Stony Brook University

Oblivious Remote Data Access Made Practical

A Dissertation Presented by

Peter Thomas Williams

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

May 2012

Stony Brook University

The Graduate School

Peter Thomas Williams

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation.

Radu Sion – Dissertation Advisor
Associate Professor, Computer Science

Erez Zadok – Chairperson of Defense
Associate Professor, Computer Science

Rob Johnson
Assistant Professor, Computer Science

Adrian Perrig
Professor, Carnegie Mellon University

Moti Yung
Research Scientist, Google Research
Adjunct Senior Research Faculty, Columbia University

This dissertation is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Dissertation

Oblivious Remote Data Access Made Practical

by

Peter Thomas Williams

Doctor of Philosophy

in

Computer Science

Stony Brook University

2012

Access pattern leaks threaten data confidentiality. The ability to access remote information without revealing the objects of interest is thus essential to remote storage privacy. Despite many challenges to deployment, this thesis asserts that there exist practical (applicable and economical) access privacy mechanisms.

Outsourced computing is a popular trend with good reason: significant cost savings can be obtained by consolidating data center management. This trend arrives with a new set of security issues, however. Companies expose themselves to significant risk by placing sensitive data in systems outside their control. Of concern are not only network security, data confidentiality, and collocation issues, but more importantly a significant shift in liability, and a new class of insider attacks.

To defend these new vulnerability surfaces, of special importance becomes the ability to provide clients with practical guarantees of confidentiality and privacy.

This thesis outlines a set of essential outsourcing challenges: (i) How can remotely-hosted data be accessed efficiently with privacy? (ii) How can multiple clients run transactions privately in parallel, with serializability assurances guaranteed by untrusted, possibly malicious transaction managers? (iii) How can new, efficient, minimal-TCB hardware be designed to better provide security and privacy outsourcing guarantees?

To answer these questions, this dissertation introduces new mechanisms for practical private data access and oblivious transaction processing, as well as new trusted hardware designs. A space-time trade-off of client storage vs. efficiency is explored, then expanded to the additional dimensions of multiplicity of clients, the nature of the trusted computing base (hardware vs. software), and the degree of client data processing (access vs. transactions vs. computation). The results are orders of magnitude more efficient than existing work. Together, they bridge the gap between theoretical possibility and practical feasibility.

Table of Contents

Table of Figures

Acknowledgements

I would first of all like to thank my parents Roger and Charlotte, my sisters, Ruth and Elizabeth, and my aunt Leslie. I thank also my friends throughout the New York area and beyond, including Aaron Jaffe, M. Caitlin Fisher-Reid, and the Fisher-Reid family, for their patience, kindness, and support.

I express deep gratitude to my advisor Radu Sion and to my committee members, and to the co-authors I worked with over the past few years. I thank the current and former members of the Network Security and Applied Cryptography lab at Stony Brook, who helped me prepare talks and edit papers.

I would also like to thank my managers and co-workers at IBM TJ Watson, including Rick Boivie, David Safford, Mimi Zohar, J.R. Rao, and Chai Wah Wu. My undergraduate mentors, including Victor Ambros, and at Brandeis University, Tim Hickey, and Jacques Cohen, also offered me a tremendous amount of help.

Finally, I would like to thank the faculty and staff of the Stony Brook Computer Science department, and in particular the administrative staff, who helped me patiently with numerous difficult requests.

I thank also the Graduate School at Stony Brook University, and the other friends and colleagues I failed to mention here.

My work was supported by NSF grants 0937833, 0845192, 0803197, 0716608, 0708025, and 0627554.

Part I

Evolution of Oblivious RAM

Chapter 1

Introduction and Models

1.1 Introduction

In an increasingly networked world, computing and storage services require security assurances against malicious attacks or faulty behavior. As networked storage architectures become prevalent—e.g., networked file systems and online relational databases in sensitive public and commercial infrastructures such as email and storage portals, libraries, and health and financial networks—protecting the confidentiality and integrity of *stored* data is paramount to ensure safe computing. In networked storage, data is often geographically distributed, stored on potentially vulnerable remote servers or transferred across untrusted networks; this adds security vulnerabilities compared to direct-access storage.

Moreover, today, sensitive data is being stored on remote servers maintained by third-party storage vendors for economic reasons. Most third-party storage vendors do not provide strong assurances of data confidentiality and integrity. For example, personal emails and confidential files are being stored on third-party servers such as FilesAnywhere.com, Gmail, Yahoo! Mail, and MSN Hotmail [57]. Privacy guarantees of such services are at best declarative and often subject customers to unreasonable fine-print clauses—e.g., allowing the server operator (and thus malicious attackers gaining access to its systems) to use customer behavior for commercial profiling, or governmental surveillance purposes [113].

To protect data stored in such an untrusted server model, security systems should offer users assurances of data confidentiality and access pattern privacy. However, a large class of existing solutions delegate this by assuming the existence of co-operating, non-malicious servers. As a first line of defense, for ensuring confidentiality, all data can be encrypted at the client side using non-malleable encryption before being stored on the server. The data remains encrypted throughout its lifetime on the server and is decrypted by the client upon retrieval.

Encryption provides important privacy guarantees at low cost. However, it is only a first step, as significant information is still leaked through the access pattern of encrypted data. As a motivating example, consider a database management system running on top of an untrusted, encrypted file system. The file system, in satisfying requests for the transaction

processor, learns semantic information about the transactions through the sequence of disk blocks accessed. If, for example, an alphabetical, encrypted keyword index is updated as an encrypted record is inserted, it can learn what keywords are present in the new record, *based only on the locations updated within the encrypted index*.

In existing work, one proposed approach for ensuring client access pattern privacy (and confidentiality) tackles the case of a single-owner model. Specifically, a *service provider* hosts information for a *client*, yet does not find out which items are accessed or what is stored. In this setup the client has full control and ownership over the data: other parties are able to access the same data through this client only. One prominent instance of such mechanisms is Oblivious RAM (ORAM) [41]. I will use the term ORAM to refer to any such outsourced data technique.

One of the main drawbacks of existing ORAM techniques is their overall time complexity. Specifically, in real-world setups, Goldreich and Ostrovsky's ORAM [41] yields execution times of hundreds to thousands of seconds per single data access. Recent work, including this work, provide various mechanisms to improve the efficiency by orders of magnitude.

This dissertation is structured in two parts. Part (I) considers Oblivious RAM. I consider first (Chapter 2) a brief overview of existing work in this area: a description of the original ORAM as well as some very recent approaches from other authors to improve performance. Chapter 3 examines the probabilistic oblivious sorting primitives I presented in 2008 [127,128] (and formalized in a journal version [130]). These are used in Chapter 4 to drastically improve performance of the original ORAM when assuming a limited amount of client storage. I apply similar techniques in Chapter 5 to speed things up by another order of magnitude, while redoing the entire ORAM data structure for reduced complexity. Chapter 6 tackles two more barriers to practicality: worst case cost and query latency. I demonstrate results with a full implementation, including a construction running parallel queries among multiple clients, and achieve usability by providing an implementation of the first oblivious file system. Chapter 7 explores a different approach to defeating query latency, providing the first poly-logarithmic single-round-trip ORAM without client storage requirements.

Part (II) considers two related access privacy scenarios. After beginning with a slightly broader survey (Chapter 8), I consider the scenario of transaction privacy (Chapter 9). Using a construction first presented in 2009 [129], I show how an untrusted (potentially malicious) service provider can provide transaction management, durability, and serialization, while allowing clients to retain full guarantees of privacy and correctness. Chapter 11 looks at the design of secure hardware, with applications to the access privacy scenario of Private Information Retrieval; I first presented this in 2011 [126].

1.2 Model

Deployment. Let us consider the following concise yet representative interaction model. A capacity-constrained *client* desires to outsource storage to an untrusted party (the *server*).

Sensitive data is placed by a client on a data server. Later, the client will access the outsourced data through an online query interface exposed by the server. Network layer confidentiality is assured by mechanisms such as SSL/IPSec. Without sacrificing generality, let us assume that the data is composed of equal-sized blocks (e.g., disk blocks, or database rows).

Clients need to read and write these stored data *block*s with correctness assurances, while revealing no information to the (curious and possibly malicious) server. I describe the protocols from the perspective of the client, who will implement two privacy-enhanced primitives: read(*id*), and write(*id*, *newvalue*). The (untrusted) server need not be aware of the protocol, but rather just provide traditional store/retrieve primitives.

Adversary. The adversarial setting considered in Chapters 5, 7, 9, and 11 assumes a storage provider that is *curious and possibly malicious*. Not only does it desire to illicitly gain information about the stored data, but it could also attempt to cause data alterations while remaining undetected. I prove that clients will detect any tampering performed by the server, before the tampering can affect the client's behavior or cause any data leaks. I will not consider timing attacks, noting that an implementation can be turned into a timing-attack free implementation without affecting the running time complexity of these protocols. I also do not address direct denial of service behavior.

The adversary considered in Chapters 4 and 6 is considered to be *curious but honest*.

Cryptography. For the majority of the constructions presented in this dissertation, I will require only three cryptographic primitives with all the associated semantic security [40] properties. First, a collision-free hash function which builds a distribution from its input indistinguishable from a uniform random distribution. Second, an encryption function that generates unique ciphertexts over multiple encryptions of the same item, such that an adversary has no non-negligible advantage at determining whether a pair of encrypted items of the same length represent the same or unique items (e.g., satisfying semantic security). Third, I will need a pseudo-random number generator whose output is indistinguishable from a uniform random distribution over the output space. Any one-way hash function satisfying the requirement above can be used for this purpose, provided that a key generator has provided a seed. Finally, some of the constructions require basic public-key cryptography signing and encryption primitives.

In these definitions, I assume the adversary is polynomially-bounded in the key size. Indistinguishability is used in the standard computational privacy sense, requiring that any polynomially-bounded adversary have only a negligible chance of violating security, taken over possible key choices. Negligibility is again used in the standard cryptographic sense: a function is **negligible** in its parameter if it is asymptotically bounded above by all inverse polynomials. Similarly, an event occurs **with high probability** if the probability of an event occurring (taken across the appropriate domain) is one minus a negligible quantity.

A trend throughout this discourse is the application of only these "basic" cryptographic primitives (with a few exceptions, including Section 5.2). This constraint follows from two motivating factors. First, establishing performance and practicality as the primary goal is often (but not necessarily) incompatible with the use of more complicated primitives.

Second, this allows us to stick to the most mainstream and widely used primitives, avoiding dependence on less-broadly-tested cryptographic constructions. This has the side effect of making the constructions algorithm-focused, rather than cryptography-focused.

1.3 Common Notation

Oblivious RAM (ORAM) refers to any construction that provides the previously specified read and write operations with privacy. n denotes the database size (in fixed-size blocks). log represents the natural log. Other notation is defined where introduced.

1.4 Performance Analysis

There are several different trends concerning representation of performance analysis in related work. Some papers look at message complexity, measuring performance in terms of the number of passed messages, according to a message size defined in the paper. Often, it is convenient to use a simplified model that measures cost in terms of a multiplicative overhead factor. This considers how many accesses sized b are required to simulate a single *oblivious* access sized b. While it provides an intuitive and useful idea of the cost overhead, it is not suitable for all constructions, as many require expenses that are not of the form of extra accesses.

To be self-consistent in this dissertation, I represent asymptotic computation complexity, storage complexity, and communication complexity in the standard asymptotic model. However, I specify storage and communication costs in terms of words, instead of bits. That is, I assume that a block identifier can be transmitted, processed, and stored at $O(1)$ cost.

Recognizing that this is useful for identifying asymptotic performance trends only, implementations are benchmarked in representative environments wherever possible. I will also find it convenient at times to model instead the *expected* performance based on the amount of computation, I/O transfer, and network transfer required in an implementation.

1.5 Security Definitions

After examining what is meant by "query privacy", this section will establish a formal definition of query privacy to capture this notion. Related ORAM papers (including work by Pinkas and Reinman [108]) typically address query privacy by showing that any sequence of block accesses used to satisfy a query appear random to the server. As shown by Kushilevitz et al. [72], this approach is incomplete, often leaving subtle privacy leaks in existing ORAMs.

This trend was partially reversed by Boneh et al. [17], which introduces a security game in a slightly broader model. That is, to capture information leaked through the timing and total number of queries, as well as through query transcripts, they model the execution of programs, rather than the execution of sequences of queries. Modeling clients as Turing machines thusly is general enough to consider almost any scenario. As this is more general

than needed for our purposes, I will provide a slightly more restricted definition, modeling clients as equal-length adaptively chosen sequences of queries, rather than adaptively chosen Turing machines.

The privacy goal of Oblivious RAM is to hide the client's access pattern from the server. Intuitively, the server should learn nothing about what the client is doing. The server should be as powerful as possible; borrowing the intuition behind IND-CPA, the server should be able to submit queries to the client, watch them run, and learn the result. It might be desirable to have an Oblivious RAM that lets the server run queries without learning anything about the client's access pattern. Unfortunately this guarantee is not satisfiable in the Oblivious RAM model: any party able to run arbitrary requests learns trivially when writes are performed by the client, by watching for changes.

This observation indicates that the advantage of the server could instead be defined as its ability to learn anything about the access pattern from the execution transcripts. That is, rather than measuring the server's advantage at guessing anything about the client's access pattern, we should measure the *additional* advantage that the execution transcripts give at guessing anything about the client's access pattern.

Note that the security definition presented by Boneh et al. [17] is adaptive only during the preparation phase. That is, while the adversary can pick queries based on previous transcripts during the preparation phase, once the verifier has picked a random bit, the adversary no longer gets to see transcripts between queries. I remove this limitation below.

This generalization reflects the observation that in practice, an adversary may have some impact over what queries are run, while it is observing transcripts. Giving the adversary the ability to adaptively choose queries during the challenge phase (not just in the preparation phase) encompasses an adversary's ability to learn from the transcripts, even as it has some influence over the queries. I then show in Theorem 1 that such a generalization is plausibly necessary: that is, there exist constructions secure against a limited-adaptive adversary but granting an arbitrarily higher advantage to a fully adaptive adversary.

1.5.1 Security Model (IND-CQA)

Definition 1. *An ORAM Instance provides* adaptive query privacy*, if for each possible sequence of accesses s of length m, the corresponding sequence of transcripts gives no information about s.*

Adaptive query privacy is captured in the IND-CQA ("indistinguishable under a chosen query attack") security game (sketched below) that represents an honest-but-curious, computationally bounded adversary. Borrowing the intuition behind IND-CPA, the server is allowed to submit an arbitrary number of queries to the client, watch them run, and learn their result:

1. The adversary provides the initial database contents.

2. The verifier flips a secret random bit b.

3. The verifier then engages in the following process, repeated polynomially many times t with a polynomially bound adversary:

   (a) The adversary specifies two queries q and r, where a query can be a read, write, or insert, for any block, and with any fixed-size block content.

   (b) If $b = 0$, the verifier executes query q; otherwise it executes r.

   (c) The verifier returns the execution transcript (but not the query result) to the adversary.

4. After t rounds, the adversary guesses b.

An ORAM construction is considered to have **computational adaptive query privacy** if there is no adversary A with a non-negligible advantage at guessing b, with the probability taken over the possible choices of b, secret key, and internal coin flips by the adversary.

Intuitively, any information about the sequence of accesses leaked through the transcript allows construction of adversary that gains an advantage at distinguishing transcripts.

Note that the inability for the server to observe the query results does not imply the server's inability to predict its query results. In fact, since the server constructs the database, it knows in advance the answer to each query. It simply does not learn, during execution, which of each pair of queries is run. This adversary still corresponds to one able to observe *externally* a client's queries and query results—in fact, the adversary has a strictly stronger ability, since it can *choose* those queries and results. The idea captured is that the *execution transcripts* give the adversary no additional advantage.

The model defined by Boneh et al. [17] allows adaptive queries only in the preparation phase. Replacing the programs with sequences (to avoid considering program length and timing attacks), I refer to this as the limited-adaptive-adversary model. As discussed above, the fully adaptive adversary of IND-CQA captures a more powerful adversary than the limited-adaptive adversary:

Theorem 1. *If there exists a construction γ-secure in the limited-adaptive adversary model (i.e., granting any polynomial adversary no more than γ advantage), then for any $0 < \epsilon \leq 1$, there exists a construction $\gamma + \epsilon$-secure in the limited-adaptive-adversary model, but for which there exists an adversary with advantage $\frac{1}{2}(1-(1-\epsilon)^{f(c)})$ under IND-CQA, for any function $f(c)$ polynomial in the security parameter c.*

Proof. Let S be a construction γ-secure under the limited-adaptive model.

Modify S to make it insecure in the following manner (and call the result S'). It behaves equivalently to S, except it includes a new, random bit string length c in each transcript output. Whenever this bit string is a prefix to the following query, then with probability ϵ append 'heads' as an additional field in that transcript output. In all other cases, append 'tails' to the query transcript.

S' is $\gamma + \epsilon$ secure in the limited-adaptive model: the only additional information available to an adversary is an ϵ chance on each query in revealing a certain property. The adversary

can only guess about one of these queries, since the challenge phase does not provide transcript feedback—so the adversary can try only use this information to guess about a single query.

Now, construct an adversary that issues $f(c)$ queries, building one sequence (b=1) that always prefixes the random bit string leaked by the construction to the next query. The other sequence (b=0) always makes a random query that does not prefix this bit string. At the end, output 1 if the field 'heads' is appended to any of the transcripts.

This event occurs with probability $1 - (1 - \epsilon)^{f(c)}$; in these cases the adversary knows with certainty that b=1. Otherwise, the adversary guesses 0. The guess is thus correct with probability $\frac{1}{2} + \frac{1}{2}(1 - (1 - \epsilon)^{f(c)})$, leading to an advantage of $\frac{1}{2}(1 - (1 - \epsilon)^{f(c)})$. □

In the example construction above, the number of queries $f(c)$ necessary to exhibit an advantage of any constant amount is proportional to $1/\epsilon$. That means that a negligible advantage requires a super-polynomial $f(c)$ to grow into a non-negligible advantage. It remains an open question whether there exists a construction which is ϵ-secure in the limited-adaptive model, for a negligible ϵ, but which grants an adversary a non-negligible advantage in the IND-CQA model.

And now I show an IND-CQA adversary is at least as powerful as the limited-adaptive adversary.

Theorem 2. *Any construction secure under IND-CQA is also secure under the limited-adaptive-adversary model.*

Proof. (outline) Suppose there exists an adversary A that breaks a construction C under the limited-adaptive-adversary model with advantage ϵ. This leads to construction of an adversary B that breaks that construction under IND-CQA with the same advantage.

Adversary B is constructed equivalently to adversary A. The additional information available to adversary B—the transcripts of queries during the challenge phase–is discarded. The advantage of B in IND-CQA is equal to ϵ. □

1.5.2 An Actively Malicious Adversary

This game can be extended to capture query privacy versus an actively malicious adversary. While the game defined by Boneh et al. [17] considers only an honest but curious adversary, the idea here is to allow the adversary to specify the return value of every request of the server. That is, instead of being provided the transcript by the verifier after each query, the adversary can choose the result and the server-supplied portion of the transcript of each request.

I note that many existing constructions, even those that assume an actively malicious adversary, are vulnerable in this model. The problem is that these protocols simply abort upon detecting incorrect behavior. While the client can detect when the server is behaving incorrectly (the server cannot get away with the malicious behavior), one of three scenarios occurs.

1. The client might leak access pattern information before detecting the incorrect behavior.

2. The point at which the failure is detected leaks access pattern information to the server.

3. Or, ideally, the client does not reveal anything to the server by acknowledging when the server deviates from protocol.

The second scenario is relevant and often leads to a significant leak. For example, the server can easily corrupt specific items, to trigger a failure on a repeated query. Moreover, it is not always straightforward to prevent this leak, since it is not clear how to proceed once the server has violated the protocol.

Fortunately, such a leak requires the adversary to reveal its adversarial nature to the client. This makes it feasible, for example, to employ economic incentives to mitigate this leak. One option is thus to restrict the adversary model, allowing it to violate the protocol only where it has a non-negligible chance of going undetected. Nevertheless, Section 5.2.1 will discuss how to hide acknowledgement of a protocol violation, in order to prevent loss of privacy.

1.5.3 Security Model (IND-DA)

Definition 2. *An ORAM Instance provides* adaptive query privacy against a deviant adversary*, if for each possible sequence of accesses s of length m, the corresponding sequence of transcripts gives no information about s.*

Adaptive query privacy against a potentially malicious adversary is captured in the IND-CQA-D ("indistinguishable against a deviant adversary") security game (sketched below) that represents an honest-but-curious, computationally bounded adversary.

1. The adversary provides the initial database contents.

2. The verifier flips a secret random bit b.

3. The verifier then engages in the following process, repeated polynomially many times t with a polynomially bound adversary:

   (a) The adversary specifies two queries q and r, where a query can be a read, write, or insert, for any block, and with any fixed-size block content. The adversary also specifies a Turing machine T that will respond to protocol requests from the verifier.

   (b) If $b = 0$, the verifier interactively executes with T the query q; otherwise it executes r.

   (c) The verifier returns the execution transcript (but not the query result) to the adversary.

4. After t rounds, the adversary guesses b.

Chapter 2

A Brief Survey of Oblivious RAMs

This chapter provides a short survey of related Oblivious RAM techniques, with a focus specifically on computational ORAM techniques in the random oracle model. A broader discussion, including theoretical work on ORAMs in the standard model [27], and lower bounds on ORAMs in the standard model [7], is out of scope.

2.1 Goldreich's Oblivious RAM

Oblivious RAM [41] provides access pattern privacy to *a single client* (or software process) accessing a remote database (or RAM), requiring only logarithmic storage at the client. The amortized communication and computational complexities are $O(log^3 n)$.

In ORAM, the database is considered to be a set of n semantically-secure encrypted blocks (with an *ORAM key* held by the client) and supported operations are read(id), and write(id, *newvalue*). The data is organized into $\log_4(n)$ *levels*, as a pyramid. Level i consists of up to 4^i blocks; each block is assigned to one of the 4^i buckets at this level as determined by a hash function. Due to hash collisions each bucket may contain from 0 to $O(\log n)$ blocks.[1]

Reading. To obtain the value of block id, the client must perform a read query in a manner that maintains two invariants: (i) it must never reveal which level the desired block is at, and (ii) it must never retrieve a given block from the same spot twice. To maintain (i), the client always scans a single bucket in every level, starting at the top (Level 0, 1 bucket) and working down. The hash function informs the client of the candidate bucket at each level, which the client then scans. *Once the client has found the desired block, the client still proceeds to each lower level, scanning random buckets instead of those indicated by their hash function.* For (ii), once all levels have been queried, the client then re-encrypts the query result (a block) with a different nonce and places it in the *top* level. This ensures that when the client repeats a search for this block, it will locate the block on its first access (in a different location), and the rest of the search pattern will be randomized. Figure 2.1

[1] A security parameter sets the bucket size to ensure there is only a negligible chance of bucket overflow.

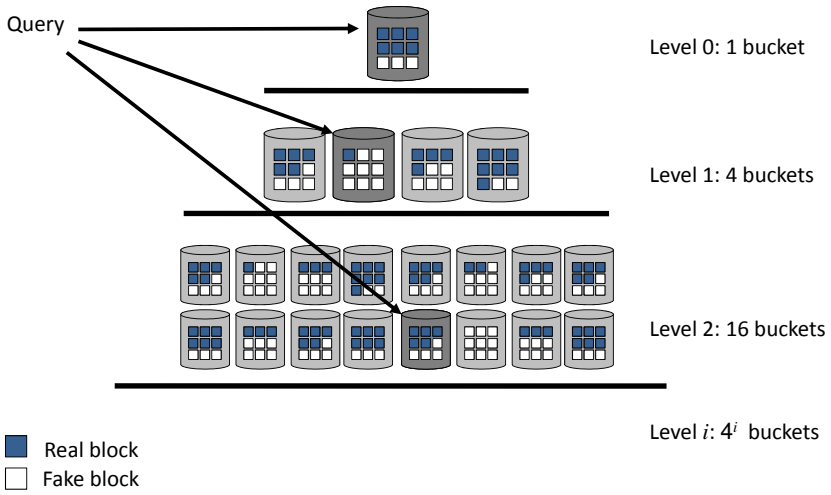

Figure 2.1: ORAM: Query Overview. Clients search from the top down, choosing the buckets indicated by a hash function. Since previously accessed items are located in the top level, and buckets are selected randomly on lower levels once an item is found, all accesses appear uniformly random to a polynomial-time observer.

illustrates this process. Note that the top level will immediately fill up; the process to dump the top level into the one below is described later.

Writing. Writes are performed identically to reads in terms of the data traversal pattern, with the exception that the new value is inserted into the top level at the end. Inserts are performed identically to writes, since no old value will be discovered in the query phase. Note that semantic security properties of the re-encryption function ensure the server is unable to distinguish between reads, writes, and inserts, since the access patterns are indistinguishable between each case.

Note that in both *read* and *write* procedures, no block is ever deleted, meaning that the old block values are kept in the database. Though this behavior can be modified, as described in Section 4.2.

Level Overflowing. Once level i is full (each 4^i steps), it is emptied into the level below, as illustrated in Figure 2.2. This lower level is completely re-encrypted, and re-ordered according to a new hash function. Thus, accesses to this new iteration of the lower level will henceforth be completely independent of any previous accesses. Note that each level will overflow once the level above it has been emptied 4 times. Any re-ordering must be performed obliviously: once complete, the adversary must be unable to make any correlation between the old block locations and the new locations. A sorting network is used to reorder the blocks.

To enforce invariant (i), note also that all buckets must contain the same number of blocks. For example, if the bucket scanned at a particular level has no blocks in it, then the

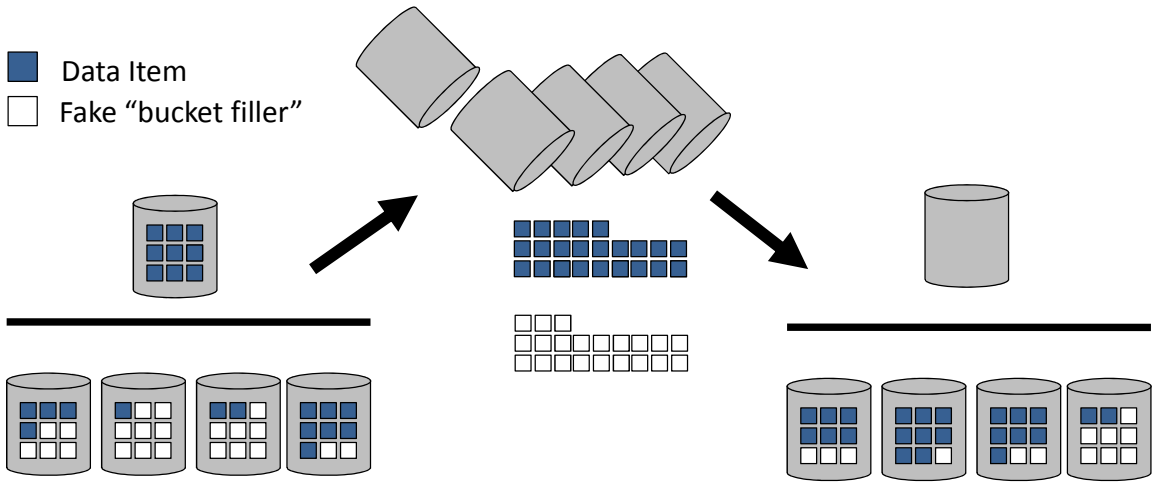

Figure 2.2: ORAM: Reshuffling a level into the level below. Once a level is full, the client places all items from that level into the level below, and shuffles obliviously in such a manner that the server can make no correlation between the old locations and the new locations.

adversary would be able to determine that the desired block was *not* at that level. Therefore, each reorder process fills all partially empty buckets up to the top with *fake* blocks. Recall that since every block is encrypted with semantic security, the adversary cannot distinguish between fake and real blocks. The client decrypts the block, then checks the value of a flag included with the block cleartext to determine whether it is fake. For discussion of how to reconcile old versions of the blocks with the new versions, and additional discussion on other considerations, please refer to Section 4.2.

Cost. Each query requires a total online cost of $O(\log^2(n))$ for scanning the $\lambda \log n$-sized bucket on each of the $\log_4 n$ levels. λ is a security parameter relating to the probability of buckets overflowing. There is also an additional, amortized cost due to intermittent level overflows. Using a logarithmic amount of client storage, reshuffling levels in ORAM requires an amortized cost of $O(\log^3 n)$ per query.

2.2 Recent Developments

An ORAM mechanism, presented in the context of Private Information Retrieval (see Chapter 8.1.1) is introduced by Wang et al. [125]. It has an amortized cost of $O(n/k)$ where n is the database size and k is the amount of secure storage (in blocks). The protocol is based on scrambling of a minimal set of server-hosted items. A partial reshuffle costing $O(n)$ is performed every time the secure storage fills up, which occurs once every k queries. While an improvement for small databases, this result is not always practical since the total database size n often remains much larger than the secure hardware size k. Boneh et al. [17] examine a similar construction, formalizing the security model.

