security and Privacy in Communication Systems*, pages 380–397. Springer, 2010.

[483] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design Test of Computers*, 27(1), Jan 2010.

[484] Timescale. `https://www.timescale.com/`, Accessed August 2021.

[485] Timescale. What is time-series data? `https://docs.timescale.com/timescaledb/latest/overview/what-is-time-series-data/#what-is-time-series-data`, Accessed August 2021.

[486] Timescale. How Everactive powers a dense sensor network with virtually no power at all, February 2011. `https://blog.timescale.com/blog/how-everactive-powers-a-dense-sensor-network-with-virtually-no-power-at-all/`.

[487] Afonso Tinoco, Sixiang Gao, and Elaine Shi. EnigMap:external-memory oblivious map for secure enclaves. In *USENIX Security*, pages 4033–4050, 2023.

[488] Tiny AES in C. `https://github.com/kokke/tiny-AES-c`, Accessed May 24, 2020.

[489] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *CCS*, pages 1299–1316, 2019.

[490] Shruti Tople, Yaoqi Jia, and Prateek Saxena. PRO-ORAM: Practical read-only oblivious RAM. In *RAID*, 2019.

[491] 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`, Accessed June 25, 2024.

[492] Tresorit. `https://tresorit.com/`, Accessed May 26, 2020.

[493] Stephen Lyle Tu, M Frans Kaashoek, Samuel R Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. 2013.

[494] Unbound tech. `https://www.unboundtech.com/`, Accessed May 25, 2020.

[495] 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*, pages 991–1008, 2018.

[496] 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 *Security & Privacy*. IEEE, 2020.

[497] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[498] 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.

[499] 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 *Security & Privacy*. IEEE, 2019.

[500] 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.

[501] Verizon. *DBIR Data Breach Investigations Report*, 2024 edition. `https://www.verizon.com/business/resources/T3c/reports/2024-dbir-data-breach-investigations-report.pdf`.

[502] Verizon. *DBIR Data Breach Investigations Report*, 2022 edition. https://www.verizon.com/business/resources/T3cd/reports/dbir/2022-data-breach-investigations-report-dbir.pdf.

[503] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *PETS*, 2019(3):370–388, 2019.

[504] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–18, 2019.

[505] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. SHORTSTACK: Distributed, fault-tolerant, oblivious data access. In *OSDI*, pages 719–734, 2022.

[506] W3C. Web authentication: An api for accessing public key credentials level 2, April 2021. https://www.w3.org/TR/webauthn-2/, Accessed 20 May 2022.

[507] Shabsi Walfish. Google Cloud Key Vault Service. Google, 2018. https://developer.android.com/about/versions/pie/security/ckv-whitepaper.

[508] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *NSDI*, pages 299–313, 2017.

[509] Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and Abhi Shelat. Blind certificate authorities. In *Security & Privacy*, pages 1015–1032. IEEE, 2019.

[510] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[511] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, pages 21–37, 2017.

[512] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *ACM CCS*, pages 215–226, 2014.

[513] Yinghao Wang, Xuanming Liu, Jiawen Zhang, Jian Liu, and Xiaohu Yang. Crust: Verifiable and efficient private information retrieval with sublinear online time. *IACR Cryptology ePrint Archive*, 2023.

[514] Jess Weatherbed. Lastpass reveals attackers stole password vault data by hacking an employee's home computer, 2023. https://www.theverge.com/2023/2/28/23618353/lastpass-security-breach-disclosure-password-vault-encryption-update.

[515] WhatsApp. Security of end-to-end encrypted backups, 2021. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf.

[516] Meredith Whittaker and Joshua Lund. Technology preview for secure value recovery, 2023. `https://signal.org/blog/signal-is-expensive/`.

[517] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, pages 977–988. ACM, 2012.

[518] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the sixth conference on Computer systems*, pages 123–138, 2011.

[519] Lei Xu, Leqian Zheng, Chengzhi Xu, Xingliang Yuan, and Cong Wang. Leakage-abuse attacks against forward and backward private searchable symmetric encryption. In *CCS*, pages 3003–3017, 2023.

[520] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security & Privacy*. IEEE, 2015.

[521] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. Efficient online-friendly two-party ECDSA signature. In *CCS*, pages 558–573, 2021.

[522] Wanli Xue, Chenwen Luo, Guohao Lan, Rajib Rana, Wen Hu, and Aruna Seneviratne. Kryptein: a compressive-sensing-based encryption scheme for the internet of things. In *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 169–180. IEEE, 2017.

[523] Shota Yamada, Nuttapong Attrapadung, Bagus Santoso, Jacob CN Schuldt, Goichiro Hanaoka, and Noboru Kunihiro. Verifiable predicate encryption and applications to CCA security and anonymous predicate authentication. In *PKC*, pages 243–261. Springer, 2012.

[524] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. A2: Analog malicious hardware. In *Security and Privacy*. IEEE, May 2016.

[525] Andrew C Yao. Protocols for secure computations. In *FOCS*, pages 160–164. IEEE, 1982.

[526] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167. IEEE, 1986.

[527] Wei-Zhu Yeoh, Michal Kepkowski, Gunnar Heide, Dali Kaafar, and Lucjan Hanzlik. Fast IDentity online with anonymous credentials (FIDO-AC). In *USENIX Security*, pages 3029–3046, 2023.

[528] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, pages 253–267, 2003.

[529] Eric Yuan. Zoom acquires keybase and announces goal of developing the most broadly used enterprise end-to-end encryption offering, May 7 2020. `https://blog.zoom.us/wordpress/2020/05/07/zoom-acquires-keybase-and-announces-goal-of-developing-the-most-broadly-used-enterprise-end-to-end-encryption-offering/`.

[530] YubiHSM 2. `https://www.yubico.com/product/yubihsm-2`, Accessed 5 February 2020.

[531] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. Zombie: Middleboxes that don't snoop. In *NSDI*, pages 1917–1936, 2024.

[532] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1919–1938, 2020.

[533] Xianglong Zhang, Wei Wang, Peng Xu, Laurence T Yang, and Kaitai Liang. High recovery with fewer injections: practical binary volumetric injection attacks against dynamic searchable encryption. In *USENIX Security*, pages 5953–5970, 2023.

[534] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*, pages 707–720, 2016.

[535] Zhiyi Zhang, Michal Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. EL PASSO: Efficient and lightweight privacy-preserving single sign on. *PoPETS*, 2021(2):70–87, 2021.

[536] 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*, pages 283–298, 2017.

[537] Yandong Zheng, Rongxing Lu, Yunguo Guan, Jun Shao, and Hui Zhu. Efficient and privacy-preserving similarity range query over encrypted time series data. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[538] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server PIR with sublinear server computation. IEEE, 2024.

[539] ZoKrates. ZoKrates. `https://github.com/Zokrates/ZoKrates`, Accessed May 30, 2022.