Starting with my work [127], presented in Chapter 4, researchers have sought to improve the overhead from the poly-logarithmic performance of the original ORAM. I then present in Chapter 5 work I first introduced in 2008 [128], a faster ORAM variant which also features correctness guarantees, with computational complexity costs of only $O(\log n \log^2 \log n)$ (amortized per-query), under the assumption of $O(\sqrt{n \log n})$ client memory, and requiring $O(n)$ server storage. The assumed client storage is used to speed up the reshuffle process by taking advantage of the predictable nature of a merge sort on uniform random data.

2.2.1 Cuckoo Hash ORAMs

A new approach to speed up ORAM was revealed by Pinkas et al. [108], showing the applicability of the cuckoo hash construction [102]. This has the same motivation as our revamped ORAM structure (Chapter 5): to speed up querying and shuffling by finding a construction more efficient than bucket-scanning. Instead of using Bloom filters to separate level membership testing from the stored data, they represent the levels as a cuckoo hash. In a cuckoo hash table, each item can exist in one of two places. If both places are occupied, then one of the items is kicked out into its alternate location (possibly pushing over a whole chain of items); this is usually fairly easy since there are twice as many slots as items. The essential challenge solved in this paper is obliviously, and efficiently, finding a mapping of items to slots. The rest of the slots are filled with fake items. The idea is that when looking up an item in the hash table, it suffices to check two locations, and the server does not learn whether the item was found. Unfortunately, this construction was shown [44, 72] to leak access privacy information. The problem is that not all hash functions result in a successful mapping of items to slots. For example, three items might all be mapped to the same two slots. Limiting the selection to those hash functions that *do* have a successful mapping (e.g., by retrying until it finds one that succeeds) reveals something significant about the *hash function* chosen.

A similar, but secure, approach, allowing efficient item lookup while hiding success was developed by Goodrich et al. [44]. The idea is to use a small "stash" to absorb the conflicting items when the selected hash function does not result in a successful mapping. The result is a $O(\log^2 n)$ solution requiring *logarithmic* client storage and requiring only $O(n)$ server storage. This approach has been re-applied by Goodrich et al. [45] (where it is turned into a de-amortized solution) and by Kushilevitz et al. [72] (who include a good review recent developments).

2.2.2 De-amortized ORAMs

Researchers have long recognized the utility of de-amortized constructions (i.e., in which the worst case cost matches the average cost). Recently, Kushilevitz et al. [72], Goodrich et al. [45], and Boneh et al. [17] all provide de-amortized constructions. The first de-amortized construction, however, follows soon after the original ORAM, going all the way back to 1997 [101]. An elegant construction is provided by Shi et al. [114], which structures the

ORAM as a tree and specifies the set of shuffle operations to perform on each query—resulting in a verifiably de-amortized solution, whereas the other de-amortized solutions typically specify only the asymptotic size of the amount of shuffle work to perform on each query. Recognizing the importance of providing efficient worst-case costs, I show in Chapter 6 how to de-amortize this construction. De-amortized constructions do not always lead readily to an actual implementation. Instructions such as "perform a chunk of work sized thusly," or "run the shuffle in the background while querying," are fine for establishing existence proofs, but can be devastatingly inappropriate in achieving an actual prototype. Nevertheless, I provide my own implementation in Chapter 6, and demonstrate it to be efficient.

2.2.3 Recursive ORAM

A promising recursive construction is introduced by Stefanov et al. [120] under the assumption of $O(n)$ words / $O(n \log n)$ bits of *client* storage. Using this storage, it promises to reduce the level construction cost. Meanwhile, it requires only a constant number of online round trips (which is an important trait when employing over high latency links, as we will see in Chapters 6 and 7). The main idea of this construction is to split the remote database across $\sqrt{n}$ smaller mini-ORAMs, each sized $O(\sqrt{n})$. These can be built from an existing construction or using the one they provide. The local storage is used for three purposes: one, to keep track of which items live in which mini-ORAMs (which is randomized and constantly changing). Two, it is used to speed up the shuffles within the mini-ORAMs. Since each are small enough to fit into client memory in entirety, any shuffle within a mini-ORAM can be performed in a linear number of accesses, requiring just a scan, some client processing, and a write back. Third, the storage is used to keep a cache of recently accessed items (similar to a construction by Wang et al. [125]), to locally shuffle items before writing them back to a new mini-ORAM.

The challenge in this construction is hiding which items come from which mini-ORAMs—and any other information that might be revealed by the choice of mini-ORAM to read from or write to. The authors mitigate this leak by ensuring that an item gets written back to a *random* mini-ORAM after it has been read. The idea is to cache items over a period long enough to ensure that each of the $\sqrt{n}$ mini-ORAMs has a roughly equal number of items waiting to be written back, then to insert cached items to their randomly selected target mini-ORAM, inserting fake items as necessary to equalize the number of writes to each mini-ORAM.

The clear drawback there is the assumed $O(n)$ client storage. The authors present the $O(n)$ client storage assumption as not necessarily unreasonable, since a large block size means that the client only needs a constant fraction of the outsourced storage. Their alternate construction recursively uses another, smaller[2] instance of this ORAM to store the position map (and to provide the working memory for shuffles). This loses some of their performance edge and requires multiple round trips per query.

[2]provided that the block size is larger than $\log n$

2.2.4 Randomized Shell Sort

A randomized shell sorting network identified by Goodrich [46] was subsequently employed to construct Oblivious RAMs [44, 108]. Goodrich et al. [44] claim logarithmic overhead, in a model calculating message complexity and based on some assumptions of block sizes, rather than the traditional computational or communication complexity models. Since it uses the randomized shell sort to perform oblivious sorts of the data, it has a lower bound on par with existing work.

This sort procedure was used in the first design of PD-ORAM (Chapter 6), but was found to unfortunately result in simply too many disk seeks to make it usable on a even a medium-sized database. The randomized nature of the shell short guarantees that the order of item access appears non sequential and random, making efficient use of rotational hard disks difficult. The amortized cost of constructing the bottom level in a terabyte database with an acceptable failure rate is in the range of hundreds of disk seeks per query, already putting the implementation outside of the targeted performance goals. See the Appendix for this analysis.

It has been observed that this sort is significantly less efficient than non-oblivious sorts. In particular, Pinkas and Reinman [108] opt to use a standard sort on the client for those levels that fit in memory. Where client memory assumptions make it possible, I choose to use my own asymptotically equivalent merge sort, described in Chapter 3. This sort runs faster (requiring $\log_2 n$ sequential passes instead of $6c \log_2 n$ random passes), but imposes a client storage requirement of $c\sqrt{n \log n}$, for a security parameter c.

Chapter 3

Oblivious Sort with Local Storage

This chapter examines a key building block of most Oblivious RAMs: the mechanism to perform an oblivious sort. An adversary (the server) has stored a list of items encrypted by the client. The client's job is to rearrange this remotely stored list without revealing anything to the about the permutation. The client has some amount of working memory available to perform this sort; the requirements vary under different constructions.

I establish two new constructions providing this building block: the Merge Sort and the Merge Scramble, which are necessary for the next chapters. I also review constructions in related work, which will be useful in scenarios with only logarithmic client memory.

3.1 Oblivious Sorting in Related Work

3.1.1 Sorting Network

Starting at the extreme end of the client-memory spectrum, let us consider clients with only logarithmic memory. As identified by Goldreich and Ostrovsky [41], sorting networks are naturally suited to this scenario. They go on to suggest employment of the $O(s \log^2 s)$-time Batcher odd-even sorting network [6], where s is the input size. At the time of that publication, this was the fastest practical sorting network. An asymptotically superior $O(s \log s)$-time AKS sort [2] is significantly slower in practice for all currently feasible database sizes due to a large constant.

3.1.2 Randomized Shell Sort

In 2010, Goodrich et al. [46] presented a randomized shell sorting network that requires $O(s \log s)$ comparisons and succeeds with high probability (e.g., c repetitions can be combined to guarantee success with high probability in c). In comparison, traditional sorting networks always succeed. Nevertheless, this result is directly relevant to all logarithmic-client-memory ORAM scenarios. There are still significant overheads involved, however, making other oblivious sorts more efficient when there is more client memory. This is both

because the sort requires a random, non-sequential access pattern, and because the repeated application requires a significant number of passes over the data.

This section analyzes the *practical* performance of the randomized shell sort in ORAM settings on large databases. Consider the construction of the largest level of a 1 TB database. This level contains at least 0.5 TB of data. For 10 KB blocks, this translates into 50 million blocks. The randomized shell sort makes $6c \log_2 n$ random passes across the database, incurring a total of $6cn \log_2 n$ seeks every n queries, where c influences the sort failure probability. For $n = 5 \times 10^7$, $\log_2 n = 23$, translating to $138 \times c$ disk seeks per query for the largest level alone.

For $c = 4$ as suggested in the original paper, this amortizes to 550+ disk seeks per query. Even for high-speed, low-latency disks with 6ms seek times, this becomes at least 3.3 seconds/query (in addition to any/all other significant network and CPU overheads)!

Unfortunately, ORAM imposes a unique sorting requirement that is difficult to satisfy using the randomized shell sort. This requirement derives from the requirement that, to maintain privacy, the sort must succeed with overwhelming probability. Moreover, all sorts must be indistinguishable, eliminating the possibility of retrying in the case of failure. This is because observation of a sort failure translates into an advantage at distinguishing the permutation from random, which translates into a privacy leak.

While in other applications it may suffice to repeat the sort until it succeeds, when applied to ORAM, the sort parameter c must be chosen to guarantee success with overwhelming probability. In Goodrich's presentation of the sort [46], the failure rate is determined experimentally; it is not obvious from the proofs what c is necessary to obtain a given acceptable error rate on database sized n. While the paper does prove that the probability of failure is negligible in c, which is sufficient for a construction existence proof, it is unclear what parameters can be safely chosen for a practical implementation.

3.1.3 External Memory Sort

An external memory oblivious sort is provided by Goodrich et al. [47]. They use a modified sorting network to make efficient use of local memory; the idea is to apply the k-way merge sorting network introduced by Lee and Batcher [73]. This provides a promising result, allowing a sort of s items, requiring $O(s \log_k s)$ network transfers, for local memory k. For any constant d, and $k = s^{1/d}$, this provides a sort requiring an amount of network transfer only linear in s. However, the constant factors are not considered.

3.2 Merge Sort

I now present an algorithm that performs a merge sort on an array of size s, with $2c\sqrt{s}$ local memory, in $O(s \log_2 s)$ time, without revealing any correlation between the old and new permutations. The algorithm runs recursively on the remote array as described in Procedure ObliviousMergeSort. The recursion depth is $\log_2 s$, and each level of recursion entails a single pass of size s across the entire array.

The idea is to perform a merge sort with a fixed access pattern. A traditional merge sort splits an array to be sorted into two, recursively sorts the subarrays, then merges the two subarrays together. This last "merge" step performs the work: combining two sorted subarrays together. Traditionally, the merge maintains two pointers: one at the head of each array. It incrementally compares the values at the pointers, outputs the smaller, and advances that one pointer. The insight here is that since we will use random sort keys, these pointers will remain very close together. With high probability, in fact, they will deviate only by $O(\sqrt{s})$ (Theorem 4).

This means a fixed access pattern can merge any two arrays using a small local cache. First sort the two subarrays (line 1) Fill each $O(\sqrt{s})$-sized buffer halfway full, reading from the fronts of each array (line 2). Now, read two more items (one from each subarray, line 3) and output two (according to the sort keys, line 4).

The correctness of this algorithm depends on the uniformity of the starting permutation of the items being sorted, as illustrated in Theorem 4. Its oblivious nature derives immediately by construction:

Theorem 3. *The Oblivious Sort algorithm is private: no more than a negligible amount of information about the new permutation is leaked to a computationally bounded adversary.*

Proof. (outline): The ordering of reads and writes in every instantiation of the scramble is identical: observe that in the supplied pseudo-code for ObliviousMergeSort, the `readNextBlockFrom` and `writeNextBlockTo` functions are called in the same pattern every time, depending only on s, not the comparisons made on the output of `HashLocation`.

The semantic security properties of the symmetric encryption scheme guarantee that the adversary cannot correlate any two blocks based on the encrypted content (the server cannot determine whether t is from q_1 or q_2). Therefore, every instantiation of the scramble appears identical to the server: it sees a fixed pattern of reads interspersed with a fixed pattern of writes of unintelligible data. The specific fixed pattern is known beforehand to the server (from the algorithm definition), and the content of the reads has no correlation to the content of the writes since the blocks are re-encrypted with a semantically secure encryption scheme at the client.

Therefore, in observing any iteration (or sequence of iterations) of the oblivious merge sort, the (computationally bounded) adversary learns nothing. Moreover, the final permutation is chosen from among all possible permutations. Since the access pattern is identical when generating each of these permutations, the server has no ability to guess the resulting permutation.

A small number of permutations will cause the algorithm to fail and output $\perp$, if the queues overflow, but this absence of a failure reveals only a negligible amount of information about the new permutation, since failure occurs with negligible probability, as shown next. □

Theorem 4 proves that $2c\sqrt{n}$ memory is sufficient for the Oblivious Sort algorithm, with high probability in c. I prove this by showing that all but a negligible portion of the possible permutations encode walks that remain within $\pm c\sqrt{n}$ of the starting location. This is a

```
/* ObliviousMergeSort: Recursively sort the server-stored array A on
   HashLocation(a) in a manner such that the server learns nothing about the
   permutation. A may be too big to fit entirely in client memory.          */
if A is size 1 then
    return A
end
s ← size of A
A_1 ← first half of A
A_2 ← second half of A
1  A_1 ← ObliviousMergeSort(A_1)
A_2 ← ObliviousMergeSort(A_2)
B ← New remote buffer with the same size as A
q_1 ← empty queue stored locally, to fit up to c√s items
q_2 ← empty queue stored locally, to fit up to c√s items
/* Each queue may contain up to c√s items; however |q_1| + |q_2| ≤ c√s      */
2  for x = 1 to c√s/2 − 1 do
    q_1.enqueue(decrypt(readNextBlockFrom(A_1)))
    q_2.enqueue(decrypt(readNextBlockFrom(A_2)))
end
/* At this point, each queue will have c√s/2 − 1 blocks                     */
3  for x = c√s/2 to s/2 do
    q_1.enqueue(decrypt(readNextBlockFrom(A_1)))
    q_2.enqueue(decrypt(readNextBlockFrom(A_2)))
    /* Now we've read 2 blocks; time to output 2 blocks                     */
    for i = 1 to 2 do
        t_1 ← peek(q_1);
        t_2 ← peek(q_2);
4       if HashLocation(t_1) > HashLocation(t_2) then
            t ← q_1.dequeue()
        else
            t ← q_2.dequeue()
        end
        writeNextBlockTo(B, encryptWithNewNonce(t))
    end
end
for x = 1 to c√s do
    /* Now output the remaining c√s items distributed between the queues    */
    t_1 ← peek(q_1);
    t_2 ← peek(q_2);
    if t_1 ≠ ∅ and (t_2 = ∅ or HashLocation(t_1) > HashLocation(t_2)) then
        t ← q_1.dequeue()
    else
        t ← q_2.dequeue()
    end
    writeNextBlockTo(B, encryptWithNewNonce(t))
end
return B
```

Procedure ObliviousMergeSort(A)

tighter bound on the required memory bound than what is achieved in Phases 1 and 3; and I note that as an alternative proof, the random walk proof used in Phase 1 can be applied here to achieve a looser bound.

I show now that queues of size $c\sqrt{s}$ are sufficient (with high probability) for random sorting of an array of length s. That is, when selecting elements in a random order from two arrays, the array sizes will likely remain close over time. The queue size can be modeled intuitively as a one-dimensional random walk with s steps, with the additional property, since selection is performed without replacement, that the further the deviation from balance, the more likely each step is to return back towards balance.

The queue size at step j in any step of the merge sort algorithm is a probabilistic function $Q(j)$ defined iteratively as follows:

$$\begin{cases} Q(0) = c\sqrt{s/2} \\ Q(j) = Q(j-1) + \frac{1}{2} & \text{Pr.} \quad \frac{1}{2} - (Q(j-1) - c\sqrt{s/2})/(s-j) \\ Q(j) = Q(j-1) - \frac{1}{2} & \text{Pr.} \quad \frac{1}{2} + (Q(j-1) - c\sqrt{s/2})/(s-j) \end{cases}$$

On each step, we either dequeue an item from this queue or the other, symmetric queue (not modeled here). If we dequeue an item from this queue, the queue size will decrease by one; if not, it will remain the same. Additionally, once every two steps we enqueue one item to the queue; my model captures this by enqueueing half an item on every step. This results in stepping $+\frac{1}{2}$ or $-\frac{1}{2}$ (instead of $+0$ or -1) depending on whether we dequeue an item on this step.

I show that with high probability, the queue size at any step along the walk will be greater than zero, and less than $2c\sqrt{s/2}$. The step function is chosen to select with equivalent probability one remaining item from either array. Since the arrays are both size $s/2$, when the arrays finally empty, we will have selected an equal number of items from each. Observe that $Q(s) = c\sqrt{s/2}$: as j approaches s, the "pressure" to tend $Q(j)$ to $c\sqrt{s/2}$ increases.

This function is equivalent to pulling steps out of an urn without replacement, starting with $s/2$ step-ups and $s/2$ step-downs in the urn. The further we deviate from an equivalent number of each step, the more likely it is for the next step to bring the tally closer to equivalent counts.

Let us re-frame the probabilistic function $Q - c\sqrt{s/2}$ as a random walk of length s, starting and ending at 0, with $d = Q(j) - c\sqrt{s/2}$, according to the following step function, with step size $\frac{1}{2}$.

$$\begin{aligned} P(up) &= \frac{1}{2} - \frac{d}{s-j} = \frac{(s-j-2d)}{2(s-j)} \\ P(down) &= \frac{1}{2} + \frac{d}{s-j} = \frac{(s-j+2d)}{2(s-j)} \end{aligned}$$

Let us define a *biased random walk* to be a sequence of steps chosen according to the step function defined above. I now show that all such biased walks are equally likely. Intuitively, the bias in each step is exactly the amount necessary to choose equally from all the remaining walks that end at 0.

Lemma 1. *All biased walks are all equally likely*

Proof. At any position in the walk, 2 steps constituting an up followed by a down can be replaced by a 2 steps constituting a down followed by an up to obtain an equally likely walk. This property falls from the step function, (height d, step j, length s):

$$\begin{aligned} Pr[up@j,d]Pr[down@j+1,d+\frac{1}{2}] &= \frac{(s-j-2d)}{2(s-j)}\frac{(s-(j+1)+(2(d+\frac{1}{2})))}{2(s-(j+1))} \\ &= \frac{(s-j-2d)(s-j+2d)}{4(s-j)(s-j-1)} \\ &= \frac{(s-j+2d)}{2(s-j)}\frac{(s-(j+1)-(2(d-\frac{1}{2})))}{2(s-(j+1))} \\ &= Pr[down@j,d]Pr[up@j+1,d-\frac{1}{2}] \end{aligned}$$

This shows that at any point in a walk the pair of steps up-down (UD) is as likely to exist as the pair down-up (DU), independent of the rest of the walk. Thus, any such pairs can be interchanged in any walk to find an equally likely walk. This covers all possible biased random walks, showing they are all equally likely: UUUUDDDD=UUUDUDDD=UUDUDUDD, etc. □

Let us define an *unbiased random walk* to be a random walk starting at 0, and defined according to this following step function, with step size $\frac{1}{2}$:

$$\begin{cases} Pr[up@j,d] &= \frac{1}{2} \\ Pr[down@j,d] &= \frac{1}{2} \end{cases}$$

Lemma 2. *There is a one-to-one correspondence between the biased random walks of length n and the unbiased random walks of length n that end at* 0

Proof. Every *biased random walk* of length s can be mapped to a unique *unbiased random walk* that has the same sequence of steps, ending at 0. Moreover, every *unbiased random walk* of length s that ends at 0 can be mapped to a unique *biased random walk* that follows the same sequence of steps. □

As shown by Feller in 1967 [31], there is a one-to-one correspondence between random walks starting at position $(0,0)$, reaching (j,d) at some j along the way, and ending at $A=(s,0)$, with the random walks that start at position $(0,0)$ and end at $A'=(s,2d)$. Namely, for every walk that goes through (j,d) and eventually makes it to $(0,0)$, there is another walk that is equivalent until (j,d), at which point it begins mirroring the original walk over $y=d$. This provides an easy way to compute the number of walks that go "out of bounds". This reflection principle is illustrated in Figure 3.1.

Lemma 3. *There are $\binom{s}{\frac{s}{2}+2d}$ unbiased random walks from 0 to 0 that hit d on the way.*

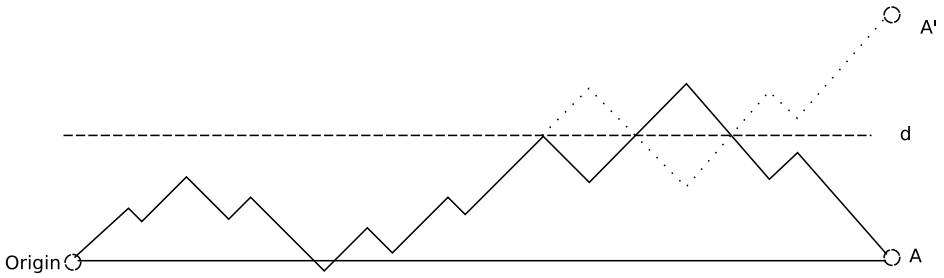

Figure 3.1: Feller's Reflection Principle [31]. Every one-dimensional walk of s steps that starts and ends at 0, but reaches a distance d from the origin along the way, corresponds to exactly one walk of length s that starts at 0 and ends at $2d$.

This lemma is included here for completeness; it may be found in Feller's book [31].

Proof. A walk of length s from $(0,0)$ to $(s,2d)$ consists of $4d$ more step-ups than step-downs (step sizes are still $\frac{1}{2}$). Therefore, it must consist of $\frac{s}{2}+2d$ step-ups and $\frac{s}{2}-2d$ step-downs. These steps can be in any order; therefore there are a total of $\binom{s}{\frac{s}{2}+2d}$ such walks.

The reflection principle shows that there are hence a total of $\binom{s}{\frac{s}{2}+2d}$ walks from $(0,0)$ to $(s,0)$ that hit d somewhere on the way. □

Lemma 4. *A randomly selected biased random walk hits d with probability* $\frac{\binom{s}{s/2+2d}}{\binom{s}{s/2}}$

Proof. Lemma 3 shows that there are $\binom{s}{\frac{s}{2}+2d}$ walks from $(0,0)$ to $(s,0)$ that hit d somewhere on the way. Lemma 2 shows that since each of these unbiased random walks goes from $(0,0)$ to $(s,0)$, it constitutes a valid *biased* random walk as well. Therefore, there are $\binom{s}{\frac{s}{2}+2d}$ biased random walks that hit d.

Lemma 1 shows that all biased random walks are equally likely. Since there are $\binom{s}{s/2}$ biased random walks total, a randomly chosen biased random walk hits d with probability

$$\frac{\binom{s}{s/2+2d}}{\binom{s}{s/2}}$$

□

Our queue size function is a biased random walk from $(0,0)$ to $(0,s)$ (with height 0 on the walk corresponding to size $c\sqrt{s/2}$ on the queue). The queue overflows, if at any point, the walk is above $c\sqrt{s/2}$ (and "under-runs" if below $-c\sqrt{s/2}$).

Lemma 4 shows that the chance a uniformly randomly selected biased walk hits d is $\binom{s}{s/2+2d}/\binom{s}{s/2} = \binom{s}{s/2-2d}/\binom{s}{s/2}$. By showing this quantity is negligible in c when $d = c\sqrt{s/2}$, I now show that with high probability the queue will remain within bounds.

We can upper-bound the value of $\frac{\binom{s}{s/2-2d}}{\binom{s}{s/2}}$ as follows: We have that

$$\frac{\binom{s}{s/2-2d}}{\binom{s}{s/2}} = \frac{\frac{s!}{(s/2-2d)!(s/2+2d)!}}{\frac{s!}{(s/2)!(s/2)!}} = \frac{(s/2)(s/2-1)\dots(s/2-2d+1)}{(s/2+2d)(s/2+2d-1)\dots(s/2+1)}.$$

Since the function $f(x) = \frac{x}{x+a}$ is increasing for $a > 0$, we conclude that

$$\frac{\binom{s}{s/2-2d}}{\binom{s}{s/2}} = \frac{(s/2)(s/2-1)\dots(s/2-2d+1)}{(s/2+2d)(s/2+2d-1)\dots(s/2+1)} \leq \left(\frac{s/2}{s/2+2d}\right)^{2d}.$$

For $d = c\sqrt{s/2}$ we get:

$$\frac{\binom{s}{s/2-2c\sqrt{s/2}}}{\binom{s}{s/2}} \leq \left(\frac{s/2}{s/2+2c\sqrt{s/2}}\right)^{2c\sqrt{s/2}} = \left(1 - \frac{1}{\sqrt{s/2}(1/(2c)+1/\sqrt{s/2})}\right)^{2c\sqrt{s/2}}.$$

For $s \geq 2$, $1/\sqrt{s/2} \leq 1$ and thus we obtain:

$$\begin{aligned}\frac{\binom{s}{s/2-2c\sqrt{s/2}}}{\binom{s}{s/2}} &\leq \left(1 - \frac{1}{\sqrt{s/2}(1/(2c)+1)}\right)^{2c\sqrt{s/2}} \\ &= \left(\left(1 - \frac{1}{\sqrt{s/2}(1/(2c)+1)}\right)^{\sqrt{s/2}(1/(2c)+1)}\right)^{2c/(1/(2c)+1)}. \qquad (3.1)\end{aligned}$$

We substitute $m := \sqrt{s/2}(1/(2c)+1)$ to find that:

$$\left(1 - \frac{1}{\sqrt{s/2}(1/(2c)+1)}\right)^{\sqrt{s/2}(1/(2c)+1)} = (1-1/m)^m \leq e^{-1}.$$

From (3.1) we now conclude that

$$\frac{\binom{s}{s/2-2c\sqrt{s/2}}}{\binom{s}{s/2}} \leq e^{-2c/(1/(2c)+1)}.$$

For $c \geq \frac{1}{2}$,

$$\frac{\binom{s}{s/2-2c\sqrt{s/2}}}{\binom{s}{s/2}} \leq e^{-c}.$$

It remains to mention that in fact, for sufficiently large s, the value of $1/\sqrt{s/2}$ can be arbitrarily small, which implies that for large s, the term $e^{-2c(1/(2c)+1)}$ can be replaced by $e^{-2c(1/(2c)+1/\sqrt{s/2})}$ which can be made smaller than $e^{-2c(1/(2c)+\delta)}$ for any $\delta > 0$ by choosing

s sufficiently large. Therefore, the upper bound on the probability that a walk exceeds the allowed bounds can in fact be made close to e^{-4c^2}. It follows that to get the probability of a random walk of length s exceeding bounds $\pm c\sqrt{s/2}$ smaller than 2^{-100}, it is sufficient to set $s \geq 300$ and $c \geq 6$.

Theorem 4. *With probability $\geq 1 - 2e^{-c}$, Oblivious Merge Sort queues never overflow or empty early.*

Proof. We have already established that the queue size is a biased random walk of length s, offset by the constant $c\sqrt{s}$. Therefore, the chance that a queue hits $2c\sqrt{s}$ is $\leq e^{-c}$. Symmetrically, the chance that a queue hits 0 is also $\leq e^{-c}$, so the chance that a queue goes out of bounds during one instance of the merge sort phase is $\leq 2e^{-c}$. This is negligible for a security parameter c, which is independent of s.

For sorting of level i, which contains $\leq 4^i$ items, the number of steps in the walk is $s \leq 4^i \leq n$. Following from the bounds derived above on symmetric walk overflow, with probability $\geq 1-2e^{-c}$ the walk remains within $\pm c\sqrt{n}$, and $2c\sqrt{n} < \psi(n)$ memory suffices. □

In summary, the Oblivious Sort algorithm sorts all the data blocks on the server into their final permutation, without revealing anything that could allow the server to correlate the two permutations.

3.3 Merging Multiple Arrays: Oblivious Merge Scramble

I now describe an algorithm that performs an oblivious scramble on an array of size m, with $c\sqrt{m \log m}$ local memory, in $O(m \log \log m)$ time with high probability. This is based on the algorithm I present in Section 3.2, which scrambles an array obliviously in time $O(m \log m)$ by ways of a merge sort. When the application only requires a scramble, and not a complete sort, the asymptotic complexity can be improved by merging multiple arrays at once.

Informally, the algorithm is still a merge sort, except a random number generator is used in place of a comparison, and multiple arrays are merged simultaneously. The array is recursively divided into subarrays, which are then scrambled together in batches. The time complexity of the algorithm is better than merge sort since multiple subarrays are merged together simultaneously. Randomly selecting from the remaining arrays avoids comparisons among the leading items in each array, so it is not a comparison sort—allowing us improved asymptotics.

The Oblivious Scramble Algorithm proceeds recursively as follows, starting with the remote array split into subarrays of size $s = 1$, a security parameter c, and an array to scramble of size m.

1. For subarrays sized s, allocate $\lceil \sqrt{m/s} \rceil$ buffers of size $c\sqrt{s \log m}$ (requiring $c\sqrt{m \log m}$ space total)

2. Divide the m/s subarrays into batches of $\sqrt{m/s}$ subarrays to merge together.

3. For each of the $\frac{m/s}{\sqrt{m/s}} = \sqrt{m/s}$ batches:

   - Obliviously merge the subarrays in this batch together into one new subarray of size $(s)\sqrt{m/s} = \sqrt{ms}$, by performing the Oblivious Merge Step on the allocated buffers. The Oblivious Merge Step requires $c\sqrt{s \log m}$ local working memory for each of the $\sqrt{m/s}$ buffers, for a total of $c\sqrt{m \log m}$ working memory, and operates in $O(\sqrt{ms})$ time.

4. In the end there are $\sqrt{m/s}$ subarrays of size $\sqrt{ms}$. Recurse with this larger subarray size.

One recursion of this algorithm requires a single pass across the level, costing $O(4^i)$ for level i. Each pass brings the total number of subarrays from m/s to $\sqrt{m/s}$, and we repeat until there is one subarray left. After iteration p, the number of subarrays remaining will be $m^{1/2^p}$. There will be 2 subarrays left when $p = \log_2 \log_2 m$. Since it takes $\log_2 \log_2 m$ passes to go from m to 2 subarrays, and each pass involves a single read and write of the entire array, the total running time / communication complexity for running the oblivious scramble on an array sized m is $O(m \log \log m)$.

I now describe the last remaining piece of the Oblivious Merge Scramble Algorithm, the Merge Step.

3.3.1 Oblivious Merge Step

The Oblivious Merge Step, whose pseudo-code is included here, takes r arrays of size s, and merges them randomly into a single array of size rs, preserving the ordering among the input arrays in the output arrays: if an item a is before item b in original array i, it will also be before b in the final array.

The permutation is chosen uniformly randomly out of all permutations that preserve the ordering of the original input items. To ensure this, we will take rs steps, choosing an item from the front of one of the r arrays at every step. The choice is biased since we choose each item *without replacement* randomly from the remaining items. If a particular array has a items left at step j, it has a $\frac{a}{rs-j}$ chance of being chosen at this step. The pseudo-code uses the procedure `RandomlyChooseASubarray` for this purpose.

Implementation of such a random function is not quite trivial, so we also include its description. Given a set of partially filled arrays the `RandomlyChooseASubarray` procedure chooses, in $O(1)$ expected time, using $cr\sqrt{s} \leq c\sqrt{n}$ local memory, one of the arrays such that the probability of being chosen is proportional to the number of items remaining in the array.

```
B ← New remote destination buffer size rs
t ← size of local queues, 2c√(s log m)
for i = 1 to r do
    q_i ← empty queue stored locally, size t
    for x = 1 to t/2 do
        Enqueue(q_i, decrypt(readNextItemFrom(A_i)))
    end
end
/* At this point, each queue will have t/2 items                    */
for x = t/2 to rs + t/2 do
    if x ≤ rs then
        for i = 1 to r do
            Enqueue(q_i, decrypt(readNextItemFrom(A_i)));
        end
    end
    /* Now we've read r items; time to output r items               */
    for i = 1 to r do
        v ← RandomlyChooseASubarray
        x ← dequeue(q_v)
        writeNextItemTo(B, encryptWithNewNonce(x))
    end
end
return B
```

Procedure ObliviousMergeStep($A_1, \ldots A_r$)

RandomlyChooseASubarray Algorithm

This algorithm outputs a random permutation of the sequence of length sr consisting of r values each repeated s times, i.e., a random permutation of $1^s 2^s \dots r^2$. All permutations are chosen with equal likelihood. This algorithm requires $O(cr\sqrt{s})$ memory.

For first $sr/2$ selections, a table T of r numbers is used; each entry is initialized to s. For the selection, a random number v is simply chosen from $1 \dots sr$. The candidate array i is $\lfloor \frac{v}{sr} \rfloor$. If $v \mod sr < T[i]$, decrement $T[i]$ and return i; else pick a new v. This allows us to pick items without replacement, in expected $O(1)$ time, with only r working memory (to record the remaining items left in each array).

This will succeed in an average of 2 tries for the first $\approx sr/3$ choices. After this, we must decrease the target range (and thus decrease the expected number of required repicks). If the target number space is viewed as a dartboard with region size $T[i]$ proportional to $s - i$, this can be viewed as cutting off an empty half of the dartboard, to double the probability of hitting a region instead of an empty space. Importantly, this does not affect the odds that the next number output will be i: it remains at $T[i]/\sum_{j=0}^{r} T[j]$.

To achieve this, let us change to pick the random number v from $1 \dots sr/2$, this time choosing candidate array $\lfloor \frac{2v}{sr} \rfloor$, repicking if $v \mod sr/2 \geq T[i]$. This halving of the target range can be repeated in the future since the arrays are picked from with roughly equal frequency; the discrepancy between any two arrays will never pass $c\sqrt{s}$ as shown in the following theorem. Once the arrays are down near size $c\sqrt{s}$, they can no longer be divided into equal-sized arrays that are dense enough to successfully pick in a small number of tries. However, at this point the remaining sequence will now fit in local memory: there are only $rc\sqrt{s}$ numbers left to output. The table T is now expanded into a list of numbers: each i is repeated $T[i]$ times. A standard Fisher-Yates scramble is then performed, and the resulting sequence output.

Analysis of the Oblivious Merge Scramble

The key to obliviousness is to perform the random array merging without affecting the actual access pattern of reading from the server. In Section 3.2 this is implemented for 2 arrays; I now extend this to merge r arrays. By simply reading the input evenly at a fixed rate, and outputting the items indicated by the random function, the uniform nature of the random function will cause the output rates to be very similar with high probability.

In other words, we can maintain a series of caching queues that are fed at a certain rate. According to a random function, we remove items from the queues. By the nature of the random selection, with high probability the queues will never overflow from being dequeued too slowly, nor empty out from being dequeued too quickly, as shown in Theorem 12.

Now, to show that the Oblivious Merge Scramble produces output indistinguishable from a uniformly randomly chosen permutation (from here on referred to as a "random permutation"), we will start with a set of theorems proving that a single Oblivious Merge Step outputs a random permutation of the contents of its input queues. We will break this into three steps. First I show that choosing a random permutation is equivalent to

selecting according to a random interleaving of the input queues (Theorem 5). Second, I show that given random coin flips, the `RandomlyChooseASubarray` algorithm produces such a random ordering of the input queues (Theorem 6). Theorem 7 shows the combination of these procedures produces a random permutation of the elements of the input arrays.

Theorem 8 shows that these constructions can be combined to produce a uniform random permutation of a single array. Finally, I show that the ability to distinguish the output permutation from a random permutation reduces to the ability to distinguish these coin flips from random (Theorem 9).

Theorem 5. *Selection from a set of randomly permuted input queues, according to a random interleaving of the input queues, produces a random permutation of all the elements of the input queues. In particular, given input queues $I_1 \dots I_r$ of unique elements whose contents have been randomly permuted, removal from the queues according to a random permutation of the sequence $1^{|I_1|}2^{|I_2|} \dots r^{|I_r|}$, (the "interleaving"), produces a random permutation of all the elements of $I_1 \dots I_r$.*

Proof. Since each output element belongs to a single (un-permuted) input queue, each output permutation is generated by a unique set of input queue permutations and interleaving. The reverse mapping from output to input is identified thusly: each input queue permutation is obtained by taking the ordering in the output array of the elements belonging to the given input array. The interleaving is obtained by taking the input queue identifier of each element in the output array. It follows from this one-to-one mapping of all output permutations back to the unique input that generates it, that all output permutations are possible.

Moreover, all possible input array permutations are independent and equally likely (by definition). All possible interleavings are also equally likely (also by definition).

Because there is a one-to-one mapping from inputs to the outputs, all outputs are possible, and all combinations of input permutations and interleavings are equally likely, it follows that all output permutations are equally likely. □

I now show that we can build a random interleaving of the r subarrays by repeatedly removing and outputting the head of one of the r subarrays selected randomly, with probability weighted according to the number of elements left in each subarray. More specifically, we show that the order of subarray selection, denoting subarray i by the integer i, is a random permutation of the sequence $1^s2^s \dots r^s$, with all permutations equally likely.

Theorem 6. *Choosing elements of a sequence S of rs integers 1 through s, with element S_i (for i from 0 through $rs-1$) chosen according to the probability density function $p(S_i = x) = (r - \sum_{j=0}^{i-1}(S_j = x?1 : 0))/(rs - i)$, results in a random permutation of the sequence $1^s2^s \dots r^s$.*

Proof. I show that all sequences are equally likely to be chosen this way. First, I show that swapping any two consecutive elements in any given sequence S results in an equally likely sequence.

Take consecutive positions i and $i+1$ in a sequence S, with elements from $S_i = k$ and $S_{i+1} = l$, where $k \neq l$. The chance $P(S)$ of generating S using the incremental construction

is $p_{0,i-1}P(S_i = k|S_0..S_{i-1})P(S_{i+1} = l|S_0..S_i))p_{i+2,rs-1}$, where $p_{0,i-1}$ is the probability of choosing the portion of the sequence leading up to position i, according to the incremental definition. $P(S_i = k|S_0..S_{i-1})$ is the chance that a sequence starting as such will be followed by k, and is obtained from the definition, $(r - \sum_{j=0}^{i-1}(S_j = k?1 : 0))/(rs - i)$. Letting N_k be the number of times the element k appears in the subsequence $S_0..S_{i-1}$, this simplifies to $(r - N_k)/(rs - i)$. Likewise, $P(S_{i+1} = l|S_0..S_i)$ is seen to be $(r - N_l)/(rs - i)$, letting N_l similarly be the number of times the element l appears in the subsequence $S_0..S_{i-1}$. Note that N_l is also the number of times l appears in $S_0..S_i$, since $S_i \neq l$. Let us define $p_{i+2,rs-1}$ as the product of the probabilities of each choice in the rest of the sequence, $S_{i+2}..S_{rs-1}$. This allows us to express the likelihood of S as $P(S) = p_{0,i-1}\frac{r-N_k}{rs-i}\frac{r-N_l}{rs-i}p_{i+2,rs-1}$.

Consider now the sequence S' obtained by swapping positions i and $i+1$. Since $S'_0..S'_{i-1} = S_0..S_{i-1}$, the probability of obtaining $S'_0..S'_{i-1}$ is still $p_{0,i-1}$. The probability that prefix subsequence is to be followed by l is now $P(S'_i = l|S'_0..S'_{i-1}) = P(S'_i = l|S_0..S_{i-1}) = (r - N_l)(rs - i)$. The probability of choosing $S'_{i+1} = k$ is $P(S'_{i+1} = l|S'_0..S'_i) = (r - \sum_{j=0}^{i}(S'_j = l?1 : 0))/(rs - i) = (r - N_k)(rs - i)$. Finally, we see that the choice of the elements at the end of the sequence, $S'_{i+2}..S'_{rs-1}$ is unaffected by the order of elements at S'_i and S'_{i+1}. This is because the choice of each element depends only on the number of times each element precedes it, not the order of those elements. Thus, $P(S') = p_{0,i-1}\frac{r-N_l}{rs-i}\frac{r-N_k}{rs-i}p_{i+2,rs-1} = P(S)$.

Finally, observe that any permutation can be obtained by repeatedly swapping consecutive elements. Since we are equally likely to generate any two sequences that have only swapped consecutive elements, we are equally likely to generate any permutation. □

Theorem 7. *Running the Oblivious Merge Step on unbiased random coin flips and randomly permuted input arrays generates a random permutation.*

Proof. This follows from Theorems 5 and 6, and the definitions of the Oblivious Merge Step and `RandomlyChooseASubarray` algorithms: the Oblivious Merge Step selects a random permutation of input queues, according to an interleaving provided by `RandomlyChooseASubarray`. Theorem 6 shows this interleaving to be random, and Theorem 5 shows this output to be a random permutation. □

Theorem 8. *Given unbiased random coin flips, the Oblivious Merge Scramble produces a permutation selected with equal probability from all possible permutations of inputs.*

Proof. The scramble consists of a series of array merge-scrambles, starting with arrays of size 1 on the first merge and resulting in an array of size n after the last merge.

Let us use an inductive argument on the merge step j.

As a base case, the inputs to the first merge step ($j = 1$) are arrays of size 1. These arrays are already in the only possible permutation, so by definition, this is a permutation selected uniformly randomly from all possible permutations.

Assume as the inductive hypothesis that the input to step j is a set of random permutations of the input arrays.

The Oblivious Merge Scramble produces as the output of step j (the input to step $j + 1$) the set (referred to as a "batch of subarrays" in the Oblivious Merge Scramble definition)

of outputs of Oblivious Merge Step, which is run multiple times across a partition of the output arrays from step j. By Theorem 7, each Oblivious Merge Step instance outputs a permutation of the elements of those input arrays selected uniformly randomly. Thus the input to step $j + 1$ is a set of randomly permuted arrays.

By induction, the output of any merge step past $j = 1$ is a set of randomly permuted arrays. In particular, the output the final merge step ($j = \log \log n$), which is a set containing a single element, consists of a randomly permuted array. □

Now I show that replacement of the source of randomness can be safely replaced with a PRNG source indistinguishable from random. That is, I will show that an adversary with an advantage ϵ at distinguishing the output array from a random permutation of the contents of the input arrays also has an advantage ϵ at distinguishing the PRNG output from random.

Theorem 9. *(Replacement of the random source with a source* indistinguishable *from random.) The advantage of an adversary at distinguishing the output of Oblivious Merge Scramble from a uniformly randomly chosen permutation of the elements in the Merge Scramble input is equal to the advantage of the adversary at distinguishing the coin flips used in Merge Scramble from random.*

Proof. An adversary B, given a source of coin flips, can distinguish it from uniform random as such: simulate the Merge step procedure using the set of coin flips; run A on the output. The advantage of adversary B at distinguishing the set of coin flips from uniform random will equal to the advantage of A at distinguishing the merge step output from random.

Take an algorithm A with advantage ϵ at distinguishing the Oblivious Scramble output from a true random permutation. This allows construction of an algorithm B that distinguishes the output of the employed PRNG from random, as follows. Algorithm B is given a sequence of bits S, with the goal of outputting 0 if it believes they are random, and 1 if it believes they are from the PRNG. Algorithm B takes the input and runs the Oblivious Merge using the bit sequence S, to produce a permuted array P. It then returns $A(P)$.

By definition, the probability that $A(P) = 1$ for a randomly permuted array P is at least ϵ less than the probability that $A(P) = 1$ for an array constructed by the Oblivious Merge Step using bits from the PRNG. Since running the Oblivious Merge Step on random coin flips generates a random permutation, as shown in Theorem 7, the probability that $A(P) = 1$ when S is a random set of bits is at least ϵ greater than when S is chosen by the PRNG.

The advantage of the algorithm at distinguishing the permutation from random thus reduces to its advantage at distinguishing the PRNG output from random. □

I now show correctness of the probabilistic Oblivious Merge Step. As described above, the Merge Step starts with a warm up phase (input but no output) in which each of r local queues are half-filled with elements from the corresponding input array. Next, there is a series of iterations, which each read 1 element from each of the r arrays, and output r elements, chosen randomly from the input arrays, according to the random permutation. Finally, in a "cool down" phase (output but no input), the remaining elements are emptied out of the queues, again according to the permutation. The warm up and cool down phases

have no chance of overflowing or prematurely emptying the queues; I just need to show that the queues stay within their bounds in the main phase.

First consider the notion of a *zero-sum asymmetric random walk*. Given a list of scalars that sum to 0, in the form of sp scalars of value 1 and $s(1-p)$ scalars of value $-\frac{p}{1-p}$, a zero-sum asymmetric random walk consists of the sequence of steps described by a randomly selected ordering of those scalars. We will show that the queue sizes over an execution of the ObliviousMergeStep algorithm corresponds to a *zero-sum asymmetric random walk*, and moreover, that the overwhelming majority of random orderings result in a walk that remains within $\psi(n)$. Let us first build probabilistic bounds on the extents of a zero-sum asymmetric random walk.

Probabilistic bounds on a zero-sum asymmetric random walk

Here we are interested in the probability that a walk of length s, starting at 0, and containing sp $+1$-steps and $s(1-p)$ $-\frac{p}{1-p}$-steps, exceeds bounds $-c\sqrt{ps\log s}$ or $c\sqrt{ps\log s}$. I show that for any $s > 1$, this probability is negligible in c.

First, I show that we can upper-bound the probability that a walk drawn from a hypergeometric distribution (i.e. containing the fixed number of up-steps and down-steps) exceeds the given bounds by the probability that a walk drawn from the associated binomial distribution exceeds some bounds close to the given ones.

Definition 3. *We will say that a sequence* $(a_1, \ldots, a_s) \in \mathbb{R}^s$ exceeds $u \geq 0$, *if for some* $k \in \{1, \ldots, s\}$, $|\sum_{j=1}^{k} a_k| > u$.

To prove the following two statements, the Chernoff bound [52] is used, which we briefly recall.

Theorem 10 (Chernoff bound)**.** *For i.i.d. random variables* $X_j \in \{0, 1\}$ *with* $\mathbb{E}(X_j) = p$ *and* $\epsilon > 0$,

$$\Pr\left[\left|p - \frac{1}{s}\sum_{j=0}^{s-1} X_j\right| \geq \epsilon\right] \leq 2e^{-\frac{\epsilon^2}{4p}s}.$$

Lemma 5. *Let* $Y := (Y_1, \ldots, Y_s)$ *where* Y_j*'s are i.i.d. random variables such that* $P_{Y_j}(1) = p$ *and* $P_{Y_j}(-p/(1-p)) = 1-p$. *Let us assume that a sequence sampled uniformly from the set of sequences* $\mathcal{W}$ *of length* s, *containing* sp $+1$*-items and* $s(1-p)$ $-p/(1-p)$*-items, exceeds* u *with probability* κ. *Then a sequence drawn from* P_Y *exceeds* $u - \delta\sqrt{s}$ *with probability at least* $\kappa/2$ *for* $\delta \geq 2\sqrt{\log(4)p}$.

Proof. Note that $sp \in \mathbb{N}$ and $s(1-p) \in \mathbb{N}$. Let $\mathcal{U}$ denote the set of sequences from $\mathcal{W}$ exceeding u. First, notice that all sequences drawn from P_Y and having $s(p+d)$ positive items are equally likely (each of them is just a permutation of another one). Furthermore, any such sequence can be obtained from some sequence in $\mathcal{W}$, by replacing ds negative items with positive items (if $d > 0$) or by replacing ds positive items by negative items (if $d < 0$). Let us analyze the first case in detail. The transformation can be done in $\binom{s(1-p)}{ds}$ ways,

since a sequence from $\mathcal{W}$ contains exactly $s(1-p)$ negative items. On the other hand, each sequence having $(p+d)s$ positive items can be obtained from at most $\binom{(p+d)s}{ds}$ sequences from $\mathcal{W}$. Therefore, replacing ds positive items by negative items in all sequences from $\mathcal{U}$, maps these sequences into at least $|\mathcal{U}|\binom{s(1-p)}{ds}/\binom{(p+d)s}{ds}$ sequences having $(p+d)s$ positive items. Furthermore, any such sequence exceeds $u - ds(1+p/(1-p))$. Let a set of these sequences be denoted by $\mathcal{U}_d$. Let $\mathcal{W}_d$ denote the set of all sequences having $(p+d)s$ positive items. Since

$$|\mathcal{W}_d| = \binom{s}{s(p+d)},$$

we obtain

$$\frac{|\mathcal{W}_d|}{|\mathcal{W}|} = \frac{\binom{s}{s(p+d)}}{\binom{s}{ps}} = \frac{s(1-p)\dots(s(1-p-d)+1)}{s(p+d)\dots(sp+1)} = \frac{\binom{s(1-p)}{ds}}{\binom{(p+d)s}{ds}}.$$

Consequently,

$$\frac{|\mathcal{W}_d|}{|\mathcal{W}|} = \binom{s(1-p)}{ds}/\binom{(p+d)s}{ds} \leq \frac{|\mathcal{U}_d|}{|\mathcal{U}|},$$

yielding

$$\kappa \leq \frac{|\mathcal{U}|}{|\mathcal{W}|} \leq \frac{|\mathcal{U}_d|}{|\mathcal{W}_d|}.$$

For $d < 0$ we get an analogous result. Therefore, we can write:

$$\begin{aligned}
&\Pr[Y \text{ exceeds } (u-\delta\sqrt{s})] \geq \Pr\left[Y \in \bigcup_{j=-\delta\sqrt{s}}^{\delta\sqrt{s}} \mathcal{U}_{j/(s(1+p/(1-p)))}\right] \\
= &\sum_{j=-\delta\sqrt{s}}^{\delta\sqrt{s}} \Pr[Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] \cdot \Pr[Y \in \mathcal{U}_{j/(s(1+p/(1-p)))} | Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] \\
\geq &\kappa \sum_{j=-\delta\sqrt{s}}^{\delta\sqrt{s}} \Pr[Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] \geq \kappa\left(1 - 2e^{-\frac{(\delta\sqrt{s}/s)^2}{4p}s}\right) = \kappa\left(1 - 2e^{-\frac{\delta^2}{4p}}\right) \geq \kappa/2,
\end{aligned}$$

since according to the assumption for δ, $2e^{-\frac{\delta^2}{4p}} \leq \frac{1}{2}$. To conclude that for each j, $\Pr[Y \in \mathcal{U}_{j/(s(1+p/(1-p)))} | Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] \geq \kappa$, we will use the fact that all walks having the same number of steps up occur with the same probability. Inequality

$$\sum_{j=-\delta\sqrt{s}}^{\delta\sqrt{s}} \Pr[Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] \geq 1 - 2e^{-\frac{\delta^2}{4p}}$$

follows from the Chernoff bound. □

Now we are ready to formulate and prove the desired statement.

Theorem 11. *Let $p < \frac{1}{2}$. Let W be a random variable distributed uniformly on the set of all walks $\mathcal{W}$ of length s, starting at 0, and containing sp $+1$-steps and $s(1-p)$ steps of $-p/(1-p)$. For such a walk w, let $f(w) = 1$ if and only if w exceeds $c\sqrt{ps\log s}$. Let $F_W := f(W)$. Then for each $s > 1$, $P_{F_W}(1) \leq 4se^{-\frac{c^2 \log s(1-p)^2}{16}}$ which is negligible for $c \geq 4\sqrt{\log(4)}/\sqrt{\log s}$.*

Proof. For a walk y of length s, starting at 0, and containing $+1$ and $-p/(1-p)$ steps, let $f'(y) = 1$ if and only if y exceeds $c\sqrt{ps\log s}/2$. Again, let $Y = (Y_1, \ldots, Y_s)$ where Y_j's are i.i.d. random variables with $P_{Y_j}(1) = p$ and $P_{Y_j}(-p/(1-p)) = 1-p$. Let $F'_Y := f'(Y)$. Lemma 5 then implies that for $c\sqrt{\log s} \geq 4\sqrt{\log(4)}$,

$$P_{F_W}(1) \leq 2P_{F'_Y}(1).$$

We will now bound $P_{F'_Y}(1)$, and show that it is negligible in c.

To prove the statement, we use the Chernoff bound and the union bound. For $j \in [s]$, let $X_j \in \{0,1\}$ be i.i.d. random variables with $\mathbb{E}(X_j) = p$. Random variables $Z_j := \frac{1}{1-p}X_j - \frac{p}{1-p}$ then correspond to the steps of a random walk described in the statement. Let $u > 0$. We get $\sum_{i=0}^{s-1} Z_j > u$ if and only if

$$\frac{1}{s}\sum_{j=0}^{s-1} X_j > p + \frac{u(1-p)}{s} = \mathbb{E}(X_j) + \frac{u(1-p)}{s}.$$

Analogously, we get $\sum_{j=0}^{s-1} Z_j < -u$ if and only if

$$\frac{1}{s}\sum_{j=0}^{s-1} X_j < \mathbb{E}(X_j) - \frac{u(1-p)}{s}.$$

According to the Chernoff bound,

$$\Pr\left[\left|p - \frac{1}{s}\sum_{j=1}^{s-1} X_j\right| \geq \frac{u(1-p)}{s}\right] \leq 2e^{-\frac{u^2(1-p)^2}{4ps}}$$

yielding

$$q_s := \Pr\left[\left|\frac{1}{s}\sum_{j=1}^{s-1} Z_j\right| \geq u\right] \leq 2e^{-\frac{u^2(1-p)^2}{4ps}},$$

where q_s denotes the probability that a random walk of length s exceeds either u or $-u$ at the last step. Let p_s denote the probability that a random walk of length s exceeds either u or $-u$ at any step (i.e. exceeds u according to Definition 3). By applying the union bound, we get

$$p_s \leq \sum_{j=0}^{s-1} q_j. \tag{3.2}$$

Since $g(s) := 2e^{-\frac{u^2(1-p)^2}{4ps}}$ is an increasing function, we can write:

$$p_s \leq \sum_{j=0}^{s-1} q_j \leq 2se^{-\frac{u^2(1-p)^2}{4ps}}. \tag{3.3}$$

Therefore, p_s is negligible if $u \geq c\sqrt{ps\log s}/2$ (for $c \geq 4\sqrt{\log(4)}/\sqrt{\log s}$).

It follows that the probability that a walk drawn from P_W exceeds $c\sqrt{ps\log s}$ with probability $2p_s$, is also negligible. □

Theorem 12. *The Oblivious Merge Step succeeds, with high probability: for any s such that $1 \leq s \leq m/2$, the chance that the queue buffers sized $c\sqrt{s\log m}$ overflow or under-run is negligible w.r.t. the security parameter c.*

Proof. Each instance of the Oblivious Merge Step reads from the set of $r = \sqrt{m/s}$ input arrays sized for some $1 \leq s \leq m/2$ in a random order. This order is chosen according to a random permutation of $1^s 2^s \ldots r^s$, as shown in Theorem 6. The walk is length $rs = \sqrt{ms}$. The size versus time of any array in the merge step is a walk chosen according to a random permutation of s $(+1)$-steps and $s(r-1)$ steps sized $(-(1-r)/r)$.

Setting $p = 1/r = 1/\sqrt{ms}$ and $z = rs = \sqrt{ms}$, Theorem 11 shows that the chance of such a walk exceeding a bound of $\pm c\sqrt{s\log\sqrt{ms}}$ is negligible in c. Since $s < m$, $c\sqrt{s\log\sqrt{ms}} < c\sqrt{s\log m}$, and the buffers sized $c\sqrt{s\log m}$ suffice. □

Theorem 13. *Any advantage of the adversary (server) at deciding any function of the permutation output by Oblivious Merge Scramble on input of a publicly known size,* gained by learning *the sequence of reads and write tuples of the form ("read" or "write",location,value), translates into a violation of the semantic security of the encryption.*

Proof. The sequence of locations read and written by the oblivious scramble defined in Algorithm ObliviousMergeStep is deterministic for a given input array size and *independent of the permutation being performed*. The encrypted blocks are all equivalently sized, and *value* is always a fresh output of a semantically secure encryption algorithm.

Any knowledge gained about the permutation from the encrypted values directly implies the server's ability to distinguish between or correlate different instances of encrypted equally sized data values. This violates the semantic security of the encryption algorithm. □

Chapter 4

An Applied Approach to Existing ORAMs

In this chapter I introduce a technique speeding up Oblivious RAM by an order of magnitude. In the presence of a small amount of temporary memory (enough to store $O(\sqrt{n \log n})$ items and IDs, where n is the number of items in the database), we will achieve access pattern privacy with computational complexity of $O(\log^2 n)$ per query (as compared to e.g., $O(\log^3 n)$ for the traditional Oblivious RAM).

I achieve these novel results by applying the probabilistic-minded insights presented in Chapter 3 to Oblivious RAM, allowing me to significantly improve its asymptotic complexity. Not only do I replace the sorting networks with my fast Oblivious Merge Sort, but I show how to use related techniques to avoid paying for the costly shuffle of $O(n \log n)$ fake items.

This results in a protocol crossing the boundary between theory and practice and becoming generally applicable for access pattern privacy. I show that on off-the-shelf hardware, large data sets can be obliviously queried, orders of magnitude faster than in existing work.

4.1 A Solution

My solution deploys new insights based on probabilistic analyses of data shuffling in ORAM allowing a significant improvement of its asymptotic complexity. These results can be applied under the assumption that clients can afford a small amount ($\psi(n) = 4c\sqrt{n \log n}$ blocks, where c is a constant) of temporary working memory.

For simplicity, this chapter assumes a curious but honest provider. Among other things, the next chapter shows how to remove this assumption.

4.1.1 Additional Client-side Working Memory

Simply adding memory to ORAM in a straightforward manner does not significantly improve its complexity. Consider that there are two stages where additional memory can be deployed. First, the top levels could be stored exclusively on the client, allowing the bypassing of all

reads and writes to the top levels, as well as the re-shuffling of these levels. Since there are $\log_4 n$ levels, with sizes (number of buckets) 4^i for i from 1 to $\log_4 n$, the blocks belonging to the first $\log_4(\psi(n)) = \log_4(4c\sqrt{n \log n}) = \log_4 4c + \frac{1}{2}\log_4 n + \frac{1}{2}\log_4 \log n$ levels can fit in this memory. This however, would only eliminate slightly more than a constant fraction of the $\log_4 n$ levels, leaving the most expensive levels operating as before.

Second, as indicated by Goldreich and Ostrovsky [41], additional client-side memory can be deployed in the sorting network used in the level reshuffle. The sorting network is the primitive that performs all the level reordering, requiring $O(n \log n)$ time for the client to obliviously sort data on the server (with no client memory). Then, in the presence of additional memory, the normal sorting network running time can be improved by performing comparisons in batches on the client. However, performing batch comparisons with a limited amount of memory does not greatly improve the complexity of the sorting network. Moreover, no amount of memory can cause the sorting network to do a comparison sort (as required in Oblivious RAM) in the standard computational complexity model better than $\Omega(n \log n)$ [25]. This still results in overall amortized overhead of $\Omega(\log^3 n)$. [1]

An Approach. I propose to tackle the complexity of the most time-consuming phase of ORAM, the level reorder step. We shall take advantage of the consistent nature of uniform random permutations to perform an oblivious scramble with a low complexity and little client memory. The intuition is that given two halves of an array consisting of uniformly randomly permuted sequence of items, the items will be distributed between the halves almost evenly. That is, if we pick the permuted items in order, counting the number of times each array half is accessed, the counts for each array half remain close for the entire sequence, with high probability.

This allows us to implement a novel merge sort that hides the order in which items are being pulled from each half. Once the two array halves are each sorted and stored on the server, we can combine them into a sorted whole by reading from each half into the client buffer, then outputting them in sorted order without revealing anything about the permutation. By the uniform nature of the random permutation, for arrays of size n, I show that the running tally of picks from each array half will never differ by more than $\psi(n)$, with high probability in the security parameter c. This means that we can preset a read pattern from the server without knowing the permutation, and still successfully perform the permutation! The pattern of accesses between the two array halves will deviate slightly, but with high probability they will fall within the window of $\psi(n)$ from the fixed pattern.

This oblivious merge sort is the key primitive that allows us to implement access pattern privacy with $O(\log^2(n))$ overhead. We will use it to implement a random scramble, as well as to remove the fake blocks that are stored in each level. Being able to do both of those steps efficiently, we can then replace the oblivious permutation used in ORAM with a more efficient version.

Recall that buckets in ORAM are sized $\lambda \log n$, where λ is a security parameter chosen to

[1] Considering cost in other models, besides asymptotic communication / computational overhead, however, can be useful, since the cost of making $\Omega(n \log n)$ numerical comparisons is typically smaller than the $\Omega(n)$ required network transfer.

reduce the possibility of a bucket overflow due to hash function collisions—representing an access pattern leak—to $< e^{-\lambda}$. See the balls and bins result used in 14 for a precise relation between the overflow probability and λ.

From here on, we will be concerned mainly with the process of re-ordering a level, since the rest of my algorithm is unchanged from ORAM. A level re-ordering entails taking the entire contents of level i, consisting of 4^i buckets of size $\lambda \log n$, containing a total number of real blocks between 4^{i-1} and 4^i, with the remainder filled with fake blocks, and rearranging them to the new permutation obliviously—without revealing anything about the new permutation to the server.

4.1.2 Strawman: Client with n Blocks of Memory

Before describing the main result, let us informally analyze a strawman algorithm that achieves my desired time complexity, in the presence of enough client-sided memory to fit the *entire* database (n blocks).

Observe that if the client has n blocks of temporary memory, it can perform a reorder of level i in only $O(4^i \log n)$ steps, in the following manner. By reading the entire level into the temporary memory, throwing out the fake blocks as they are encountered, it can store all 4^i blocks locally. This requires processing $4^i \lambda \log n$ blocks (4^i real ones, and the rest fakes) It then performs a comparison sort on the local memory (which is hence done without revealing the new permutation to the server) to permute these blocks to their new location at a computational cost of $O(4^i \log_2(4^i)) = O(i4^i)$. The blocks are then all re-encrypted with new nonces for a cost of $O(4^i)$ (so the server is unable to link old to new blocks). Copying this data back to the server, while inserting fakes to fill the rest of the buckets, requires writing another $4^i \lambda \log n$ blocks to the server, for a cost of $O(4^i \log n)$. The total cost of reordering is $O(4^i(i + \log n)) = O(4^i \log n)$.

Since each level i overflows into level $i + 1$ once every 4^i accesses, level $i + 1$ must be reordered at each such occurrence. As there are $\log_4 n$ levels total, the amortized computational cost per query of this level reordering approach, across all levels, can therefore be approximated by

$$\sum_{i=1}^{\log_4 n} \frac{O(4^i \log n)}{4^{i-1}} = \sum_{i=1}^{\log_4 n} O(\log n) = O(\log^2 n)$$

This offline level reordering cost must be paid in addition to the online query cost to scan a bucket at each level. This part of the algorithm is equivalent to ORAM. Since the buckets have size $\lambda \log n$, the online cost of scanning $\log_4 n$ buckets is $O(\log^2 n)$. Thus the average cost per query, including both online cost and amortized offline cost, is $O(\log^2 n)$.

In summary, in the presence of $O(n)$ client memory the amortized running time for ORAM can be cut down from $O(\log^3 n)$ to $O(\log^2 n)$. Of course, assuming that the client has n blocks of local working memory is not necessarily practical and could even invalidate the entire cost proposition of server-hosted data. Thus little has been gained so far.

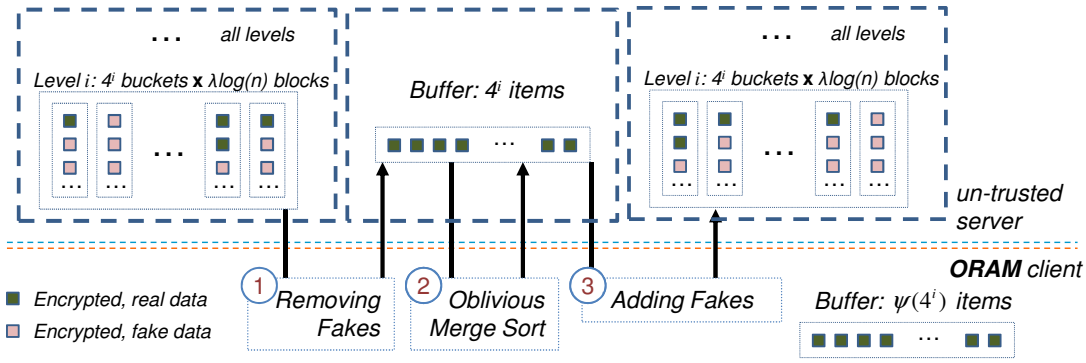

Figure 4.1: Solution Overview. Building a new level consists of obliviously removing fakes from the old levels, obliviously permuting, and obliviously adding new fakes back (to provide uniform bucket sizes).

4.1.3 Overview: Client with only $\psi(n)$ Blocks of Memory

I will now describe an algorithm for level re-ordering with identical time complexity, but requiring only $\psi(n) = 4c\sqrt{n \log n}$ blocks of local working memory from the client. The client's reordering of level i is divided into Phases (refer to Figure 4.1). We now overview these phases and then discuss details.

1. **Removing Fakes.** Copy the 4^i original data blocks at level i to a new remote buffer (on the server), obliviously removing the $(\lambda \log n - 1)4^i$ fake blocks that are interposed. Care must be taken to prevent revealing which blocks are the fakes—thus copying will also entail their re-encryption. This decreases the size of the working set from $4^i \lambda \log n$ to 4^i if the level is full, or to $\frac{1}{4}4^i$, $\frac{1}{2}4^i$, or $\frac{3}{4}4^i$ for the first, second, and third reorderings of this iteration of level i. Let us assume we are dealing with a full level (fourth reordering) to make the remainder of this description simpler; earlier reorderings proceed equivalently but with slightly lesser time and space requirements. The computational complexity of this phase is $O(4^i \log n)$. (see Section 4.1.4)

2. **Oblivious Merge Sort.** Obliviously merge sort the working set in the remote buffer, placing blocks into their final permutation according to the new hash function for this level. Perform the merge sort in such a way that the server can build no correlation between the original arrangement of blocks and the new permutation. The computational complexity of this phase is $O(4^i \log n)$. (see Section 4.1.5)

3. **Add Fakes.** Copy the 4^i blocks, which were permuted by Phase 2 into their correct order, to the final remote storage area for level i. They are not in buckets yet, so we build buckets, obliviously adding in the $4^i(\lambda \log n - 1)$ fake blocks necessary to

guarantee all buckets have the same size. The computational complexity of this phase is $O(4^i \log n)$. (see Section 4.1.6)

The above algorithm reorders level i into the new permutation, in time $O(4^i \log n)$. Therefore the derivation of the amortized overhead is equivalent to the derivation performed for the strawman algorithm, leading to an amortized overhead of $O(\log^2 n)$ per query. I now show how to efficiently implement each phase, using only $\psi(n)$ local memory.

4.1.4 Phase 1: Remove Fakes

Fake blocks can be removed from level i in a single pass, without revealing them, by copying into a temporary buffer that hides the correspondence between read blocks and output blocks (refer to the illustration in Figure 4.2 and provided pseudo-code for Procedure RemoveFakes). The client scans the level, storing the real blocks into a local queue and tossing the fake blocks. Once the queue is expected to be *half full*, the client starts writing blocks from the queue (while also continuing the scan), at a rate corresponding to the overall ratio of real to fake blocks. The goal is to keep the queue about half full until the end. (The server can observe the total number of fake and real blocks in a particular level, which is independent of the data access pattern.) Assuming the temporary queue never overflows or empties entirely until the end, the exact pattern of reads and writes observed by the server is dependent only on the number of blocks, and the ratio of fakes. The server learns nothing of which are the fake blocks by observing the fake removal scan. I show in Theorem 14 that, with high probability, a queue of size $\psi(n)$ will not overflow or empty out.

This phase requires time linear to the size of the level being read. For level i, which contains 4^i buckets of size $\lambda \log n$, the running time is $O(4^i \log n)$.

Since the location of real blocks is determined by a secure hash function on the unique block index, the distribution of the blocks (and fakes) is indistinguishable from the case where each real block is assigned a bucket, chosen independently and uniformly at random from the set of all buckets on the actual level.

Theorem 14. *The probability the Remove Fakes queue overflows or empties early is negligible in c. I.e., for $\lambda \log n > 4$ and $c \geq 4\sqrt{\log(4)}/\sqrt{\log s}$, the probability of failure due to overflow or underflow is $\leq 2e^{-c^2 \frac{1}{64}}$, where $\lambda \log n$ denotes the bucket size.*

Proof. Using balls and bins analysis to bound the acceptable probability of a bucket overflow to p' (given n balls and n bins), we take $\lambda = \max\{\mathrm{e}, \frac{\log 1/p'}{(\log n) \cdot (\log \log n)}\}$. Because this probability derives from a balls and bins result, note that $\lambda \geq n/\log n$ provides an overflow probability of exactly 0 (since we are only placing n items into bins sized $\lambda \log n$), so it is safe to assume $\lambda \leq n/\log n < n$.

First, observe that the real items are positioned according to a uniform random distribution among the fake items. Since a secure hash function builds a uniform random distribution of real items in buckets (each real item is equally and independently likely to end up in any

```
/* RemoveFakes:  Scan all items at level i on the server, and write them
   back excluding the fake ones.  Use a local buffer to hide which were
   the fake ones.                                                        */
q ← empty queue stored locally, with room for up to ψ(n) elements
r ← ratio total blocks to real blocks (λ log(n))
/* There are 4^i real items and (r − 1) ∗ 4^i fakes at this level.      */
y ← 0 /* Number of blocks output thus far                               */
for x = 1 to r ∗ 4^i + r ∗ ψ(n)/2 do
    if x < r ∗ 4^i then
        /* Read one item on every iteration (until all are read)       */
        t ← decrypt(readNextBlockFromLevel(i));
        if t is a real block then
            q.enqueue(t)
        end
    end
    if y < x/r − ψ(n)/2 then
        /* Output one real item every r iterations, starting once the queue
           is expected to be about half full.                          */
        y ← y + 1
        t ← q.dequeue()
        writeNextBlockToRemoteBuffer(encrypt(t))
    end
end
```

Procedure RemoveFakes(i)

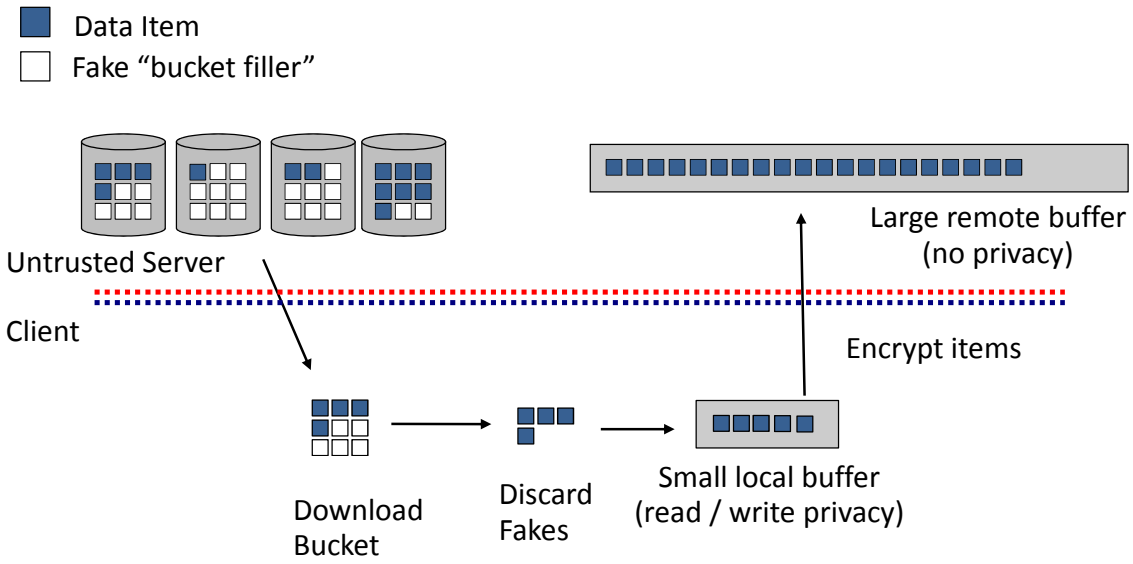

Figure 4.2: Reshuffle Phase 1: Remove fakes. In a single pass, the client obliviously removes all the fake items from remotely stored buckets, cutting the sort data size by a factor of $\lambda \log n$. The total number of fakes is public, but the location of the fakes is secret.

bucket), the fake items are also distributed uniformly randomly; that is, the permutation of real and fake items is chosen with equal probability from all possible permutations.[2]

Take r to be our bucket size, $\lambda \log n$. Disregarding for now the first and last $r\psi(4^i)/2$ iterations, which are used to prime the queue at the beginning and empty it at the end, the queue size in the RemoveFakes algorithm can be modeled by a zero-sum asymmetric random walk, according to this distribution of fake and real items. On the iterations where a real item is encountered, the queue size increases by 1 (due to the queue append command). Once every r iterations, the queue size decreases by 1.

The queue is easier to model if we assume that instead of decreasing in size by 1 once every r iterations, the decrease is amortized over the r iterations. This idealized queue size differs by at most 1 at any point from the actual queue size. Thus, there are 4^i iterations in which the queue size increases by 1, and in all iterations, the queue size decreases by $1/r$. This is represented by a walk that increases by $1 - 1/r$ in 4^i steps and decreases by $1/r$ in $(r-1)4^i$ steps. On any particular execution, the ordering of these steps is chosen uniformly randomly from the set of all possible orderings.

Theorem 11 states that zero-sum asymmetric random walk containing sp steps sized $+1$ and $s(1-p)$ steps sized $-p/(1-p)$ remains within $\pm c\sqrt{ps \log s}$ with high probability

[2]The presence of hash collisions (and buckets to resolve these collisions) creates a potential deviation from this idealized uniform distribution. This is resolved if we ensure all items in a bucket are stored in a random permutation. This requirement is unnecessary in practice because our idealized queue size remains within $\pm\lambda \log n$ of the actual queue size, even without the requirement of individual bucket contents being randomly permuted. This is because the buckets are of size $\lambda \log n$.

in c. Specifying $s = r4^i$, and $p = 1/r$ proves that that a walk containing 4^i steps sized 1 and $r4^i(r-1/r) = (r-1)4^i$ steps sized $-1/r/(1-1/r) = -1/(r-1)$ remains within $\pm c\sqrt{(1/r)(r4^i)\log(r4^i)} = \pm c\sqrt{(4^i)\log(r4^i)}$ with high probability in c.

In order to apply this result, we scale the step size down to $(1-1/r)$ on the up-step, multiplying by $1-1/r$. This gives us 4^i step-ups of size $1-1/r$, and $(r-1)4^i$ step-downs of size:

$$\frac{1}{r-1}\left(1-\frac{1}{r}\right) = \frac{1}{r-1}\left(\frac{r-1}{r}\right) = \frac{1}{r}$$

Thus, a random walk containing 4^i steps sized $1-1/r$ and $(r-1)4^i$ steps sized $1/r$ remains within $\pm(1-1/r)c\sqrt{(4^i)\log(r4^i)}$ with high probability in c.

The largest number of real items a level can possibly contain, for a database sized n, is n items. On the bottom level, where $i = \log_4 n$, $4^i = n$ (recall that a level may have from 4^{i-1} to 4^i items during the shuffle; we consider the maximum 4^i items here). On every other level i, $4^i < n$. Thus, since $4^i \le n$, and with $r = \lambda \log n$, with high probability, the random walk for level i remains under

$$\begin{aligned}(1-\frac{1}{r})c\sqrt{(4^i)\log(r4^i)} &< c\sqrt{n\log(n\lambda\log n)} = c\sqrt{n}\sqrt{\log n + \log\log n + \log\lambda} \\ &< c\sqrt{n}\sqrt{3(\log n)} < 2c\sqrt{n\log n} = \psi(n)/2.\end{aligned}$$

To apply the upper bound on this probability derived in Theorem 11, we substitute $s = n\lambda\log n$ and $p = 1/\lambda\log n$.

For $c \ge 4\sqrt{\log(4)}/\sqrt{\log(n\lambda\log n)}$, We obtain that the probability that a walk consisting of n +1-steps and $n\lambda\log n - n$ steps of -1/$(\lambda\log n - 1)$ will overflow bounds of $c\sqrt{n\log(n\lambda\lg n)} < \psi(n)/2$ with probability

$$\le 4(\lambda n\log n)e^{-\frac{c^2\log(\lambda n\log n)(1-\frac{1}{\lambda\log n})^2}{16}} = 4e^{-\frac{c^2(1-\frac{1}{\lambda\log n})^2}{16}}$$

which decreases in n. For, e.g., $\lambda\log n > 4$, this quantity is less than $2e^{-c^2\frac{1}{64}}$.

An equivalent symmetric argument holds for the lower bound on the random walk. I have shown that with high probability in c, the walk representing the queue size never exceeds $\pm\psi(n)/2$. The actual queue size must never be negative, of course, and we solve this apparent discrepancy using the first and last $r\psi(n)/2$ steps of the walk. In particular, items are never removed from the queue during the first $r\psi(n)/2$ iterations. This has the effect of increasing the number of items in the queue in relation to the model walk by $(1/r)\psi(n)/2$ each iteration. Thus, past the beginning of the walk, and until the cleanup phase at the end, the actual queue size contains exactly $\psi(n)/2$ items more than the model walk. Thus, since the model walk remains within $\pm\psi(n)/2$, accounting for this difference, the actual queue size remains between 0 and $\psi(n)$.

I just need to show now that the queue size is also within bounds in the beginning and ending phases. In the beginning, observe that an "underflow" is impossible since we are not removing items in this phase. Moreover, since the model walk remains below $\psi(n)/2$,

and since our actual queue size is at any point always x more than the model walk, for some $0 \leq x \leq \psi(n)/2$, our actual queue size in the beginning is below $\psi(n)$. Finally, in the ending phase (the last $r\psi(n)/2$ iterations), we are past the end of the walk since we are no longer reading items from the server. The job here is simply to empty out the queue, which contains exactly $\psi(n)$ at the end of the walk, since the model walk ends at 0. Thus, overflow/underflow does not occur here either.

□

To summarize, Phase 1 copies all of the real blocks out of level i, into a new remote (server-side) storage buffer that only contains real blocks. In copying, a small local (client-side) buffer is used to avoid leaking which blocks were fake. This is possible since the fake blocks are evenly distributed throughout the level.

4.1.5 Phase 2: Oblivious Merge Sort

An oblivious merge sort, as described in the previous chapter, is performed, sorting items by the new keyed hash of its item ID. This obliviously puts the level items into a new, random, permutation, according to the new hash function for this level.

4.1.6 Phase 3: Add Fakes

In the final phase, the permuted blocks are added back to server-hosted buckets where they will be located by the next iteration of the secure hash function. At the same time, fake blocks are added to make all buckets mutually indistinguishable. This is the exact inverse of Phase 1. Pseudo-code is provided for Procedure AddFakesToLevel; an illustration is provided in Figure 4.3.

For correctness I must also show here that the buckets of size $\lambda \log n$ will not overflow. It is easy to see that if s blocks are randomly thrown into s buckets, the fullest bucket has more than t blocks with probability at most $\frac{s}{t!}$. This follows from the fact that each bucket gets at least t blocks with probability at most $\binom{s}{t}\frac{1}{s^t} \leq \frac{1}{t!}$ and the union bound. Using Stirling's approximation of $t! \approx e^{t\log(t)-t}$, it follows that for $t = \lambda \log s$, the probability that each bucket gets at most t blocks is therefore at least $1 - (s/t!) \approx 1 - s^{-\lambda\cdot(\log\log s + \log(\lambda))}$. If we fix the probability of bucket-overflow to be at most p', then for a database of size n (which is the size of the lowest pyramid level), $\lambda \log n$-sized buckets are sufficient for $\lambda = \max\{e, \frac{\log 1/p'}{(\log n)\cdot(\log\log n)}\}$. For all practical purposes, λ can be considered a constant once p' is fixed. For example, for a reasonable $p' = 2^{-100}$, if $n > 2^{14}$ (more than 16384 blocks) $\lambda = e$.

When blocks are placed into the buckets on higher levels, having less than n buckets, the probability of bucket-overflow is upper-bounded by the probability that buckets on the lowest level overflow. In the event that an overflow ever does occur, the original ORAM [41] prescribes a level re-order abort and restart; the expected time complexity of that algorithm is not affected. However, notice that by conditioning on the fact that no bucket ever overflows, we differentiate the distribution of bucket assignments from the truly uniform one. Since the original ORAM algorithm prescribes to randomize the pyramid traversal once the desired

```
/* AddFakesToLevel: Scan all items in array A on the server, which are
   sorted by the bucket they belong in, and write them to level i, adding
   fakes to make equal-sized buckets. Use a local buffer to hide which
   are the fake ones. */
q ← empty queue stored locally, with room for up to ψ(n) elements
r ← ratio total blocks to real blocks (λ log(n))
y ← 0 /* Number of blocks read so far */
b ← 0 /* Current bucket number */
q′ ← empty queue stored locally, with room for r elements
for x = 1 to r ∗ 4^i + r ∗ ψ(n)/2 do
    if y < x/r and y < 4^i then
        /* Input one real item every r iterations. */
        y ← y + 1
        q.enqueue(decrypt(readNextBlock(A)));
    end
    if x > r ∗ ψ(n)/2 then
        /* Add one item to the bucket every iteration, real if there is one
           waiting for this bucket, fake otherwise (starting once the queue
           is about half full) */
        t ← peek(q)
        if BucketNumber(t) = b then
            q′.enqueue(encryptWithNewNonce(q.dequeue(),t))
        else
            q′.enqueue(encryptWithNewNonce(fake))
        end
        if b < x/r − r ∗ ψ(n)/2 then
            /* Output one bucket every r iterations. */
            writeBucketToLevel(q′,i)
            b ← b + 1
            empty(q′)
        end
    end
end
```

Procedure AddFakesToLevel(A,i)

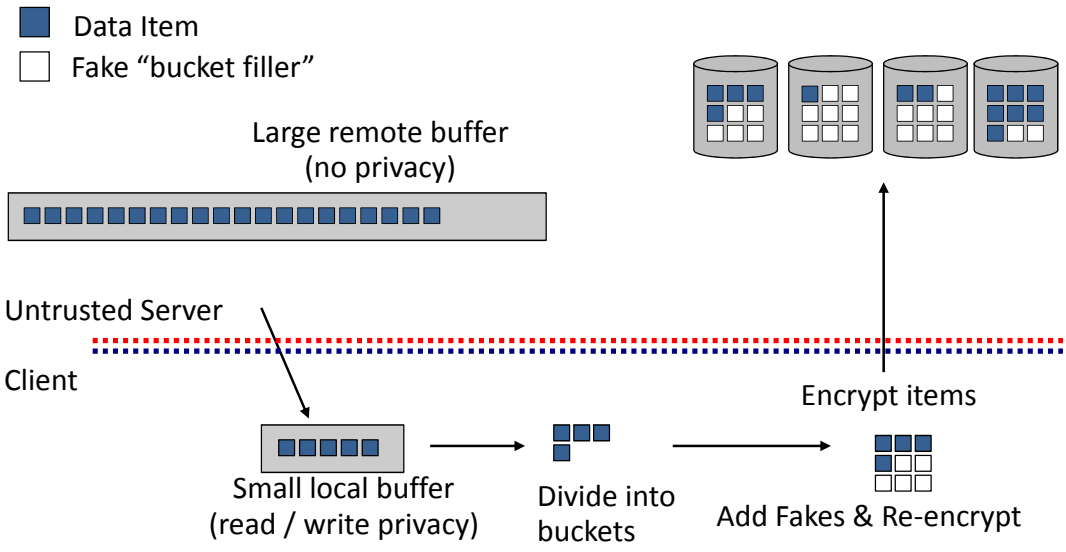

Figure 4.3: Reshuffle Phase 3: Add fakes. Analogously to Phase 1, the client adds fake items in a single pass. This makes the buckets equivalently sized, despite collisions in the hash function that determines item locations.

block is found, this can potentially become a privacy leak. Hence, we need to ensure that throughout the whole program execution, the need for restarting a level re-order happens with sufficiently low probability. If the probability of bucket-overflow for one level-reorder step is at most p', then the union bound yields the probability of level re-order restart throughout the whole execution of an ORAM program (query sequence) to be at most $p' \cdot k$, for k being the number of level re-order steps. Since the i-th level is re-ordered every 4^i steps, where $i > 0$, for a query sequence of length K, there are $k \leq \sum_{i=1}^{\infty} \frac{K}{4^i} = \frac{K}{3}$ level re-order steps. Hence, the probability of bucket-overflow in any step for a query sequence of length K is at most $Kp'/3$.

As in Phase 1, we will employ a local buffer of size $\psi(n)$ to prevent the server from learning where fakes are being added. The client scans the array of real blocks stored in the remote server by Phase 2 into a local queue. Once the local queue is half full, it begins constructing server-side buckets with the blocks from the queue, writing into one bucket for every block read. As long as the temporary queue does not overflow or become empty, the exact pattern of reads and writes observed by the server is dependent only on the number of blocks. Therefore, the server learns nothing of which are the fake blocks by observing this process. Moreover, it does not reveal the number of blocks in each bucket, since the buckets are written sequentially to the server in full, so the read and write pattern for this step is identical on every repetition.

The algorithm runs in time linear to the size of the level being written. For level i, which contains up to 4^i buckets of size $\lambda \log n$, the running time is $O(4^i \log n)$. We now establish correctness: showing that the assumed amount of client storage suffices.

Theorem 15. *With high probability, the Add Fakes algorithm queue never overflows or empties early.*

Proof. The result is a random distribution of fake and real items. This distribution encodes a zero-sum asymmetric random walk, as in the Remove Fakes algorithm. Every iteration adds one item to a bucket. On iterations in which there is a real item to add (on average once every r iterations), the queue size decreases by 1. When we read from A, once every r iterations, the queue size increases by 1. As in the analysis of the RemoveFakes algorithm, we instead consider the queue size to increase by $1/r$ every iteration, giving us 4^i step-downs of size $1 - 1/r$ and $(r-1)4^i$ step-ups of size $1/r$. The remainder of this proof is an exact parallel of Theorem 14, with the step sizes inverted. □

Theorem 16. ***Correctness.*** *After Phase 3, all blocks will be in the correct bucket (determined by the secure hash function).*

Proof. This proof follows from the construction of Phases 2 and 3. Phase 3 correctly builds the buckets for level i when its input array satisfies the follow properties: (1) all data blocks corresponding to i are in the array. (2) For all data blocks b, b', if the bucket corresponding to data block b precedes the bucket corresponding to data block b', then b is listed in the array before b'. After the sort in Phase 2, all blocks are in sorted order, according to their bucket, therefore meeting the two requirements for the input to Phase 3. □

Theorem 17. ***Privacy.*** *The contents of the level make it from the old permutation to the new permutation without revealing any non-negligible information about either permutation. The location of the fake blocks is not revealed.*

Proof. Theorem 3 shows that the level permutation performed in Phase 2 does not reveal any correlation between the old locations and the new locations. Furthermore, the read and write pattern of Phase 3 is independent of the data items and the final permutation, so Phase 3 does not reveal anything about the location of the fake blocks, or the permutation. □

4.2 Discussion and Extensions

Handling Duplicates. While for simplicity, I did not discuss this aspect, the following scenario can ensue and requires handling. Read blocks are added back to the top of the ORAM, resulting in potentially conflicting versions that need to be reconciled. To prevent this, we perform an extra step in the online query phase: scanning a bucket entails downloading the bucket to client memory, removing the item we were looking for and replacing it with a fake item, re-encrypting, and writing it back. This results in doubling the network traffic in the online query phase: every bucket read is accompanied by a write. This is reflected in Figure 4.7.

Goldreich and Ostrovsky's ORAM handles this scenario by reconciling multiple versions of an item in the reshuffle phase; as an alternative to the above online-phase mechanism we could adopt a similar approach. This will improve the overhead of the online phase in

exchange for extra work in the reshuffle. To achieve this, we could add an extra pass during the reshuffle to remove duplicates after Phase 2 (Oblivious Merge Sort), then repeat Phase 2. Recall that after the Phase 2, the blocks are sorted by the buckets they will end up in. This single pass reads blocks, re-encrypts them, replaces them with fakes if they are duplicates, and writes them back, ensuring there are no duplicates during the second iteration of Phase 2. This step can be performed without affecting the asymptotic complexities since the communication, computational, and memory requirements, are less than the requirements of the other phases of the shuffle.

Asymptotic Considerations. To maintain indistinguishability between reads and inserts, the database size increases on every access. Of course, this property may be undesirable for some users; if so, it is simple to consolidate the duplicates periodically, at the expense of revealing the ratio of reads to inserts over that period.

This growth affects computation of the asymptotic complexity. Indeed, for simplicity reasons, the above complexity is an overestimation of the actual cost seen in an implementation. This is so because earlier queries operate over a smaller n. Nevertheless, in computing the complexity, we consider the current, larger n. Overall, this should not significantly diverge from the actual seen amortized cost since the cost of recent queries dominates. To see why this is the case, consider that every $4n$ accesses a new level is created, resulting in the size of the database effectively quadrupling in size. Thus, for any considered period ending in the creation of a new level, we are overestimating at most for a quarter of the queries.

Implementation: Required Memory. Finally, for a fixed size of the database, larger blocks require more memory on the client side to achieve the same privacy guarantees. For a database of size N bytes, with $n = N/B$ blocks of size B, $B\sqrt{n \log n} = \sqrt{NB(\log N - \log B)}$ memory is needed. Hence, the amount of memory scales proportionally to $\sqrt{B}$.

As can be seen in the pseudo-code for Procedure AddFakesToLevel, in the actual implementation, we use slightly more than $\psi(n)$ storage. In particular, we need room for a small amount of additional memory, e.g., to store temporary data. In the provided pseudo-code this amounts to approximately one more bucket (sized $\lambda \log n$).

4.3 Performance

In evaluating the feasibility and performance of the architecture we consider the sample configuration illustrated in Figure 4.4. Further, Figures 4.5 and 4.6 illustrates multiple such data points. We set the bucket size to be $\lambda \log n = e \log 10^9$. Using the formula from Section 4.1.6, we obtain the probability of bucket overflow to be at most 2^{-100}.

Online Cost. The query requires online scans of one bucket at each level, plus a write to the top level. The scan of the $\log_4 n$ levels are interactive; the bucket scanned at each level depends on the results of the previous level. Figure 4.7 displays the expected online cost.

It is clear from these estimates that in a sequential access model, the network latency is responsible for most of the query latency. This is due to the interactive nature of the scans; the client cannot determine the next bucket until it has seen the contents of the previous.

	Server	Client
RAM	4 GB	1 GB
Processor		2 GHz
Disk seek time	5 ms	
Sustained disk read/write	50 MB/s	
Link bandwidth		10 MB/s
Link round trip time		50 ms
En/Decryption		100 MB/s (AES [77])
Outsourced data set size	1 TB, in 1000-byte blocks; $n = 10^9$	

Figure 4.4: Configuration used to compute sample values in the following tables and graphs.

Offline Reorder Cost. The offline cost resulting from the reordering of level i (performed once every 4^{i-1} accesses) consists of three phases including a sequential level scan of size $4^i \lambda \log n$, and a sequential write-back of size 4^i to remove fakes (Phase 1). The oblivious sort (Phase 2) consists of $\log_2 4^i$ sequential scans of size 4^i. Adding fakes (Phase 3) requires requires copying back $4^i \lambda \log n$ items. To estimate this cost we must sum over all $\log_4 n$ levels, recalling that each level is reordered only once every 4^{i-1} queries. We therefore remove a factor of 4^{i-1}, and sum over all levels, to calculate the amortized overhead. Figure 4.8 shows the resulting formulas. If all of these costs are incurred sequentially, we have an amortized response rate of approximately 1.5 sec/query offline plus 1.5 sec/query online, for 3 sec/query.

The bottleneck when determining the parallel query throughput is the network throughput, at 842 ms/query. Thus, when making simultaneous use of multiple resources by running queries in parallel, this results in a query throughput of over a query per second.

By comparison, in ORAM, the network transfer time alone for reshuffling level i consists of about 10 sorts of $4^i \log n$ data, each sort requiring $4^i \log(n) \log^2(4^i \log n)$ block transfers, for a total of $10 \cdot 4^i \cdot \log(n) \cdot \log^2(4^i \log n) \cdot 2^{10}/10MB/sec$. Summing over the $\log_4 n$ levels, and amortizing each level over 4^{i-1} queries, ORAM has an amortized network traffic cost per query of $\sum_{i=1}^{15} 10 \cdot 4 \cdot 15 \cdot \log^2(15 \cdot 4^i) \cdot 2^{10}B = 614KB \sum_{i=1}^{15} (\log 15 + \log 4^i)^2 \approx 3.680GB$. Over the sample 10 MB/s link this is a 368 sec/query amortized transfer time, almost three orders of magnitude slower.

4.4 Conclusions

I introduced a (first) practical oblivious RAM mechanism, orders of magnitude faster than existing mechanisms. I have analyzed its overheads and security properties. I validated its practicality by exploring achievable throughputs on current off-the-shelf hardware.

Future chapters will increase achievable throughputs an de-amortize the offline level reorder cost. Moreover, as the bulk of the overhead in this technique is related to the fake blocks, I next present an alternate construction that hides which level is accessed for a particular query, bringing the amortized overhead to $O(\log n \log^2 \log n)$ per query.

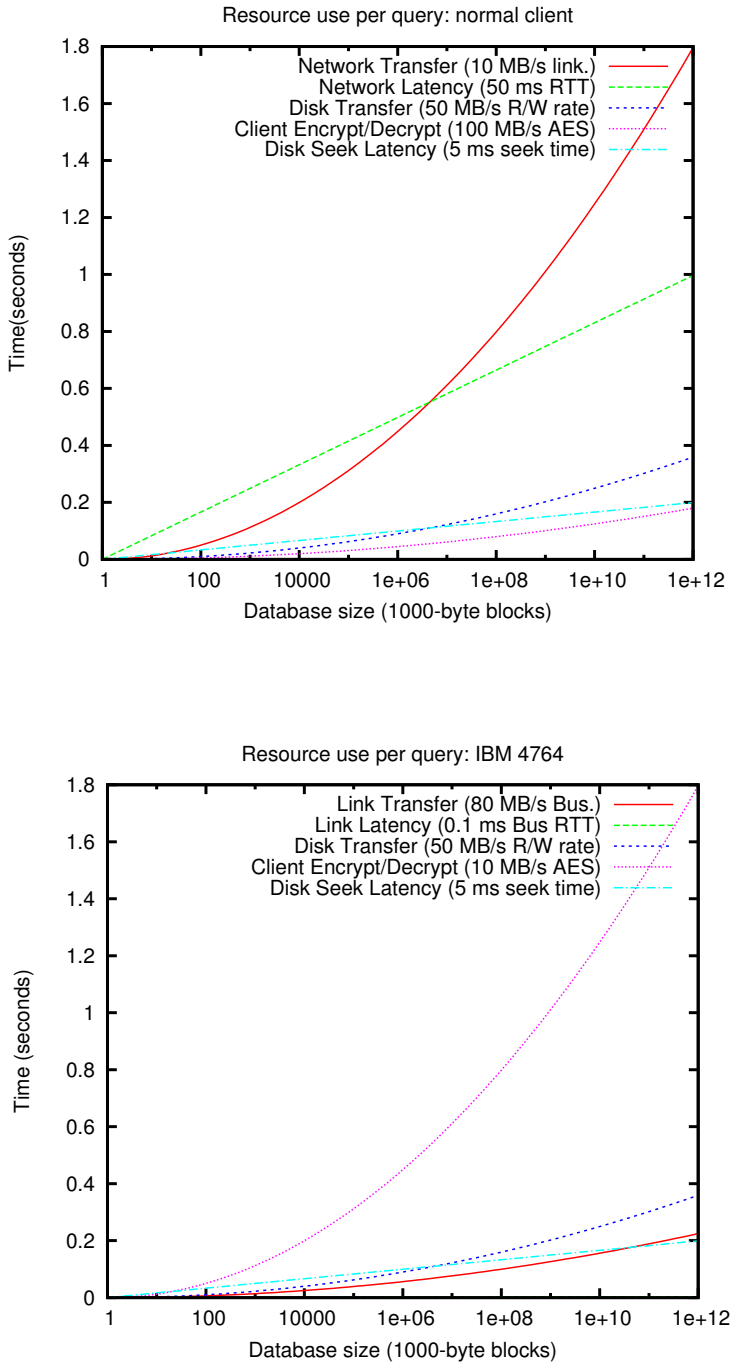

Figure 4.5: Amortized use of resources per query (top) for a normal client and (bottom) using a IBM 4764 SCPU as a client.

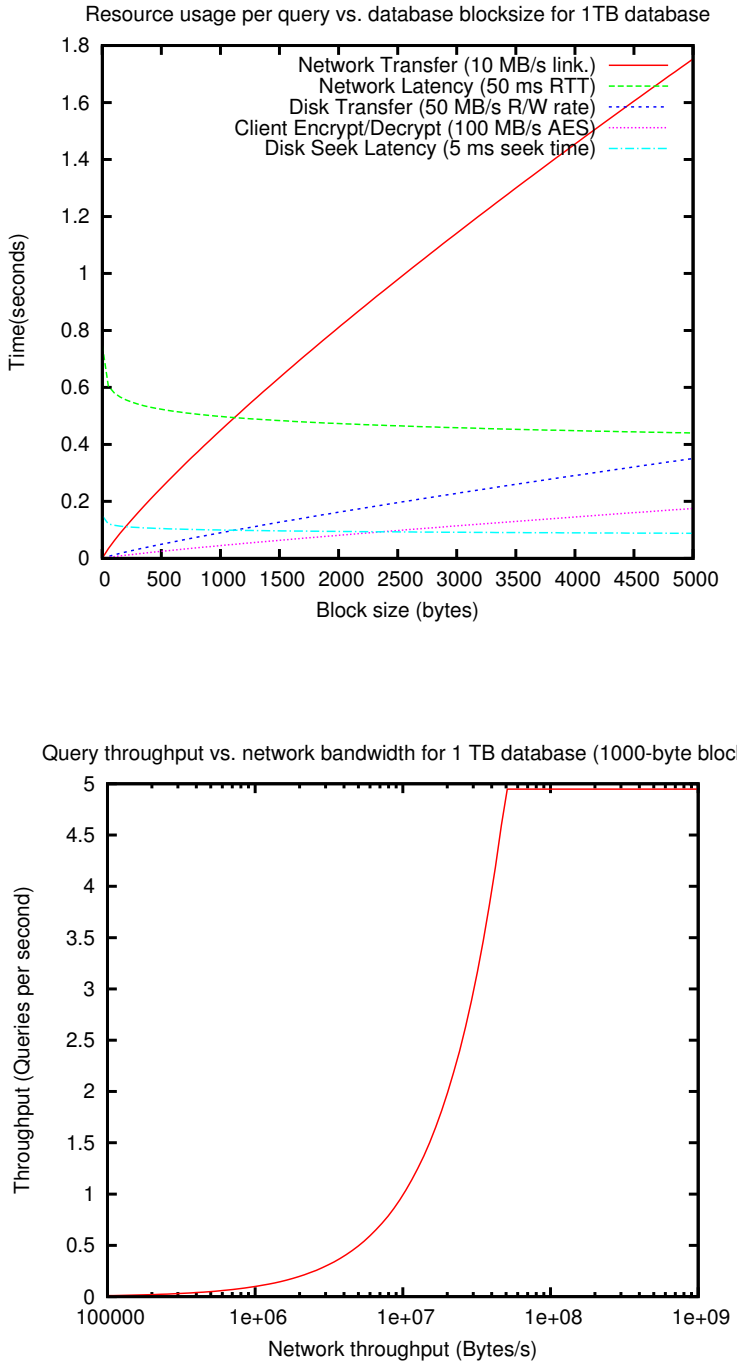

Figure 4.6: Resource usage as the the database block size is varied (top), and (bottom) estimated query throughput as the the network link capacity is varied. A fixed database size of 1TB is assumed.

	Formula	Sample
Network latency	$RTT_{link} \cdot \log_4(n)$	750 ms
Disk seek	$Latency_{seek} \cdot \log_4(n) \cdot 2$	150 ms
Network transfer	$\log_4(n) \cdot \lambda \log(n) \cdot 2 \cdot \text{blocksize}/\text{Throughput}_{\text{link}}$	168 ms
Client en/decryption	$\log_4(n) \cdot \lambda \log(n) \cdot 2 \cdot \text{blocksize}/\text{Throughput}_{\text{crypto}}$	17 ms
Server disk transfer time	$\log_4(n) \cdot \lambda \log(n) \cdot 2 \cdot \text{blocksize}/\text{Throughput}_{\text{disk}}$	34 ms

Figure 4.7: **Online** cost per query, resulting from scanning a bucket at each level.

	Formula	Sample	
Network latency.	n/a	< 1 ms	Level reordering is not interactive, so idling can be avoided here.
Network transfer	$\frac{\log_4(n) \cdot \lambda \log(n) \cdot 10 \cdot \text{blocksize}}{\text{Throughput}_{\text{link}}}$	842 ms	The Phase 1 and 3 scans account for its bulk.
Disk seek latency	n/a	< 1 ms	Seek time will be hidden by disk transfer during reordering.
Disk transfer	$\frac{\log_4(n) \cdot \lambda \log(n) \cdot 10 \cdot \text{blocksize}}{\text{Throughput}_{\text{disk}}}$	168 ms	This load can be split among several disks.
Client processing	$\frac{\log_4(n) \cdot \lambda \log(n) \cdot 10 \cdot \text{blocksize}}{\text{Throughput}_{\text{crypto}}}$	84 ms	

Figure 4.8: Amortized **offline** cost per query.

Chapter 5

A Fresh Approach: Revamping the Query Process

I now build on my results in Chapters 3 and 4 to introduce an efficient ORAM protocol with significantly reduced communication and computation complexities. My protocol still uses the ORAM-based [41] pyramid-shaped database layout and reshuffling schedule employed in the last chapter. This yielded a protocol with a communication complexity of $O(\log^2 n)$ in the presence of $O(\sqrt{n \log n})$ client working memory, for a database sized n. Here, however, we shall deploy a new construction and more sophisticated reshuffling protocol, to reduce both the communication complexity (to $O((\log n)(\log^2 \log n))$) and the server storage overheads (to $O(n)$)—yielding a comparatively fast and *practical* oblivious data access protocol. Moreover, we restore the correctness guarantees that were suspended in the last chapter, and ensure privacy even under an actively malicious service provider. I describe a practical system that can execute an unprecedented *several queries per second on terabyte-plus databases* while maintaining *full computational privacy* and *correctness*.

Efficiency. One of the main intuitions here is to store each pyramid level as an encrypted hashtable and an encrypted Bloom filter (indexing elements in the hashtable). The Bloom filter allows the client to privately and efficiently—without scanning of $O(\log n)$ fake block buckets for each stored block to hide the success of each level query as in previous ORAMs—identify the level where an item of interest is stored, which is then retrieved from the corresponding hashtable. Less server-side storage is required ($O(n)$ instead of $O(n \log n)$), thus both increasing throughput and reducing required server-side storage by an order of magnitude.

Privacy. The approach guarantees client access pattern privacy, since the same operations are performed at all pyramid levels, in the same sequence for any item of interest. The use of the encrypted Bloom filters allows the client to query an item directly at each level without revealing the success, instead of relying on a series of $O(\log n)$ fake blocks for each stored block to hide the success of each level query. My contributions consist also of a new reshuffling algorithm that obliviously builds and maintains the encrypted Bloom filters and of a more efficient oblivious merge-and-scramble.

Correctness. Moreover, authenticated per-level integrity constructs provide clients with *correctness* assurances at little additional cost, specifically ensuring that illicit server behavior (e.g., alterations) does not go undetected.

5.1 A Solution

In Chapter 4 we achieved a complexity of $O(\log^2 n)$ in a protocol offering *access privacy* but *no correctness* assurances. Here I build on that result by deploying a new construction and more sophisticated reshuffling protocol, to reduce communication complexity to only $O((\log n)(\log^2 \log n))$ (amortized per-query), under the same assumption of $c\sqrt{n \log n}$ temporary client storage, while reducing the required server storage from $O(n \log n)$ to $O(n)$, and endowing the protocol with correctness assurance.

5.1.1 Overview

Similar to ORAM (see Section 2.1 for more details), data is organized into $\log(n)$ levels, pyramid-like. Level i consists of up to 4^i items, stored on the server as label-value pairs. These pairs can be stored and retrieved in $O(1)$ time if the storage provider implements a suitable hash table (such as a constant time, constant space hash [91]). This differs from ORAM, which stores an item at level i using a keyed hash function to determine its storage bucket (of size $O(\log n)$, to allow for hash collisions) within the level. The use of *fixed-sized hash buckets in ORAM instead of a simple hash table adds a $O(\log n)$ storage overhead multiplier, and slows down query processing*, but the buckets are necessary; otherwise queries to a hash table could reveal whether the item was found at this level.

Here *we will avoid the overhead of using buckets* to mask the query result by using Bloom filters [12] (constructed to be collision-free). Before attempting to query for an item that might not be at the current level, a per-level Bloom filter is queried first. The bits of the Bloom filter are encrypted, hiding the result of the query. If the Bloom filter indicates that the item is *not* at this level, we query the level for a unique fake item instead and continue with the next level. Once we eventually find the desired item (at a future level)—it will be moved into the top pyramid level—above the levels where it was searched for before (as in ORAM). This ensures that the same item will never be queried for in that instantiation of the Bloom filter again (as now it will be found higher in the pyramid, or a reshuffle would have been triggered).

Insight One: Faster Lookup. Thus one key insight in this mechanism is that we can construct an encrypted Bloom filter to perform set membership tests, without revealing the success of our query. Additionally, I design a novel construction procedure that assembles the encrypted Bloom filter without revealing any correlation between scanned items and associated Bloom filter positions. The final benefit of using encrypted Bloom filters is that all unique queries are computationally indistinguishable due to the nature of the keyed hash function used to index the filter. This allows us to modify ORAM with significant

performance benefits, since we can avoid handling hash collisions, which add a $\log n$ factor in total database size as described above.

The notion of encrypting a Bloom filter has been studied previously [9]. However, here we use a novel construction that hides the construction process, the inputs, and the results of the Bloom filter.

Insight Two: Correctness. Moreover, we deploy a set of authenticated, per-level integrity constructs to provide clients with *correctness* assurances at minimal additional cost. We specifically ensure that illicit server behavior (e.g., alterations) does not go undetected.

I now detail these components.

5.1.2 Bloom Filters

Bloom filters [12] offer a compact representation of a set of data items. They allow for relatively fast set inclusion tests. Bloom filters are *one-way*, in that, the "contained" set items cannot be enumerated easily (unless they are drawn from a finite, small space). Succinctly, a Bloom filter can be viewed as a string of b bits, initially all set to 0. To *insert* a certain element x, the filter sets to 1 the bit values at index positions $H_1(x), H_2(x), \ldots, H_h(x)$, where $H_1, H_2, \ldots, H_h$ are a set of h crypto-hashes. Testing set inclusion for a value y is done by checking that the bits for *all* bit positions $H_1(y), H_2(y), \ldots, H_h(y)$ are set.

By construction, Bloom filters feature a controllable rate of false positives r for set inclusion tests—this rate depends on the input data set size z, the size of the filter b and the number of cryptographic hash functions h deployed in its construction: $r = \left(1 - (1 - 1/b)^{hz}\right)^h$.

Encrypted Bloom Filters. An Encrypted Bloom filter, once we construct it obliviously, will allow us to quickly an obliviously query check any level for the presence of the desired item. We shall make a set of four modifications to turn a Bloom filter into a remote data structure that we can query privately. First, we use keyed hash functions in place of public hash functions. Second, we store their bit representation encrypted while still allowing client-driven Bloom filter lookups. This is done by dividing the Bloom filter into small chunks, encrypting each chunk individually, and attaching a message authentication code (MAC) for integrity. Third, we use an oblivious construction process that guarantees that the construction transcript appears independent of the contents and queries. Fourth, we sacrifice the constant time query cost in exchange for a guarantee that the possibility of a false positive is negligible.

5.1.3 Query Processing

A query consists of a read or write request for a data item. These items are kept at the storage provider at a particular level; part of the client's job is to determine which level the item is at without revealing this to the server. Algorithm `15` shows the pseudo-code of this operation; it is also illustrated in Figure 5.1.

To process a query, the client first downloads and scans the server-stored item cache (line 1) then proceeds to search each level, starting at the top (line 2). A labeling function

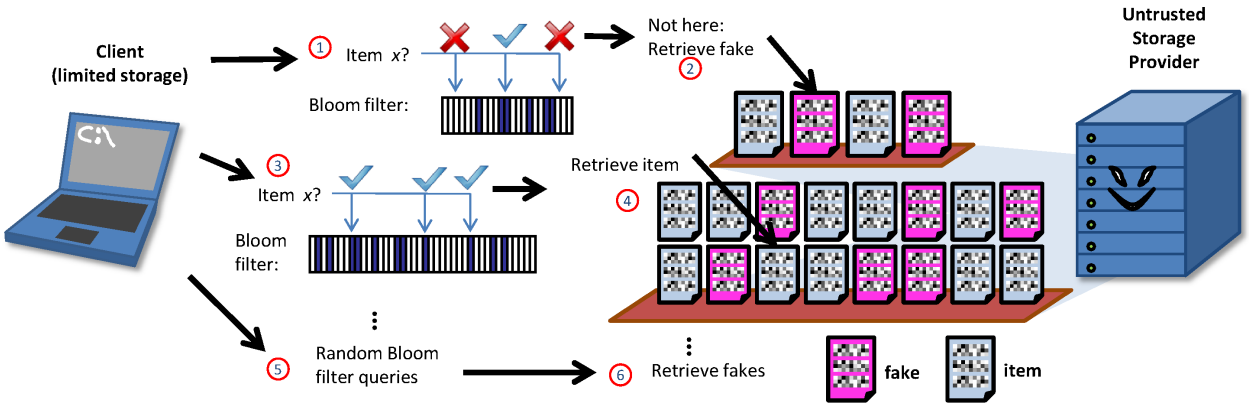

Figure 5.1: Sample query. The client first checks the level 1 Bloom filter (1) to see whether the item is at Level 1. Finding at least one encrypted bit set to *0*, the client knows it is not. The client then asks for a fake item (2) at Level 1. This fake item was previously placed here for just such a scenario; the server does not know that it is fake. The client then queries the Level 2 Bloom filter (3). Seeing all encrypted bits are set to *1*, the client knows the item is stored at Level 2. It requests the item (4); again, the server does not know whether it is returning a fake or real item. On subsequent levels (5), the client now issues random Bloom filter queries and requests for fake items, to prevent the server from learning that the item was found at Level 2. Not illustrated in this figure is the final step, which places the discovered item back into the top level, and possibly initiates a reconstruction.

consisting of a hash of the item ID with several level parameters ($fakeAccessCtr(level)$ and $Gen(level)$) generates the unique label by which the client can find the item at a particular level, if the item is indeed there. $fakeAccessCtr(level)$ represents the number of accesses to $level$ since the last reshuffle, and $Gen(level)$ represents the number of times $level$ has been reshuffled. The use of $fakeAccessCtr$ (line 3) ensures that successive queries request unique fake items, which the client knows are stored on the server. The use of $Gen(level)$ ensures that items on every subsequent reshuffle of a level have unique labels. Both functions are computed from the total number of accesses to the system thus far.

The search of any one level requires $O(\log \log n)$ time since the number of hash functions to request h is $O(\log \log n)$. If at the ith level the item has not already been found (in the cache or a previous level) (line 5) the client first computes the positions where the item would be stored in the ith level Bloom filter (line 6). It then retrieves those encrypted bits from the server-stored Bloom filter (line 7). If the decrypted bits are all *1* (line 8), the client has found the item at the ith level. It then computes the label under which the item is stored in the level (line 9) and asks the server to remove and return the corresponding item (line 10). If at least one decrypted Bloom filter bit is *0* (line 11), the client instead performs the same operation using a *fake* label (built at line 4), known to be stored at the server.

Once the client has found the item (line 12), it proceeds by seeking fake items on the subsequent levels. This avoids revealing which level answered the query, which would provide a correlation between queries. The client first searches a random set of positions (line 13) in

the Bloom filter at the ith level (line 14), then asks the server to retrieve and remove a fake item from the level (line 15).

Note that the client queries the remotely-stored encrypted Bloom filter by requesting the encrypted values of positions indicated by the label function. While this reveals the requested Bloom filter positions to the remote server, *nothing is lost* as we prevent correlation by guaranteeing that any Bloom filter is only ever queried for any particular item once. Since the positions in the filter are each encrypted, the server never learns the result of the Bloom filter query.

5.1.4 Access Privacy

The client achieves access pattern privacy by maintaining two conditions. First, *no item is ever queried twice using the same label.* This is achieved by removing the item, once it is found, and placing it in the item cache. Thus, on future queries the client will locate it in the item cache before repeating a label request; fake requests will be substituted on the lower levels. As items propagate out of the item cache (described in Section 5.1.5), the label functions are updated, so that the item has a different label by the time it makes it back down to a particular level.

Second, *the access patterns must appear indistinguishable* from random no matter where the item is located. A set of fake items is used to guarantee this: if the Bloom filter returns negative, indicating that the item is not stored at this level, a fake item from this level is retrieved instead.

On every single query, the server observes the same pattern. The client first scans the item cache, then queries a random value (chosen uniformly randomly, independently from all other information available to the server) from the level 1 encrypted Bloom filter—never queried by the client before. The server can observe the position of bits in the Bloom filter accessed, but it cannot observe whether each bit is set to *1* or *0*. The server then observes the client retrieve and delete one item from the level 1 hash table—never retrieved by the client before. This identical pattern of a random Bloom filter lookup followed by a random label-value retrieval and deletion continues through each level. Finally, the client appends a (semantically secure) encrypted value to the item cache.

Success or failure at each level is not revealed—the server cannot distinguish queries to fake entries in the Bloom filter from queries to real items in the filter, and the server cannot distinguish either of those from real items that are not in the filter. Additionally, the server cannot distinguish requests to real items from requests to fake items from the hash table, since the secure hash function used is non-invertible.

In addition to the constant amount of data transfer and computation exercised on each level by every query, we will see that $O(\log \log n)$ bits are examined to perform a Bloom filter lookup. Since there are $\log_4 n$ levels, the online cost per query is $O(\log n \log \log n)$ (measured in computation or transfer of words). Level reshuffling, described in the next section, will bring the total amortized cost per query of $O((\log n)(\log^2 \log n))$.

I now fill in the missing pieces: how to empty the item cache when it becomes full, and how to build the levels and the Bloom filters obliviously.

```
    server: Server                                          /* Server stub */
    bits: int[]                    /* encrypted bit values of Bloom filter */
    label, fakeLabel: int[]                                 /* search labels */
    bfpositions: int[]                                      /* search labels */
    fakeAccCtr: int[]                          /* per level access counter */
    found: boolean
    K: byte[]                                               /* secret key */
    h: int ← c log log n          /* number of Bloom filter hash functions */
    k: int                         /* Bloom filter configuration parameter */
    v: Object                                             /* value for name x */
    L: Object[] ← server.itemcache                      /* scan of item cache */
    m_i: int                                              /* Size of level i */
 1  found, v ← L.search(x)                    /* Check recent queries first */
 2  for i ← 1 to log_4 n do
 3      fakeAccCtr[i]++
 4      fakeLabel ← hash(i||'fake'||Gen(i)||fakeAccCtr[i]||K)
 5      if found = false then
            for j ← 1 to h do
 6              bfpositions[j] ← hash(i||j||'BF'||Gen(i)||x||K)   mod k · h · m_i
            end
 7          bits ← server.getBloomFilter(i,bfpositions)
 8          if decrypt(bits) = "11..1" then
 9              label ← hash(i||'data'||Gen(i)||x||K)
10              v ← server.getAndRemove(label)
                found ← true
            else
11              server.getAndRemove(fakeLabel)
            end
        else
12          for j ← 1 to h do
13              bfpositions[j] ← random   mod k · h · m_i
            end
14          server.getBloomFilter(i,bfpositions)
15          server.getAndRemove(fakeLabel)
        end
    end
    itemCache.append(x,v)
    return (v)
```

Procedure Query(x)

5.1.5 Handling Level Overflows: Reshuffle

The construction of the initial database structure is explained by the process of emptying the item cache: items are inserted into the item cache, which then overflows into the lower levels. Similar to ORAM, when the item cache is emptied, the contents are poured into level 1. In that process, level 1 and its new contents are reshuffled according to new label functions, removing any correlations between past and future lookups. When level 1 becomes full, it is poured into level 2, and so forth. Thus, the reshuffle process empties one level $i-1$ into the level i below it, which is four times as large as level $i-1$. Level i is then scrambled, hiding the correlation with the items' previous levels. A new Bloom filter for the lower level is constructed—even items which happened to be at level i anytime in the past are now identified by a new unique label.

Let m be the size of the new level ($m \leq 4^i$). Let $h = O(\log \log m)$ be the number of hash functions used to generate a Bloom filter. The number of bits in the Bloom filter BF at level i will be equal to a constant k, times the number of hash function h and the number of items in the Bloom filter m. This ensures that any single bit in the Bloom filter is set with probability less than $1/k$. The database size n grows by one on each query, and thus varies over the lifetime of this level; h is chosen based on the value of $2n$ at the time of level construction, since n will never double during this period. The constant k is a configuration parameter balancing a trade-off between the number of Bloom filter hash functions and the Bloom filter size to meet the configured acceptable failure rate (e.g. 2^{-128}).

The client secret key is K. Let W be a working set stored on the server. Let L be a list of $O(m)$ entries, stored on the server. Let T be a $\sqrt{m}$ integer array stored at the client. Let Bkt be a server-hosted list of $O(\sqrt{m})$ buckets, of $\sqrt{m}$ entries each. Initially, W, L, T and Bkt are empty and all the bits in the server-stored BF are set to *0*.

The overflow process, performed by the client to pour level $i-1$ (or the item cache) into level i proceeds as follows (see Figure 5.2 for an illustration).

After merging the levels (step 1), steps 2 and 3 place the items in the new level while eliminating any correlation between the old and the new level structures. Costs are expressed in terms of $m = 4^i$, the size of the current level.

Finally, we create a list of what positions must be set in the Bloom filter to add each item, and store this list encrypted on the server (step 4). This will be used to construct the Bloom filter in step 5.

1. **Merge levels.** Move all items from level $i-1$ and level i into W, a working buffer on the remote server. Discard the Bloom filters attached to both levels.

2. **Scramble the items.** Finally, the client uses the Oblivious Merge Scramble Algorithm to scramble the actual items in the working buffer W. The Oblivious Merge Scramble requires $O(m \log \log m)$ time and $O(\sqrt{m \log m})$ private storage.

3. **Add items back to level i.** Once scrambled, the items inserted under their new labels, according to the new labeling function for level i. For each item in $x \in W$ let $label(x) = hash(i||\text{`}data\text{'}||Gen(L_i)||x.id||K)$. Insert the pair $(x, label(x))$ into the set

of items stored at level i. Add m fake items to level i, so that a query that turns out not to be for this level will have an item to retrieve instead (most of these m fakes will be deleted by the query process before the next reshuffle).

4. **Build a list representation of the new Bloom filter.** Increment $Gen(i)$: the generation of level i has increased. Read each item $x \in W$ exactly once, and for each compute its $h = O(\log \log n)$ positions of bits $p_j(x) = hash(i||j||'BF'||Gen(i)||x.id||K) \mod k \cdot h \cdot m$, for $j = 1..h$, in the new Bloom filter. Encrypt each $p_j(x)$ separately and store all $E(p_j(x))$ values on the server-side list L. This step takes $O(m \log \log n)$ time, and $O(1)$ private (client-side) storage.

5. **Obliviously construct the Bloom filter.** Options for this step are described in the following sections. Section 5.1.6 provides a solution requiring $O(m \log n)$ time and either $O(\log n)$ client memory, or $O(c\sqrt{m \log m})$ client memory, depending on whether a randomized shell sort or Oblivious Merge Sort is employed. Section 5.1.7 provides a solution requiring $O(m \log^2 \log n)$ time and $O(c\sqrt{m \log m})$ client memory, employing only the Oblivious Merge *Scramble*.

Level $i-1$ is now empty, and level i now contains all the items that were in level $i-1$. If level i is now full, this is then repeated as level i is then dumped into level $i+1$; the process is repeated recursively.

This procedure shows how level $i-1$ can be dumped into level i at a cost of $O(m \log^2 \log m)$ $= O(4^i \log^2 i)$. Level $i-1$ is emptied once every 4^{i-1} queries, thus resulting in an amortized cost per query due to reshuffling of

$$\sum_{i=0}^{\log_4 n} O(\frac{4^i \log^2 i}{4^{i-1}}) = \sum_{i=0}^{\log_4 n} O(\log^2 i) = O((\log n)(\log^2 \log n))$$

As described below, client memory sized $c\sqrt{m \log m}$ is required for this procedure. The most stringent requirement is for the construction of the bottom level, when $m = n$. This case requires $c\sqrt{n \log n}$ memory.

The positions of the Bloom filter bits retrieved to check an item will appear to be chosen uniformly random, and completely independently of each other; therefore, any bit pattern indicates nothing about the query or the success of the query. They are independent of the bucket sort write pattern, which is the only other piece that could be tied to it, since the bucket sort write pattern is the only access pattern that varies during the reshuffle. The bucket writes are all identical except for the order of the writes, which is uniform random because of the scramble. The scramble has no bearing on the Bloom filter access pattern, which is dependent only on the query and the current Bloom hash function. Therefore the Bloom filter construction process yields no information about the items to the server in the resulting Bloom filter.

The level reorder process results in a new level that has no correlation to the old level, since the new permutation is chosen uniformly randomly (Theorem 8). The scramble process

itself reveals no information about the new or old permutations, since the scramble has the same access pattern in all instantiations.

5.1.6 Oblivious Bloom Filter Construction (Simpler)

The idea is to use an oblivious sort that puts the indices into the positions corresponding to segments. However, any sort requires some extra computation (at least $O(m \log m)$ computation to sort m items instead of scrambling them). Such an oblivious sort can be obtained by replacing the client random array selection described in Section 3.3 with a comparison that returns the smallest of the items in the fronts of the queue buffers. This does not affect the communication complexity, but it does slightly raise the mechanism's overall amortized *computation* complexity.

5a **Add segment identifiers.** In a single pass over the encrypted items of the level undergoing construction, build an encrypted, server-stored, list of the positions that need to be set in the encrypted BF. Divide the BF into segments of size $O(\log n)$. Append one encrypted segment identifier for each possible segment to this list, stored in such a way that the server cannot distinguish segment identifiers from BF positions.

5b **Sort the list by segment.** Obliviously sort this entire list with segments distributed among the positions, directly following the positions they belong in.

5c **Set the right positions.** In a single pass over this sorted list, output one BF segment for every list element. For each segment identifier, this is an encrypted segment, with the appropriate positions set. Recall that these positions immediately preceded this segment identifier in the sorted list. For each position encountered in this list, this is a dummy (empty) segment.

5d **Move the dummies to the end.** Obliviously sort the resulting list by segment identifier, with the dummy segments at the end.

5e **Remove the dummies.** Truncate off the dummy segments from this list. The result is the encrypted BF.

The advantage of this approach over the one described in Section 5.1.7 is that using a logarithmic-space $O(m \log m)$ oblivious sort [46] provides a logarithmic-space construction still running in $O(\log^2 n)$ time. As will be discussed in Chapter 6, because $\log_2 n$ levels must be shuffled simultaneously, employing this as a de-amortized ORAM increases the client storage requirement to $O(\log^2 n)$.

This construction process is secure, provided (a) the oblivious sort is private, (b) the server cannot distinguish an encrypted dummy segment from an encrypted BF segment, and (c) the server cannot distinguish an encrypted segment identifier from an encrypted BF position. If these three conditions are satisfied, then every run of this process over a set of m items appears identical to the server, regardless of the positions in the BF being set.

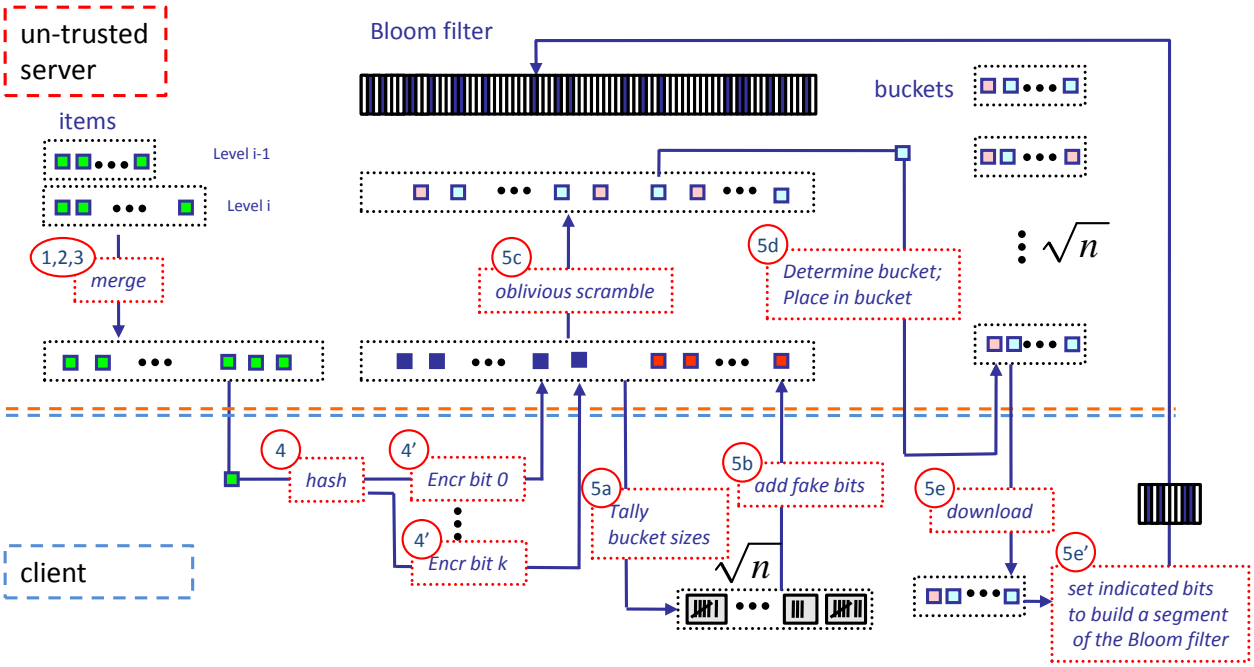

Figure 5.2: Level reshuffle: Bloom filter construction, Improved Asymptotics

5.1.7 Oblivious Bloom Filter Construction, Improved Asymptotics

I now show how to construct the Bloom filter in $O(m \log^2 \log m)$ time. In the next steps (illustrated in Figure 5.2) we build the encrypted Bloom filter without revealing the positions set in the Bloom filter.

To turn the list into a proper Bloom filter bit array, it will be sorted with a bucket sort—with $\sqrt{m}$ buckets of size $\sqrt{m}$, so that any bucket fits in private storage. To keep the buckets indistinguishable, we ensure they will all have the same size. In step 1, we calculate the size of each bucket, by scanning the list of Bloom filter positions, incrementing the appropriate bucket size tally for each position. In step 2 we add fake positions that will end up in those buckets that are lacking, according to the above (step 1) tally. Each bucket corresponds to a fixed range of positions in the final filter, so $j\sqrt{m}$ is used as a position that will wind up in bucket j. At this point the server will be able to identify the fake positions, since they are all at the end of the list. We then (step 3) scramble the list of positions to destroy all correlation between items and positions, and hide the fakes. In step 4 we move the scrambled positions into their buckets. Step 5 constructs the final Bloom filter, building a piece from each bucket.

5a **Tally Bloom filter positions to determine future bucket sizes.** Read each entry of L (the list of encrypted Bloom filter positions prepared in the previous step) exactly once. At the client side, for each entry $E(p) \in L$, let $idx(p)$ be the $\log m/2$ most significant bits of p. Then, increment $T[idx(p)]$. This step allows the client to compute the number of bit-positions of L that will later (Step 4) end up in the Bkt structure.

Effectively, the client builds a tally in local storage calculating the future size of each of the $\sqrt{m}$ buckets that will be built on the server in the step 4 bucket sort. We use $\sqrt{m}$ buckets of size $\sqrt{m}$ so that each bucket will fit in private storage in step 5, and the tally built here, with one counter per bucket, also fits in private storage. The step requires $O(m \log\log n)$ time and $O(\sqrt{m})$ private storage. (To avoid redundant scans, this step can be merged with the previous).

5b **Add fake bits to make the bucket sizes equivalent.** The local tally from step 1 indicates the size of the largest bucket. We scan the tally, adding fake encrypted positions to the server-side list of encrypted positions as we go, so that all the buckets will have the same size as the largest bucket after the step 3 bucket sort. To add a fake position that will correspond to bucket j, the position $j\sqrt{m}$ is added to the list. (A simple balls and bins result predicts that the $\sqrt{m}$-sized buckets all have similar sizes, already; the number of fakes to add is small compared to the number of real items). Let max be the index of T such that $T[max] = max_{j=1}^{\sqrt{m}} T[j]$. For each $j = 1..\sqrt{m}$ generate $T[max] - T[j]$ fake values v_l such that the $\log m/2$ most significant bits of each v_l are equal to j. Store the encrypted, $E(v_l||'fake')$ value in L. This operation ensures that all the buckets of Bkt will store the same number of elements. The bucket size tally is discarded after this step. This step requires $O(m \log\log m)$ time.

5c **Obliviously scramble the list of Bloom filter positions.** The encrypted indexes (the bit-positions of L, including the fakes) are scrambled, according to my Oblivious Merge Scramble Algorithm, which destroys all correlation between the old positions and the resulting positions, which are a new uniform random permutation. The new list L stores the scrambled values. The algorithm requires $O(m \log^2 \log m)$ time and $O(\sqrt{m \log m})$ private storage.

5d **Bucket-sort the list of Bloom filter positions.** For each $E(p) \in L$, let $idx(p)$ be the $\log m/2$ most significant bits of p. Then, do $Bkt[idx(p)].add(E(p))$. Here the Bloom filter's scrambled, encrypted positions are bucket-sorted. The client retrieves each bit index, decrypts it to read it, and writes the encrypted value back to the bucket on the server corresponding to the $\frac{\log m}{2}$ most significant bits of the position. The bucket sort allows us to construct the encrypted Bloom filter in the next step without revealing to the server which bits are set: if we were to simply scan the entire list of positions setting the corresponding bits to true, the server would observe the bit flips in our encrypted array and learn what positions are set. The bucket sort groups related positions together so that we can build the Bloom filter from left to right in a single pass. This step requires $O(m)$ time.

5e **Construct the Bloom filter.** For each $j = 1..Bkt.size$, download $Bkt[j]$. Note that the size of $Bkt[j]$ is $\sqrt{m}$. Let $BF[j\sqrt{m}]..BF[(j+1)\sqrt{m}]$ be the segment of the Bloom filter corresponding to $Bkt[j]$, where $BF[idx]$ denotes the idxth bit of BF. For each $E(p) \in Bkt[j]$, let x be the least significant $\log m/2$ bits of p. Do $BF[j\sqrt{m} + x] = 1$.

Store $E(BF[j\sqrt{m}]..E(BF[(j+1)\sqrt{m}]$ on the server. Finally, store the oblivious Bloom filter of the working set W on the server.

Here the client downloads each bucket (which conveniently fits into local storage). The bucket corresponds to a $\sqrt{m}$-sized segment of the final Bloom filter—all positions in this bucket refer to a bit in this segment of the filter. The bits corresponding to listed positions are set to true, with all other bits set to false, in the local copy. The client encrypts this Bloom filter segment and uploads it to the server. Observe that the server has no indication of how many bits are true in this segment (other than that it is limited by the bucket size), nor which are true. The Bloom filter is finished at the end of this step. This step requires $O(m)$ time, and $O(\sqrt{m})$ private storage.

5.1.8 Bloom Filter and Query Privacy

Recent analysis of my construction by Pinkas and Reinman [108] reminds us that any Bloom filter false positives result in an access pattern leak. Further, it was suggested by Kushilevitz et al. [72] that eliminating the possibility of false positives could result in a performance penalty. To address these observations, I show here that the construction avoids false positives with high probability.

A Bloom filter containing z items has two more parameters: h, the number of hash functions used (the number of bits set per item in the filter), and x, the number of bits in the Bloom filter, yielding the false positive rate $r = \left(1 - \left(1 - \frac{1}{x}\right)^{hz}\right)^h$.

The number of items z is determined by the level size, at 4^i for level i. The number of hashes used h is set to the security parameter c. Finally, we set the size of the Bloom filter x to scale proportionally with z and the number of hashes h, at $4^i ck$ for a constant k. I show now that the probability of a false positive r for a single Bloom filter lookup with these parameters is negligible in the security parameter.

Observe, as pointed out by Pinkas and Reinman [108], that we are not restricting the number of Bloom filter hashes to minimize the false positive rate. This allows flexibility in finding the performance optimal trade-off between Bloom filter size (determining construction cost) and the number of hash functions (determining lookup cost).

Lemma 6. *A Bloom filter containing $z = 4^i$ items, using $h = c$ hashes, and with $x = 4^i ck$ bits has a false positive rate negligible in the security parameter c for $k > 1$.*

Proof. The portion of bits in the Bloom filter that are set is upper bounded by $\frac{zh}{x}$, yielding the looser bound $r \leq \left(\frac{zh}{x}\right)^h$.

$$r = \left(1 - \left(1 - \frac{1}{x}\right)^{hz}\right)^h \leq \left(\frac{zh}{x}\right)^h = \left(\frac{4^i c}{4^i ck}\right)^c = k^{-c}$$

This quantity is negligible in c. □

With $x \geq cz$, the cost to obliviously construct a Bloom filter sized x is shown earlier to be $O(x \log\log x)$ (with sufficient storage); since x is proportional to z, the asymptotic performance is not affected by the Bloom filter. Moreover, the time required to query this Bloom filter is $O(c)$, which is constant for any chosen security parameter c.

However, we require $\log_4 n$ lookups (in different Bloom filters) to perform an access. $k^{-c} \log_4 n$ is not strictly negligible for a variable n. Thus, we instead require the failure probability per Bloom filter lookup to be within $k^{-c}/\log_4 n$, so that the union bound gives us a total probability of failure per *access* of k^{-c}.

To achieve this we revise the Bloom filter parameters. The number of items z is still determined by the level size, at 4^i for level i. The number of hashes used h is now set to $c + \log_k \log_4 n$. The size of the Bloom filter x should still scale proportionally with z and the number of hashes h, at $x = khz = 4^i k(c + \log_k \log_4 n)$ for a constant k. The probability of a Bloom filter false positive with these parameters is negligible in the security parameter, now over a period of $\log_4 n$ lookups.

Since the value of n increases by one each query, n is not constant over the lifetime of a level instance. However, once $n > 2$, the database size will never again double during this period. Thus, choosing a number of hash functions h based on a value of $c + \log_k \log_4 2n < c + 1 + \log_k \log_4 n$ suffices to limit the probability of a false positive over the entire duration of a level, even as n is increasing.

Theorem 18. *A set of* $\log_4 n$ *queries over a Bloom filter containing* $z = 4^i$ *items, using* $h = c + \log_k \log n$ *hashes, and with* $x = 4^i k(c + \log_k \log_4 n)$ *bits has a false positive rate negligible in the security parameter* c *for* $k > 1$ *over a period of* $\log_4 n$ *lookups.*

Proof.

$$r \leq \left(\frac{zh}{x}\right)^h = \left(\frac{4^i(c + \log_k \log_4 n)}{4^i k(c + \log_k \log_4 n)}\right)^{c + \log_k \log_4 n} = k^{-c - \log_k \log_4 n} = \frac{k^{-c}}{\log_4 n}$$

Taking the union bound over $\log_4 n$ Bloom filter lookups, the probability of failure is bounded by $r \log_4 n \leq k^{-c}$. This quantity is negligible in c. □

The time to query this Bloom filter is $O(c + \log\log n)$. The time/communication required to construct this Bloom filter obliviously is $O(x \log\log x) = O(4^i k(c + \log_k \log n) \log\log(4^i k(c + \log_k \log n)))$. Choosing a constant k, and amortizing over the shuffle period of 4^i queries, this simplifies to a shuffle cost per query of $O((c + \log\log n)(\log\log n + \log\log((c + \log\log n))))$. For a constant security parameter c, and since $\log\log\log\log n < \log\log n$ for $n > 1$, this shuffle cost is $O(\log^2 \log n)$ (per level). The construction cost across all levels is $O((\log n)(\log^2 \log n))$. In summary, guaranteeing a failure rate negligible in the security parameter c across all $\log n$ lookups for a given query raises the amortized shuffle cost per query from $O(\log n \log\log n)$ to $O(\log n \log^2 \log n)$. Further, it raises the online lookup cost from $O(\log n)$ to $O(\log n \log\log n)$.

Theorem 19. *The server learns nothing about the client access pattern by observing queries.*

Proof. Theorem 8 shows that all items at a particular level are scrambled uniformly randomly. The Bloom filter construction algorithm also scrambles the positions uniformly randomly, so that no correlation can be maintained before and after the scramble. Semantic security of the encryption function ensures that the encoding of an item reveals nothing about its position after the scramble.

The use of keyed secure hash functions guarantees that unique queries will appear uniform random; the server gains no additional information by watching the queries. The reshuffle schedule ensures that the same lookup is never run against a particular Bloom filter or level instance twice between scrambles; after an item is queried it is placed in the top level, where it will be detected the next time it is queried. As the levels overflow, reshuffling ensures that the item will have a new label when it reaches a level at which it had previously been located. Since the scrambles destroy all correlation, all queries therefore appear independent and uniform random to an observer.

The overall access pattern appears constant and predictable to the server; reshuffles happen at fixed, predetermined positions, and the only difference is in the uniformly random selection without replacement of the items queried at each level. □

5.2 Correctness and Integrity

In this section we introduce a set of integrity constructs that endow the above solution with correctness assurances. Specifically, we would like to guarantee that any storage provider tampering behavior is detected. All of these constructs can be implemented efficiently, with few or almost no overheads: (i) Message Authentication Codes (MACs) are added for all the stored items and Bloom filters; (ii) unique version labels for each item in the data covered by the MAC are added to prevent any replay-type attack in which the server incorrectly replies with a previously MAC-signed message; (iii) incremental, collision-resistant commutative checksums are added to checksum the item sets contained in each level, to prevent the server from hiding or duplicating items during the level reshuffle process.

For (i) we require a MAC function, such that the computationally bounded adversary has no non-negligible ability to construct any message, MAC pair $(M, MAC(M))$ for an M that did not originate at the client. Every item uploaded to the server is protected by such a MAC. (ii) is straightforward since level construction already labels items uniquely, and the non-repeating generation of each level is known by the client. For (iii), besides the requirement of being collision-resistant, it should be easy for clients to maintain and update a checksum of a set. In particular, when adding or deleting items, it should be possible to to so incrementally, without recomputing the entire checksum value.

The *incremental hashing* paradigm of Bellare and Micciancio [8] can be used to construct exactly such a checksum. Fix any cryptographic hash function H (viewed as random oracle) and a large prime p. To hash a set $B = \{b_1, \ldots, b_l\}$, we compute the product

$$H^*(B) := \prod_{i=1}^{l} H(b_i) \bmod p. \tag{5.1}$$

Note that this hash construction allows both for easy addition of item b (multiplication with $H(b)$) and removal of any b_i (multiplication by $(H(b_i))^{-1}$) without needing to recompute the hashes of all values $b_1, \ldots, b_l$.

It can be shown in the random oracle model [40] that it is computationally infeasible to find two sets that have the same checksum; the hash function (5.1) thus forms an easy and efficient way to authenticate the set of items in a level:

Theorem 20. *If the discrete logarithm problem in $\mathbb{Z}_p^*$ is hard it is computationally infeasible to find two sets $A \neq B$ with $H^*(A) = H^*(B)$.*

Theorem 21. *A client interface correctly implementing the above integrity constructs will detect all incorrect server responses before they reveal any part of the access pattern, or return an incorrect answer to the user of the interface.*

Proof. The proof is straightforward by construction. We enumerate all ways in which the server can violate correctness.

If the server does not respond to a query at all, the client will detect the denial of service, with the caveat that over any asynchronous communication line it may be impossible to determine who is responsible the denial of service, or whether the response may eventually arrive.

Construct (i) guarantees that if the server replies with to a query with data that is not accompanying a valid MAC for that data, the client will immediately detect the invalid MAC and return $\perp$, thus limiting the damage done to denial-of-service.

Thus, it now suffices to consider only those attacks in which the server responds with data authenticated by a valid MAC. If the MAC function is secure, this reduces to considering only attacks in which the server responds with out-of-date data that was originally sent from the client (e.g., a replay attack).

Since every correct server response in this protocol consists of one item initially uploaded by the client, the only type of tampering attack that remains is one in which the server returns the wrong item or version of items in any response.

Construct (ii) attaches a unique version ID to every subsequent version of an item. Moreover, at every step the client knows the exact unique version ID of the desired item a priori. The version id label is simply a combination of the desired item's label/id, the level id, and the reshuffle count of the level. Thus, any attack in which the server returns the wrong item or version of an item, the returned version ID (which is MAC-protected), will mismatch the expected version ID.

Finally, due to its incremental, collision-resistant nature, construct (iii) allows clients to properly authenticate the accumulated per-level writes/reads of items during the reshuffle phase. Specifically, at all points during the online query phase, the client maintains the incremental commutative checksum for a level—when a client removes/adds an item it also updates the checksum accordingly. Then, during the reshuffle phase, the level is scanned and copied at once. The client can now verify the previous operations by re-computing the checksum for the level and comparing it with the locally maintained one. □

5.2.1 Acknowledging Protocol Deviations

As discussed in the model (Section 1.2), acknowledging when the (potentially actively malicious) server violates the protocol could result in a privacy leak in itself. A protocol that halts when a MAC computation fails, for example, reveals to the server that the client is aware that the MAC fails, potentially linking this query to an item that the server had previously corrupted. Although economic incentives can discourage detectable protocol violation, and I show in Theorem 21 that protocol violations are, in fact, detectable, it is nevertheless informative to consider how to prevent a leak.

Acknowledgement of outright violation of message format specifications do not leak anything to the server, other than that the client has received and attempted to parse the message. In particular, the server only gains knowledge from deviating when it does not know ahead of time *when* the client will acknowledge the deviation.

We just have to show that two conditions of the protocol hold. First, that the client validates the (replay-resistant) MACs of server responses immediately, regardless of the nature of the data (e.g., fake or real item). This prevents the server from performing the previously described attack of corrupting data and waiting to see when the client acknowledges the corruption. This is because the server already knows exactly when the client will respond to the failure, since the client never modifies and propagates corrupted data to handle at a later time.

Second, since there is a phase of the level shuffle, where the MACs are not immediately replay-resistant, we have to show that no leaks result from in these delayed-MAC messages. That is, since we only verify the incremental level hash *after* shuffling, we have to show this does not introduce a vulnerability due to replay attacks. Fortunately, the server again already knows exactly when the client will observe a replay attack: at the end of the level shuffle. This is regardless of which data is replayed, so again the server does not learn anything by triggering client failure.

5.3 Performance Analysis

We now discuss main performance projections based on the network traffic, computation, and disk accesses. considering the following configuration: disk throughput = 80 MBytes/sec; disk sector size = 4096 bytes; disks are rotational-latency-free solid state disks (SSDs); client symmetric encryption crypto throughput = 100 MBytes/sec (from benchmarks produced by Lipmaa [77]); network throughput = 1 Gbit/sec; network latency = 5ms. We set the number of Bloom filter hash functions to 50, and determine from that the corresponding Bloom filter size required to establish a false positive rate of 2^{-128}. The construction cost amortized per query, plus the online cost per-query cost, is graphed against a growing database size in Figure 5.3.

This plot assumes an optimal implementation in which resource costs of level construction can be paid in parallel: the server disk writes simultaneously with the client encrypting data and sending it on the network. That is, instead of paying these costs sequentially, we

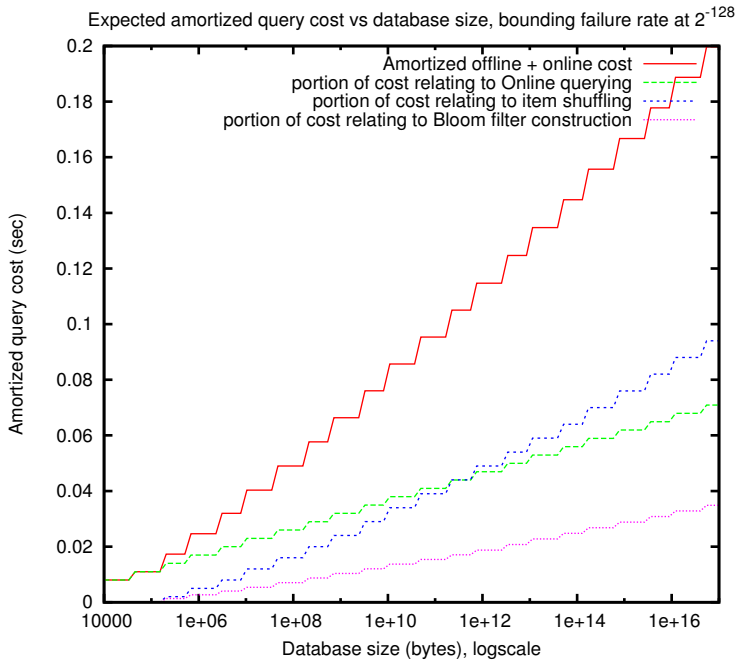

Figure 5.3: Plot of the expected amortized query cost, broken down by source. The sort cost plotted is the bottleneck resource usage, summed over each of the $\log_4(N/\text{blocksize})$ levels.

assume that the there is a bottleneck formed by the most limiting resource (in this case, the server disk throughput). Aside from start-up costs, this is roughly accurate for an optimized implementation. Reads during level construction, for example, are all deterministic; the server can read ahead to keep the disk busy even while the client is processing other data.

We plot three costs and their sum. The online query cost reflects the cost of performing $\log_4 n$ Bloom filter lookups on the server and reading the resulting items. This includes reading of $50 \log_4 n$ disk sectors: one for each Bloom filter position indicated by the secure Bloom filter lookup hash function at each level. It also models the disk read, network transfer, and decryption of the $\log_4 n$ items themselves.

The Bloom filter construction cost models the *simplified* Bloom filter construction procedure, in which a merge Sort is used in place of the merge Scramble. The merge scramble has an equivalent I/O cost but requires slightly more client computation (negligible in this scenario). The plot models the bottleneck resource usage, in this case disk throughput, limiting at 80 MBytes/sec. As justified below, we assume the full disk throughput can be obtained, despite the use of scattered reads smaller than a sector in this construction.

The item Oblivious Merge Scramble cost is analogous: it represents the bottleneck cost of $\log_2 \log_2 n$ passes over the data at each level.

5.3.1 Discussion

The graph suggests an optimized implementation can achieve over 8 queries per second on a 1 terabyte database, and over 5 queries per second on a 10 petabyte database. This assumes, of course, the previously established $c\sqrt{n \log n}$ client storage.

We assume rotational-latency-free solid state disks (SSDs) because our sort requires scattered data reading. However, use of SSDs does not immediately bring the observed disk throughput up to the optimal disk throughput, for the following reason. Although the disk throughput is roughly on par with the other potential bottlenecks of client encryption throughput and network throughput, the granularity of disk I/O comes into play. That is, even though we assume a latency-free SSD, scattered reads and writes *smaller than a sector size* still have the cost of reading and writing the entire sector.

As an example, a somewhat tricky case to analyze arises once the data associated with Bloom filter construction is too big to fit in server memory. The Bloom filters themselves contain approximately 250 bits (≈ 50 of them set to *1*) for every item stored at the level. In our configuration, assuming Bloom filter segments of 16 bytes (each with a 16 byte MAC), this works out to only around 6.25 GB of Bloom filter data for a 1 TB database. However, the temporary lists used to scramble and build this same Bloom filter with privacy and correctness require a whopping 320 GB. Recall that ≈ 50 Bloom filter positions per level item are appended to a list, along with a MAC for each, and padded to be the same size as a 32-byte Bloom filter segment.

The apparent effect is that sorting this 320 GB list might require a full disk sector read and write per sort operation, when only 32 bytes are needed. This involves no additional penalty when sorting items as big as (or a multiple of) the disk sector size, but potentially cuts disk throughput by a factor of 128 when sorting our 32-byte Bloom filter positions. This penalty is imposed in most of the passes, for example, of a Randomized Shell Sort [46]. On the other hand, the Oblivious Merge Sort described in Chapter 3 would not impose this penalty (or even an analogous disk seek penalty), since the data is read sequentially.

Where, then, does the Oblivious Merge Scramble stand? I show now that with intelligent caching and room for $\sqrt{n}$ sectors in server RAM, (e.g., 1 GB of RAM for a 1 TB database is more than enough) we can avoid this disk I/O penalty. In the first step of the Oblivious Merge Scramble, there are $\sqrt{n}$ items being merged together at once. In this case, the entire contents of any single merge operation fit in RAM simultaneously, requiring only a single sequential read at the beginning and a single sequential write at the end.

Observe that the remaining merge steps (2 through $\log_2 \log_2 n$), have two properties. First, the subarrays are all size at least $\sqrt{n}$, which is greater than a disk sector for all potentially problematic database sizes. That is, databases smaller than, e.g., 4096^2 blocks, have Bloom filters small enough to be constructed entirely in server RAM. Second, the total number of subarrays is at most $\sqrt{n}$. This means that an entire sector (e.g. 4096 bytes) of every subarray can fit in server RAM. With careful disk cache use, we can read data sector-by-sector without wasting disk throughput. This means we can operate at the full SSD disk throughput. Note that this does not imply sequential reading of the data (nor avoidance of any rotational disk seek costs).

5.4 Conclusions

In this chapter we introduced a first practical oblivious data access protocol with correctness. The key insights lie in new constructions and sophisticated reshuffling protocols that yield practical computational complexity (to $O((\log n)(\log^2 \log n))$) and storage overheads (to $O(n)$). This mechanism simultaneously provides *full computational privacy* and *correctness*.

Chapter 6

De-amortized and Parallel ORAM

We will tackle three more barriers to ORAM practicality in this chapter: serially-limited querying, usability, and worst case query cost. First, instead of waiting for the completion of all ongoing client-server transactions, client threads can now engage a server in parallel *without loss of privacy*.

This critical piece is missing from existing Oblivious RAMs, which can not allow multiple clients threads to operate simultaneously without revealing intra- and inter-query correlations and thus incurring privacy leaks. And since ORAMs typically require many communication rounds, this significantly and unnecessarily constrains throughput. The mechanisms introduced here eliminate this constraint, allowing overall throughput to be bound by server bandwidth only (and thus increase by an order of magnitude).

To tackle worst case query cost, new de-amortization techniques bring the worst case query cost in line with the average cost. Though I am not the first to provide an ORAM with similar worst and average case query costs, this property has been shown to be fundamental to any ORAM.

To address usability, I designed, built and analyzed PD-ORAM, a high performance, fully functional implementation. It performs multiple queries per second on a 1TB+ database over 50ms latency links, with unamortized, bound query latencies. Based on PD-ORAM, *privatefs*, the first oblivious file system, was built and deployed on Linux as a userspace file system, and will be made available to the open-source community.

6.1 Model Extensions

The adversary is assumed throughout the bulk of this chapter to be honest but curious; however, Section 6.3.6 details inexpensive adjustments that ensure "fork consistency" against even an actively malicious adversary. Defined by Li et al. [75], fork consistency acknowledges that a malicious server can present different versions of the database to different clients, e.g., by not including updates from some of the other clients, but guarantees that each "fork" view is self-consistent. Moreover, clients will detect this behavior if they ever communicate with each other, or if the server ever attempts to rejoin the views. For simplicity, timing

attacks are not handled.

Let us consider some new definitions, which allow us to model parallel clients, as well as apply these techniques to other ORAMs.

6.1.1 Parties

An ORAM setup consists of three types of parties: ORAM Clients, who issue read and write queries, the limited-storage ORAM Instances, who satisfy these queries for their client while maintaining privacy, and the ORAM Server, who has plentiful storage and is willing to help the ORAM Instances, but is deemed untrustworthy (Figure 6.1).

The semantics used here correspond to existing ORAMs (read and write of fixed-size blocks is supported), with the addition of concurrency and multiple "Instances". Because of the introduction of concurrency, it is also appropriate to offer an atomic test-and-set instruction to ORAM clients. Analogous to the CPU primitive, it updates the value stored at a location, returning the value previously stored there.

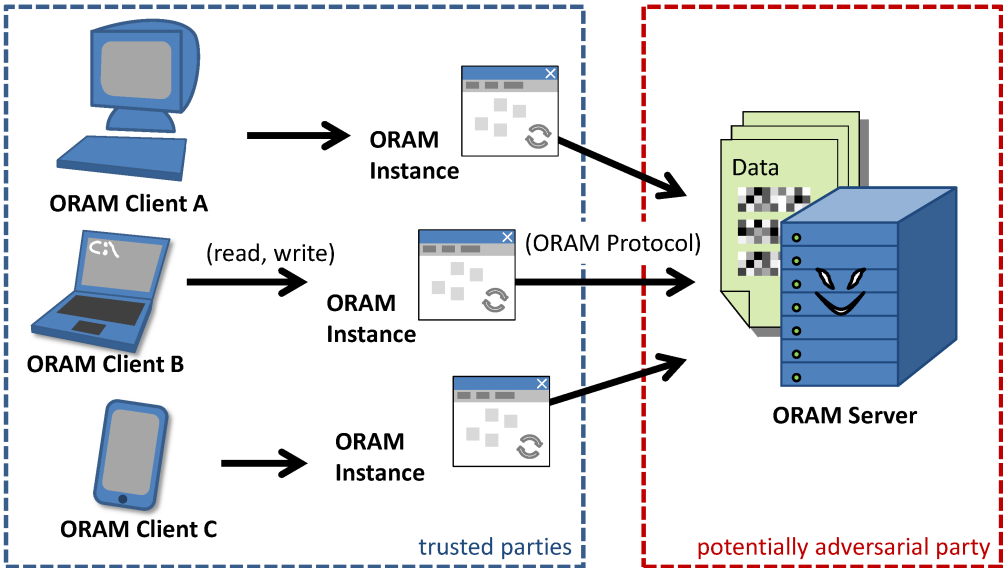

Figure 6.1: Overview: ORAM Clients access data obliviously through a simple interface of an ORAM Instance. The stateless ORAM Instances use an (untrusted) ORAM Server to store and retrieve the data obliviously.

ORAM Client. a party who is authorized to issue reads and writes to the ORAM Interface. Data is accessed in "blocks", a term used to denote a fix sized record. "Block" is used instead of "word" to convey target applications broader than memory access (including file system and database outsourcing). Block IDs are arbitrary bit sequences. There are multiple clients. Each ORAM client has access to the following interface provided by a corresponding *ORAM Instance*: read(id): val; write(id, val); test-and-set(id, val): val.

Accesses are serialized in the order the ORAM server receives them, which guarantees each client sees a serialized view of its own requests. Among multiple clients, however, the ordering is not guaranteed to correspond with the time-ordering of incoming client requests.

ORAM Instance. a trusted party providing an ORAM interface to ORAM Clients. Instances are stateless but have access to the *ORAM key* (a secret shared among instances, enabling access to the ORAM) and address of the server.

The Instance-to-Server protocol details are implementation-specific (and typically optimized to the instance to minimize network traffic and the number of round trips).

ORAM Server. the untrusted party providing the storage backend, filling requests from the instance. It is assumed here to be honest but curious.

The ORAM Instances communicate only with the server (not with each other). The server is curious but otherwise honest. Thus, "fork (in)consistency" attacks [75]—in which a malicious server presents different versions of the database to different clients, e.g., by not including updates from some of the other clients—are not considered. Additionally, for simplicity, timing attacks are also not handled here.

Communication between the ORAM Client and ORAM Instance is assumed secured, e.g., with access controls on IPC if they are on the same machine, or with SSL otherwise. Communication between the ORAM Instance and Server is also secured, e.g., with SSL.

Notation. p = upper bound on number of parallel clients, i = a level within a pyramid-based ORAM (for the smallest, "top" level, $i = 0$), c = security parameter, n = database size, in blocks, $\cong_p$ is equivalence modulo p.

6.2 Parallel Queries: a First Pass

We now examine how to query existing ORAMs in parallel. The idea is to start with an ORAM on which it is safe to run non-repeating ***unique*** queries (queries targeting different data records in the underlying database) simultaneously, and to build from this an ORAM which can also safely run ***colliding*** queries (targeting the same underlying data record).

Consider a classic ORAM [41]. We can augment it to handle parallelism for sets of unique queries as follows.

- First, consider that as a client query searches across the database for a particular item, its accesses are by design indistinguishable from random.
- Second, by ORAM construction, different unique queries touch independent sets of database locations (because their coin-flips are independent).
- Now consider a set of multiple queries. Any of its reorderings results in accessing the same locations (provided each query gets the same coin flips), albeit in a different order.
- Thus, intuitively, it is safe for clients to submit unique queries simultaneously: the server sees an identical transcript independent of the queries.

The above privacy intuition only holds for non-repeating unique sets of queries, of course. Clients simultaneously querying the same item reveal this to the server, as their accessed locations are substantially similar. This raises the interesting question of how to guarantee query uniqueness over arbitrary incoming client query patterns, without revealing any inter-query collisions to the server.

Since the model requires the ORAM Instances to communicate only via the ORAM Server[1], one idea is to have Instances synchronize with each other via a server-hosted data structure, in a way that prevents overlapping equivalent queries while still guaranteeing indistinguishability of all query patterns.

Strawman. The simplest approach to ensuring uniqueness is for each new client to examine ongoing queries. Finding an intersection, the client can then wait for ongoing queries to complete before trying again. This is fundamentally insecure, however, since the server learns when a query is being repeated by a later client, since the later client waits for the previous client. Further, correcting this by making all clients wait for all previous queries to complete before initiating their own query defeats the goal of parallelism.

Alphaman. To fix these issues, the server will help clients maintain an encrypted query log. Scanning this query log allows an Instance to identify simultaneously ongoing requests. In the case of overlap, a random unique query is executed instead. Once its query (real or random) is completed, the Instance reports its result back to the shared cache/result log. It then searches in the log for the result of the simultaneously ongoing overlapping query it had identified (if any). This guarantees that in either case the Instance gets to learn the result it seeks.

I detail this below (Section 6.3.2, Figure 6.2).

6.3 Abstractions and Solutions

This section defines the properties required to build parallel, de-amortized ORAMs, and provides corresponding constructions.

6.3.1 Abstraction for Parallelism: "Period-based ORAM"

The main idea is to run queries simultaneously *between* reshuffles. We are limited by the size of the top level: this is the number of appends that can be performed before a shuffle is required. Thus, let us designate the maximum parallelism *and* the size of the top level as c. Subsequent levels will be sized $c2^i$ for this reason.

Definition 4. *A* **period-based stateless ORAM Instance** *is an ORAM that performs a series of p queries between each shuffle. The transcripts of unique queries within a period are independent of their order. Previously executed queries are scanned from a server-stored cache sized p, triggering a fake lookup instead.*

[1] capturing an adversary with a complete view of the network

Insight. The goal of this abstraction is to capture the fact that the underlying ORAM already supports up to p simultaneous *unique* queries. Specifically, the ORAM runs in periods of queries over which the transcripts of unique queries are independent (e.g., the original ORAM [41] satisfies this). For a given period between reshuffling (which lasts several queries, until the top level overflows), and choosing the random number generator coin flips for two unique queries ahead of time, the transcript for each query is the same regardless of the order they are run. The transcripts of two identical queries, however, are inter-dependent, since the first-to-execute query necessarily searches farther down in the database than the second query, as the item is moved up to the top level once the first query completes. The second-to-execute query finds the item immediately at the top, and thus the remainder of the search is random.

6.3.2 A Solution for Parallelism

The motivation behind this protocol is to minimize waiting, while guaranteeing serializability. The basic assumption about the underlying ORAM is that its data structure supports simultaneous querying for unique blocks. Then, the *ultimate purpose of this protocol is to make sure all the simultaneous queries are unique.* The challenge is to juggle this uniqueness requirement (especially in the presence of colliding queries from different clients) with the requirement that the server not learn any inter-query correlation.

One solution is to obliviously guarantee uniqueness of queries using an append log in combination with a results log. Clients will still need to wait for previous queries to complete before outputting a result, but now there is no requirement of blocking on other clients during query execution. This increases throughput significantly (Figure 6.2):

1. The Client issues a query to its ORAM Instance.
2. The ORAM Instance generates the request, consisting of the block ID, and a bit indicating whether this is a read or a write/insert. Test-and-set operations are identified as writes.
3. The request is encrypted and sent to the server.
4. The server appends this to the query log (analogous to the top level), and returns a sequential query ID, together with the query log, containing all queries since the top level was last emptied.
5. The ORAM Instance interactively queries the underlying ORAM. If the query was already in the query log (i.e., from another running client), the instance runs a dummy query instead.
6. The ORAM Instance sends its result, encrypted, back to the server, which appends it to the query result log.

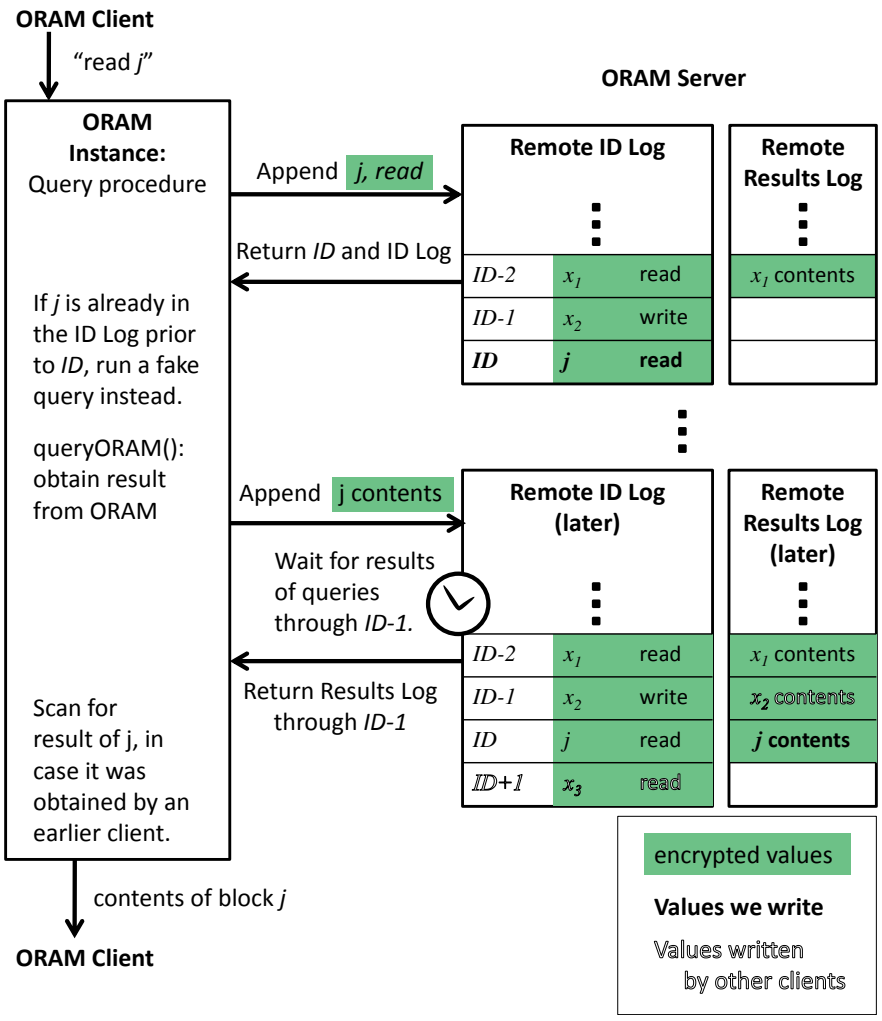

Figure 6.2: Parallel query protocol, as described in the text. This illustrates the interaction between a single client and the server for performing a single query. Other clients may be performing queries simultaneously; the challenge here is to avoid intersections during parallel query execution, without revealing when any queries intersect.

7. The ORAM Instance reads the results log up to its own entry (only interested in previous clients' results), in case the current query is accessing a block that was previously read or written. This is the only step that requires waiting for the earlier queries to complete.

8. The ORAM Instance returns to the ORAM Client the result (obtained from the database, or the result log).

Server network traffic over a period of p parallel client queries is thus quadratic in p, since each Instance needs to be aware of what each simultaneous Client is doing[2]. As a result, the number of parallel clients that optimizes total throughput is a function of network bandwidth, latency, and database size.

On the one hand, for p clients, and a database of n blocks, the sequence of $\log_2(n) + 3$ round trips per query imposes a network-latency based maximum query throughput of

$$\frac{p}{((\log_2 n) + 3) \times \text{latency}}$$

On the other hand, the cost of supporting multiple clients (quadratic in p over p queries; linear in p per query), and the online data transfer cost of $\log_2 n$ blocks, impose a server bandwidth based maximum throughput:

$$\frac{\text{bandwidth}}{((p-1)/2 + \log_2 n) \times \text{blocksize}}$$

To find the optimal number of clients p for a given configuration the lower of these upper bounds needs to be maximized. This relationship is plotted in Figure 6.3 for a representative setting.

After p queries execute, the query log is converted into the ORAM top level and thus the queries are applied to the database, in the ORAM-sense (Figure 6.4). This is where the requirement of a "period-based" Instance is necessary.

In the process, the shuffler has the role of consolidating the query log. A single block may be accessed multiple times by different queries, but is placed only once back into the database. The value chosen for a given block is either the last write, if there is one, or otherwise the first read. Recall that subsequent reads are associated with fake results.

Properties. It is now time to informally define and sketch proofs for two main properties: *optimality* and *query privacy*.

Theorem 22. *In any model in which the server can associate all visible read/write data accesses corresponding to a given query, hiding access patterns in a "non-simultaneous ORAM" requires waiting for the results of all previous queries ("wait-optimality").*

[2]To disseminate this information in a more network-efficient manner, e.g., by only requesting the log entry the client is interested in (instead of the entire log) a PIR protocol may be deployed in accessing the log. This would results in a trade-off between server bandwidth and server CPU time. As shown by Sion and Carbunar [116], this trade-off is mostly unfavorable, yet recent results [99] indicate that it may become feasible in the near future.

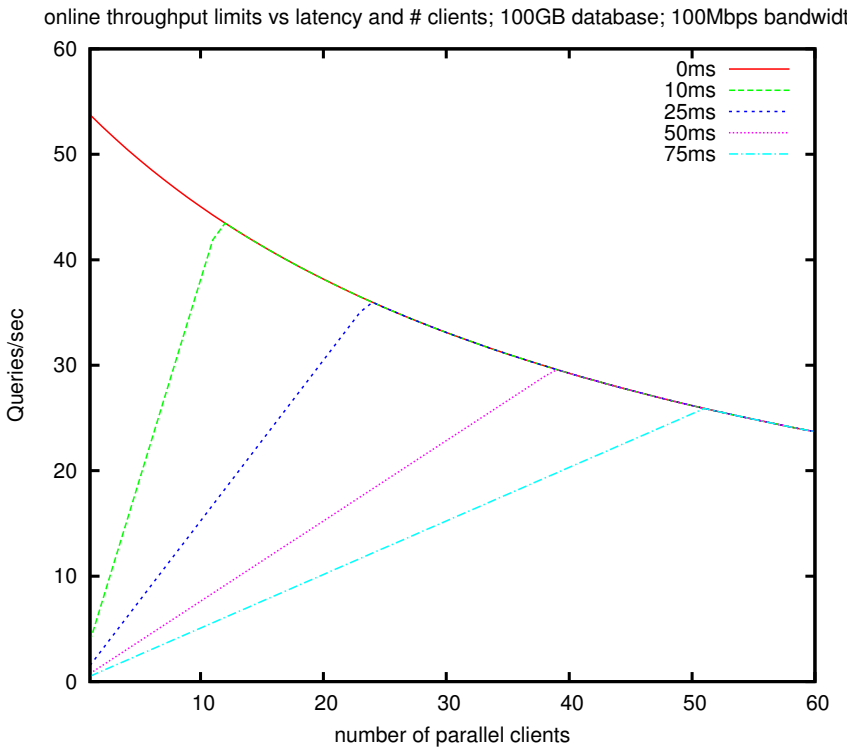

Figure 6.3: Upper bound on online query rate of a 100GB database (blocksize = 10^4 and $n = 10^7$) and assuming a 100 Mbit (12.5 MB/sec) network link. The plot is shown for various network round trip times, from 0 ms through 75 ms. Observe that single clients, incurring $\log_2 n$ round trips per query, are tightly bound by the round trip time. Adding more parallel clients increases this throughput linearly, up to the point where bandwidth limits from the query log traffic take over.

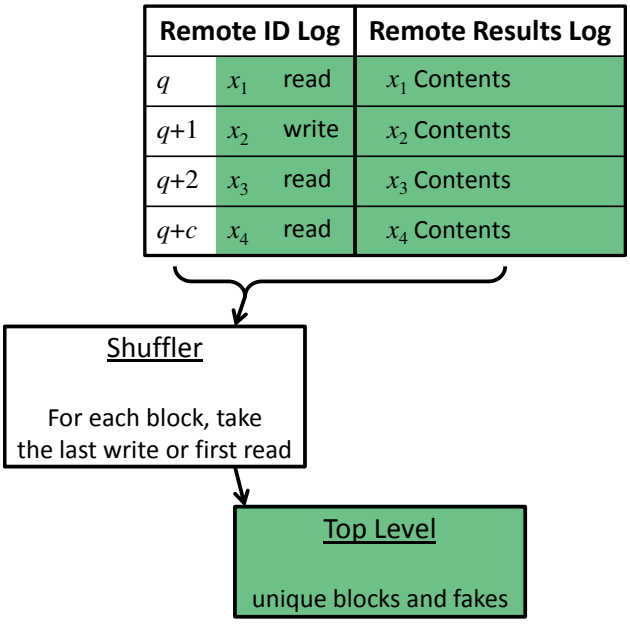

Figure 6.4: Reconciliation of the query log into the new top level. After a period of p queries, the query log is shuffled and becomes the top level. Regarding the query log hosted set of ids, consider that a block can only be read once from the ORAM during this period. Thus, the correct value for a given block is either the last write of this period, if there is one, or otherwise the first read.

Proof. (outline) Given any ORAM, unless it supports simultaneous execution of equivalent queries (for the same underlying block), Instances must wait for all queries prior to theirs to complete before outputting a final result. This is since repeating the query before the previous query has completed, (resulting in overlap among the visible data accesses) would leak the correlation between these queries. Instead of attempting to repeat a previous query, clients must obtain the result from it—but they can do so only once that query has finished executing.

Now, for parallel execution, to make all query sequences indistinguishable to the server, clients still need to wait for the prior queries, as if their query depended on all of the previous queries (lest they reveal which query they are waiting for, if any). □

Theorem 23. *If there exists an adversary with non-negligible advantage at violating query privacy in a parallel ORAM, there is an adversary with non-negligible advantage at violating the privacy of the underlying single-client ORAM ("query privacy inheritance").*

Proof. (outline) Consider that in the parallel case, client behavior from the perspective of the server is identical for each new query instance regardless of whether and how often the same equivalent query appears "earlier" in the log. For every query, the server always sees a semantically secure encrypted *append* operation to the query log, a *query* to the underlying ORAM, and a *scan* of the results log up to that point.

Intuitively, the only difference now is that transcripts of the different queries are interleaved, but otherwise contain the same accesses as when executed in a traditional ORAM. This is so because in the parallel case, if a client queries for a block that is already queried for by a simultaneously ongoing query, the client's ORAM Instance will instead issue a fake query—which is exactly what it would have done anyway in the traditional ORAM (had it found the query result at the top level). Thus, from the server's point of view, the transcripts contain the same (random looking) accesses.

Further, query transcripts are independent of their query and, without knowledge of the secret ORAM key, indistinguishable from random.

Then, an advantage at distinguishing the new transcripts translates into an equivalent advantage at distinguishing the underlying ORAM's if parallelism were not enabled. The reduction is straightforward and both inter-query correlation and access privacy can be shown to be protected in the IND-CQA model (Section 1.5.1). Details are out of scope. □

6.3.3 "Level-based Amortized ORAM" Abstraction

This section introduces a generic de-amortization technique, together with the abstractions necessary to deploy it to de-amortize a large class of ORAMs and reduce their worst case query cost to the average case.

Consider a "level-based amortized ORAM" which divides the database into levels, with the more recently accessed items stored in the smaller top levels. Retrieval requires searching within a set of specific locations in each level from the top down, and, once found, lifting the result back up into the top level. The next time the same query is repeated, the block is discovered at the top, which causes the ORAM Instance to instead access a random set of

locations at each level below. The top level overflows periodically into the lower levels, and a level is guaranteed to get re-shuffled before a given query is repeated at that level. Thus each query looks like only an selection of random, independent locations in each level.

Definition 5. *A* **level-based amortized ORAM Instance** *is an ORAM that searches levels recursively, appending the result back to the first level. Privacy results from the property than an item is sought at a particular level no more than once between two consecutive shuffles of that level.*

Intuition. The goal of this definition is to identify those ORAMs that recursively query a data structure divided into multiple levels. These ORAMs guarantee privacy by enforcing that blocks are never sought in the same location twice. That is, after seeking a block once at a given level, that level will be "re-constructed" before the block is sought there again. Level-based ORAMs move each retrieved block up to the top level of the ORAM, from where it slowly makes its way back down the data structure as the lower levels are re-constructed. Level-based amortized ORAMs include constructions presented by Goldreich and Ostrovsky [41], Goodrich and Mitzenmacher [44], Pinkas and Reinman [108], as well as the constructions presented in Chapters 4 and 5.

6.3.4 A De-amortized ORAM Construction

I now construct a level-based de-amortized ORAM from a level-based amortized ORAM.

De-amortization techniques for level-based amortized ORAMs need to deal efficiently with the level constructions resulting from overflow of the top levels. Their goal is to arrange the levels such that they can be queried while the items are simultaneously being inserted and re-shuffled into new levels. That is, instead of suspending querying to wait for shuffling to proceed, a new level must be available as soon as it is needed.

The main idea is to provide pre-emptive shuffling. Rather than waiting for querying to complete before shuffling a level, its transformation into a new level begins as soon as a level is constructed, and right as its querying begins.

To allow this, one idea is to duplicate a level into two copies: a read-only variant that is used in the querying process, and a writable variant which is dynamically updated into the new generation of this level. The read-only copy is discarded at the end of the period, since it is no longer needed for either queries or level construction.

Level De-amortization. Consider first the de-amortization of a single level. In a traditional ORAM, a level is reconstructed by combining into it the above level that has filled up and now overflows (Section 2.1). This necessarily stops the query process for its duration.

To de-amortize this, when beginning its construction, instead of pausing queries and waiting for the construction to finish, querying can continue via the read-only level copy while a new generation is produced into the writable variant. Critically, during this process, existing levels can overflow into a fresh, empty, replacement for this level. (Figure 6.5).

Delete Log. The second change is to delay any level updates until the end of the shuffle. This is important because some ORAMs avoid the complexity of reconciling multiple versions

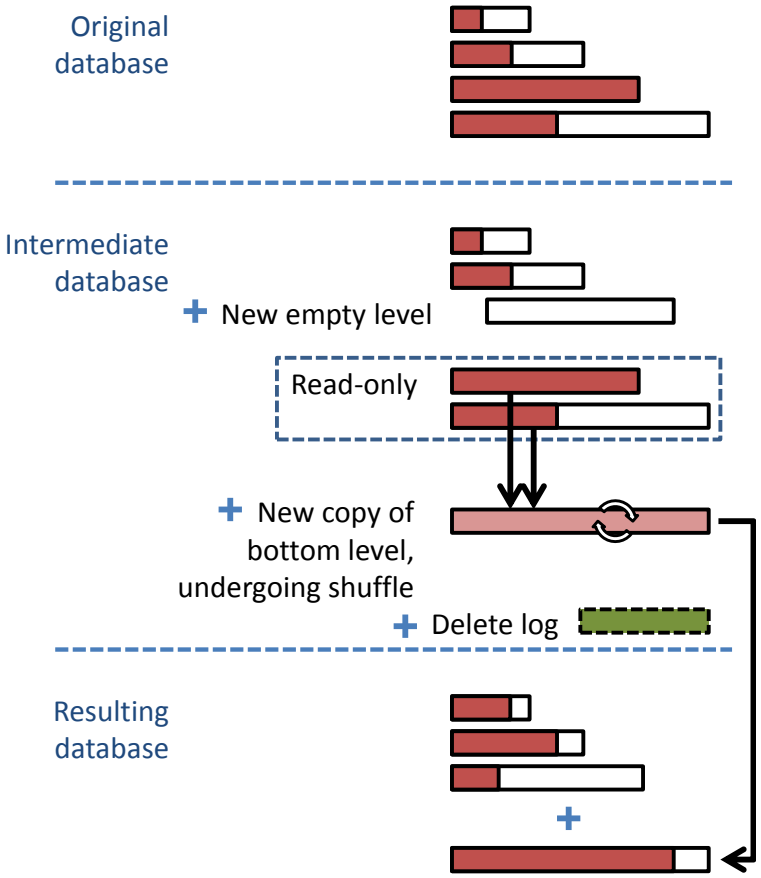

Figure 6.5: In-background construction of a single level. The top section represents the initial database state; the middle section shows the database state during the process of constructing the new bottom level; the bottom section shows the resulting database state. The scenario illustrated is as follows. The third level is full, so it needs to be combined with the fourth level. Read-only copies of those two levels are made, and can be queried while the combination is occurring. Simultaneously, overflows from the top level are placed into the a replacement third level. All five levels are accessed during queries at this time. Once the construction of the bottom level is complete, querying resumes with this new bottom level replacing the two read-only levels (which contain the same items).

of an item (and in the process also reducing storage overheads) by deleting blocks from the levels in which they are found. In the de-amortized construction, though, this is no longer possible, since the queried level copies are now read-only. Instead, these changes are appended to an update log—e.g., items marked for deletion in this log are now deleted by the server before the *next* shuffle of the level. Note that not all ORAMs modify the pyramid structure during queries; some [41] do not need such modification. This de-amortization protocol simply requires that those changes that are made can be applied *after* the level has been reconstructed.

Details. A level at height i, containing $m = p2^i$ items, is queried m times. At the end of the mth query, the shuffle will have completed, and the remaining items from this level are now in a new level.

Each level is a data set, completely specified by its height i, the sequentially increasing "generation" j at that height, and a one-bit marker for the odd generations. The two levels at an even generation j, denoted by $i.j$ and $i.j.*$, are combined to produce a level at the next height $i + 1$, generation $j/2$. Those levels at a height i with an odd generation j are reshuffled to produce level $i.j + 1.*$.

The levels currently being reshuffled are the ones queried. This means at any one time there are either one or two active levels (queried and being shuffled) at each given height, for $i \leq \log_2 n$. an illustration of the resulting shuffle schedule, is provided in Figure 6.6.

The above construction allows de-amortization of the construction of all the levels except the top level. De-amortization of the top level appears to be possible using a rotating query log, but is left as future work (since its impact is minimal as the top level is very small).

If based on an underlying stateless period-based amortized ORAM, the result is a stateless period-based de-amortized ORAM, suitable for parallelization.

Theorem 24. *If there exists an adversary with non-negligible advantage at violating query privacy in a de-amortized ORAM, then as long as the underlying level-based ORAM constructs levels according to an ideal secret random permutation, there is an adversary with non-negligible advantage at violating the privacy of this underlying level-based ORAM ("query privacy inheritance").*

Proof. (outline) The new construction maintains the same privacy-preserving property as the underlying level-based ORAMs: once a query seeks an item at a given level, no further query seeks that item at that level until reshuffled.

The construction of any given level proceeds equivalently to the level-based ORAM, except that during-query level changes are postponed to after reconstruction. When applied after level reconstruction these in effect reveal to the server portions of the level's reshuffling—since the server needs to know what to update or delete.

This is the only new information given to the adversary. Informally, privacy is shown by reducing to a property of ideal permutations: uncovering a portion of a permutation does not reveal anything about the remaining portion. Since the underlying level-based ORAM selects the level permutation according to an ideal secret permutation, this follows trivially: all arrangements of the elements in the "still secret" portion are equally likely. It constitutes a secret permutation in itself.

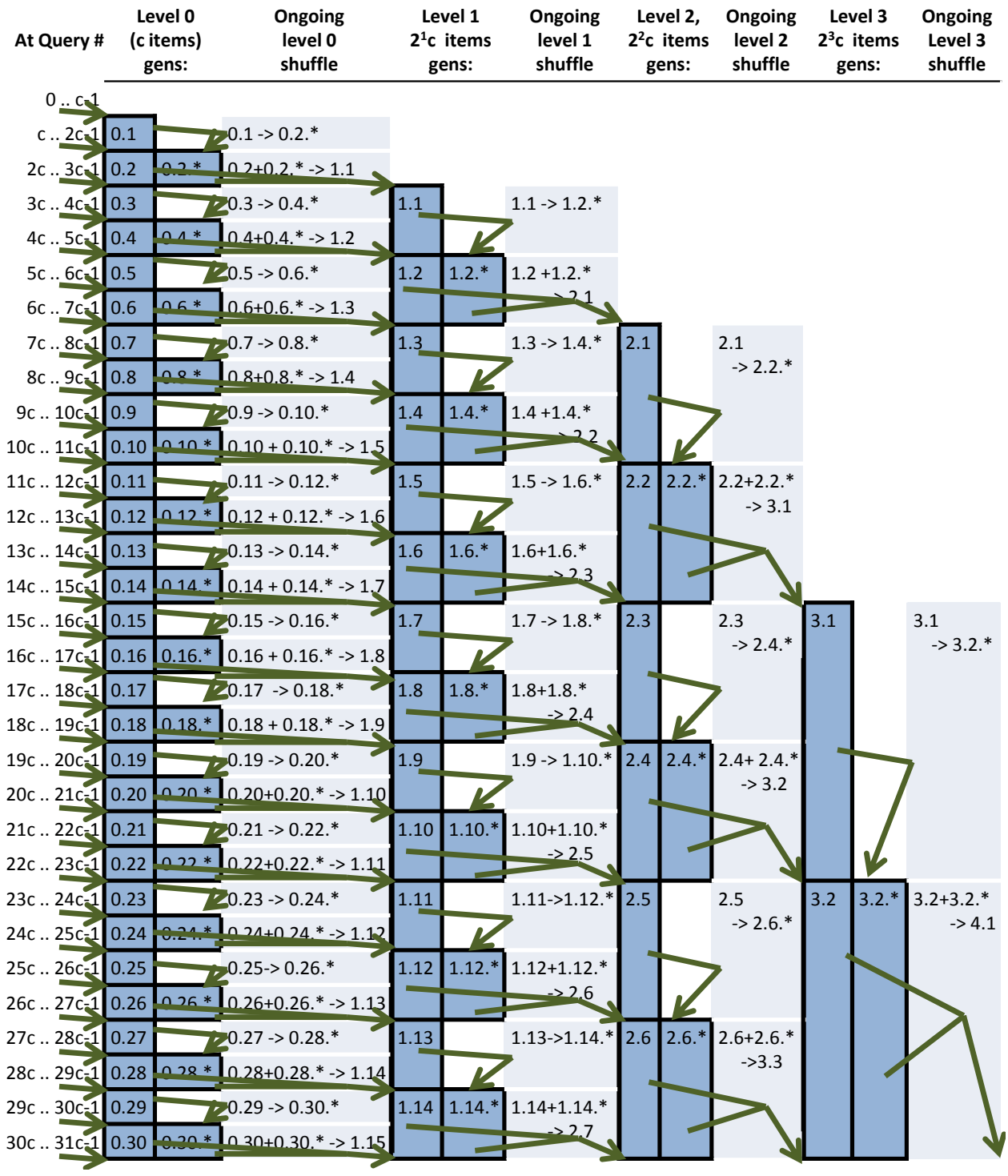

Figure 6.6: Shuffle schedule: Query numbers are listed in the first column. For a given query, the dark outlined boxes overlapping that row specify which levels and generations to access (from left to right). The light boxes identify the simultaneously occurring reshuffle.

The reduction proceeds by showing that the only additional information available to an adversary derives from these postponed level changes. Any advantage at violating IND-CQA in the de-amortized scheme translates to an advantage at violating IND-CQA in the underlying ORAM scheme. □

Storage overhead. Assuming the client storage required to shuffle level i is less than or equal to the space required to shuffle the next larger level, $i+1$, then de-amortization requires an extra factor of client storage upper-bound by $\log_2 n$ (since all $\log_2 n$ levels are being shuffled at once).

Communication overhead. This construction requires querying up to twice as many levels simultaneously. However, since it is known in advance that the item won't show up in both levels, this querying can be done in the same number of round trips.

Shuffling overhead. The amount of work performed per query is roughly similar. Additional overhead may stem from shuffling being done *before* level items are removed. But since only half of the items would be removed anyway, for ORAMs where the level i construction cost is $sm\log_2 m$ for constant s and $m = 2^i$, this keeps the new overhead for shuffling within a factor of $\frac{s(2m\log_2 2m)}{s(m\log_2 m)} = 2 + \frac{1}{\log_2 m} < 3$.

6.3.5 Bloom filter ORAM with Logarithmic Client Memory

I now detail the final piece of the construction: the underlying base ORAM mechanism. In Chapter 5 I introduced an ORAM that stores each pyramid level as an encrypted hashtable and an encrypted Bloom filter (BF)—indexing elements in the hashtable. The BF allows the client to privately and efficiently—no linear scanning of $O(\log n)$ fake block buckets for each stored block to hide the success of each level query as in previous ORAMs—identify the level where an item of interest is stored, which is then retrieved from the corresponding hashtable.

I use the simple Bloom filter construction described in Section 5.1.6 because of the reduced client storage requirements. This results in an $O(\log^2 n)$ BF-based ORAM requiring only *logarithmic* storage. Instead of using the "bucket sort" method that builds $\sqrt{n}$-sized chunks of the encrypted BF in client storage, the idea is to securely construct a list of all the BF segments and indexes, and sort them together using the memory-restricted Randomized shell sort provided by Goodrich et al. [46].

The PD-ORAM implementation assumes $c\sqrt{n\log n}$ client storage to employ the Oblivious Merge Sort instead of the disk-seek intensive inefficient randomized shell sort. Nevertheless, this simpler BF construction process is employed instead of the $c\sqrt{n}$ variant.

6.3.6 Security against malicious adversaries

Ensuring security against a malicious adversary first requires an underlying ORAM providing these guarantees. Fortunately, the Bloom filter-based ORAM I presented in Chapter 5 provides such a mechanism. Second, maintaining security in a parallel setting requires clients

to test that they all see a consistent view. This is achieved using a hash tree over the set of all previous queries. Whenever a client performs the top level shuffle, it is responsible for updating this hash tree, and signing and attaching the root value to the new query log. This entails hashing the current query log, appending it as a new leaf node to the hash tree, and recomputing hash values along the path from this new node to the top.

Second, whenever a client performs a query, it performs one additional operation: it verifies that the last query it performed is included in the hash tree, whose root corresponds to the value attached to this query log. This is done by verifying all nodes in the hash tree adjacent to the path from the root to its last request.

The PD-ORAM implementation analyzed in the following sections, however, assumes an honest but curious adversary.

6.4 Performance Analysis and Experiments

6.4.1 When Theory Meets Real Asphalt

Amortized Measurements. A significant challenge in measuring the performance of any amortized system is ensuring the trial captures the average performance, not just peak performance. This is complicated by the requirement of running trials for periods that are too short to encompass the full period over which the amortization is performed. For example, at even several queries per second, the reconstruction of the lowest level of a terabyte database is amortized over a period on the order of weeks or longer.

De-amortizing the level construction provides the opportunity of improving measurement accuracy. The challenge remains, however, of ensuring the de-amortized background shuffle proceeds proportionally to the query rate: the de-amortization is perfect when the new level construction is completed at the instant it is needed by a query. Inaccuracies in this rate synchronization will affect the measured results, since measured query throughput of a short period might be higher or lower than the sustainable rate.

To avoid this effect, PD-ORAM maintains *progress meters* for level construction, and only allows queries to proceed when every level is at least proportionally constructed. The level constructions processes are also suspended when a level gets too far ahead of the current query. This keeps querying and level construction smooth, minimizing worst case latency.

Proper de-amortization: Theory vs. Reality. Performing proper de-amortization proved a non-trivial systems challenge. Research solutions, including a construction by Goodrich et al. [45], and our own PD-ORAM (Section 6.3.4) express de-amortization in terms such as "perform the proportional amount of work required", or "perform the next $O(f(x))$ accesses." While these terms suffice for proving existence of a de-amortized construction, programming models don't typically provide this type of abstract control over code execution. PD-ORAM achieves this control using progress metering over the construction of individual levels. Since the level construction is a non-trivial operation involving different types of computation across the client and server, accurate progress metering required splitting level

construction into tasks whose progress can be reported over time. Moreover, this metering uses experimentally determined values to identify what portion of the level construction corresponds to which subtasks. Figure 6.10 illustrates the result of informing the progress meter with experimentally determined values.

As an aside, suspending the level construction when it outpaces the queries proved critical on the larger database sizes. The sheer number of requests being sent from the ORAM Instance to the ORAM Server for construction tended to starve the requests of actual queries (much fewer in number), causing the query rate to drop quickly as more levels were introduced. This behavior was corrected by forcing level construction to remain proportional to query progress: this keeps the individual query rates much closer to their average.

Due to the impracticality of repeatedly running trials over the entire (up to 1TB) measured database epoch, the database for these trials is first constructed non-obliviously on the server via a specially designed module. The items are inserted in a random order so that the final result mirrors an oblivious construction (as would occur from a sequence of write queries).

6.4.2 Setup

PD-ORAM is written in Java. Clients are distributed evenly across several Intel Xeon machines, each with two quad-core 3.16GHz Xeon X5460's. The Server runs on a single Quad-Core Intel i7-2600K Sandy Bridge 3.4GHz (3.8GHz Turbo Boost) LGA 1155 95W CPU, with 16GB DDR3 1600 (PC3 12800) SDRAM and seven HITACHI Deskstar 2TB 7200 RPM SATA 3.0Gb/s 3.5" SATA disks (software RAID0/LVM).

All the machines share a gigabit switch. Network latency is shaped by forcing server threads to sleep for the desired round trip duration upon receiving a request. This allows simulation of link latency without capping link bandwidth.

The implementation uses a BF with 8 hash functions, and 2400 bits of space per item which allows an efficient construction within the false positive rate of 2^{-64} per lookup. The resulting BF constitutes roughly 25% of the database records size.

Optimization. Rather than optimizing the BF size required to obtain this error rate by using a larger number of hashes, as suggested by Pinkas and Reinman [108], PD-ORAM uses larger BFs with fewer hashes, to minimize item lookup disk seeks while obtaining the same error rate.

6.4.3 Experiments

One main goal of the experiments is to understand the interaction between network performance parameters and the parallel nature of PD-ORAM.

Size + clients vs. query throughput + latency. Figure 6.7 plots the effect of database size and client parallelization on overall query throughput and latency. Fresh databases were used for all trials to prevent dependency of the measurements on the order of the trials, except for the 1TB trials, where this proved impractical. Even though individual query

latency increases with more clients contending for resources, the benefit of parallelization are obvious: significantly higher overall throughputs is achieved.

Clients + network latency vs. performance. Figure 6.8 plots the effect of parallel clients and network latency on overall query throughput and per-client latency for a fixed database size. The premise of this measurement is that parallelization becomes significantly more important as network latency increases.

De-amortization optimality. Figure 6.9 plots the observed latency of individual queries vs. time on a growing database. With perfect de-amortization, all queries would require the same amount of time. The majority of the queries take around 1200ms; a fixed lower limit is imposed by the network latency. The bands at 2600ms and 3100ms reflect the construction of the top, smallest level, which is not de-amortized. The slight downward slope indicates the progress estimation at the beginning of the shuffle is conservative.

Progress metering. To validate the accuracy of the progress metering, Figure 6.10 shows the reported construction progress of a single level as sampled every 5 seconds. Strict de-amortization and querying is disabled for this trial, to avoid cool-down periods when construction has progressed farther than is needed, and to ensure measurement of its progress only.

6.4.4 Impact of Rotational Hard Disk Speeds

The experiments were repeated (for database sizes up to 300GB) in a different setup, in which the server was run on dual 3.16GHz Xeon X5460 quad-core CPUs and *six 0.4 TB 15K RPM SCSI (hardware RAID0)* disks.

This configuration surprisingly outperformed the setup above by a factor of 2x in most trials. The primary advantage is the superior seek time on the server disks, so the markedly different results suggest that server disk seek costs play an important role in overall performance. This was somewhat surprising, since the level construction mechanisms were designed specifically to minimize disk seeks (with the hash table insertion being the only random-access operation during level construction, requiring an average of 2 random writes per insert). The rest of the level construction simply requires reading from one or two sequential buffers, and writing out sequentially to one or two.

The culprit is most likely the de-amortization process, which constructs different levels in parallel, and in effect randomizes disk access patterns. While individual level construction is mostly limited by sequential disk throughput, running many of these processes in parallel across the same file system results in disk seeks even in the sequential access regions, resulting in a much lower overall disk throughput.

Several software and hardware solutions present themselves. Better data placement would split data in a more efficient manner across the available disks (instead of using a RAID configuration), to allow the sequential nature of each level construction process to transfer to sequential disk access. Obtaining optimal throughput in this manner would require a relatively large number of disks. Further, the use of more expensive (but quickly dropping in cost) low-latency solid state disks (SSDs) would be a simple hardware solution to eliminate

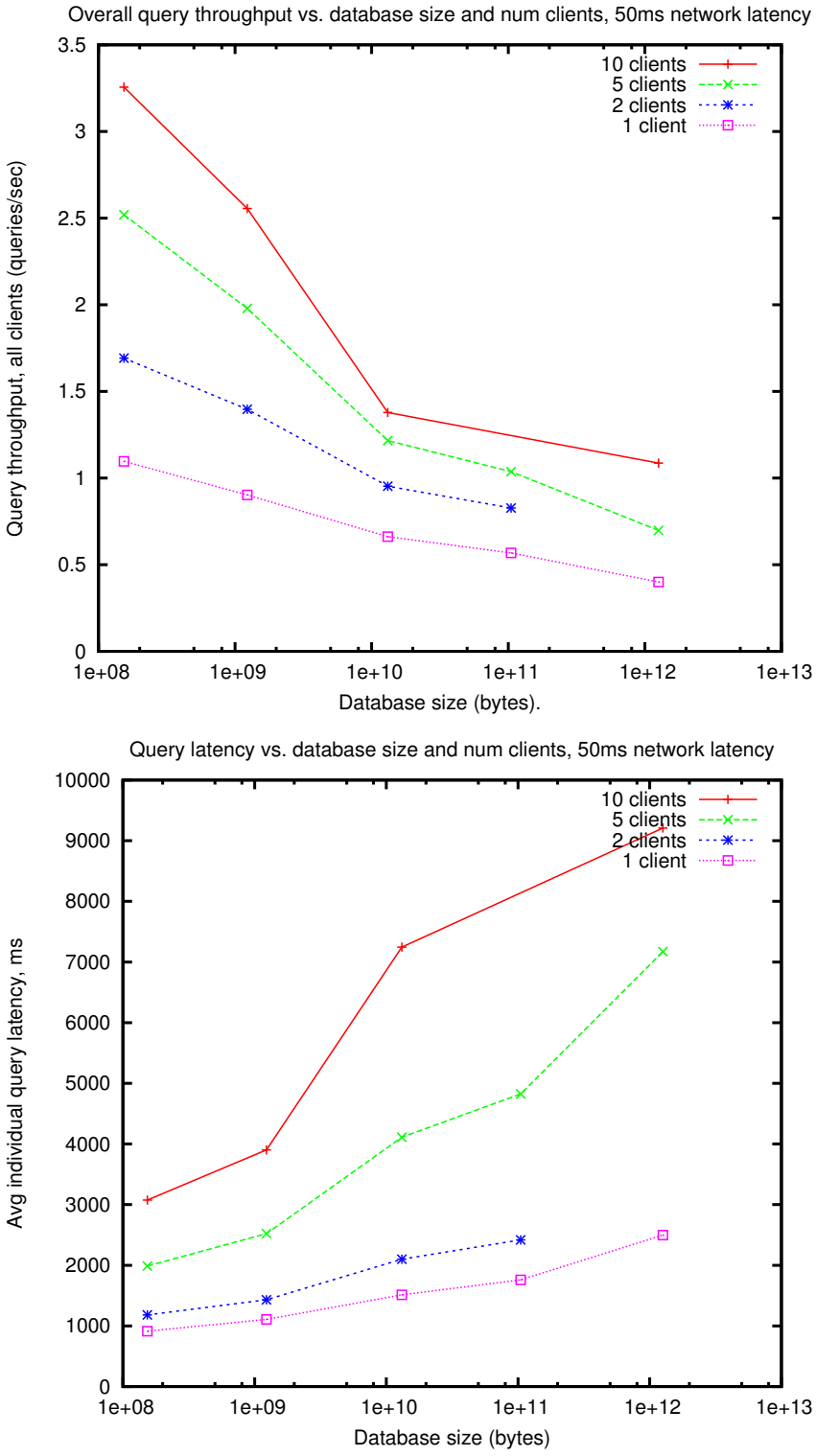

Figure 6.7: Performance vs. data size for varying number of clients. Top: throughput vs. database size. x-axis is log scale. Bottom: query latency vs. database size. Each point is sampled over 3000 queries.

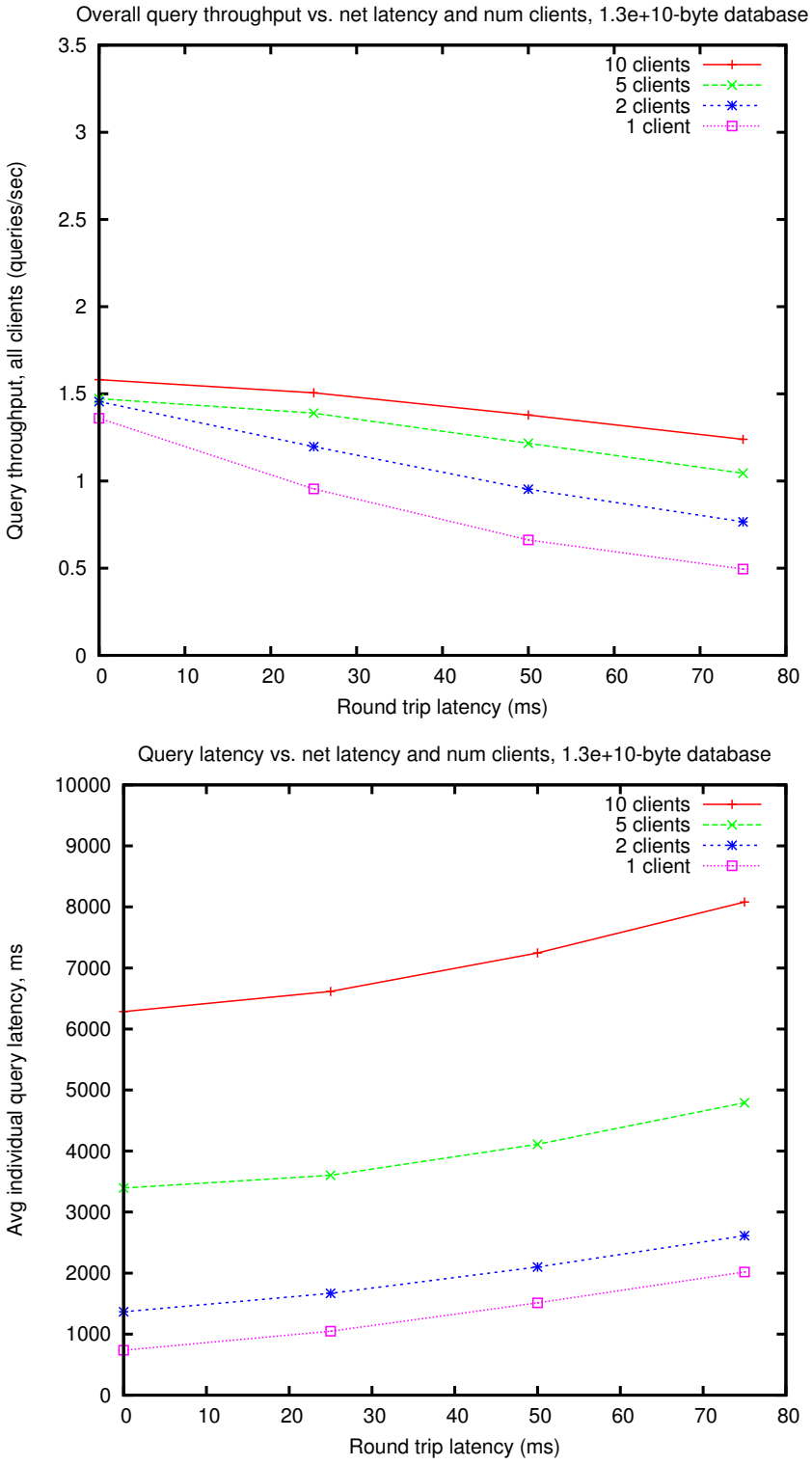

Figure 6.8: Performance vs. network latency for varying number of clients. Top: query throughput vs. network latency. Bottom: query latency vs. network latency. Each point is sampled over approx. 3000 queries.

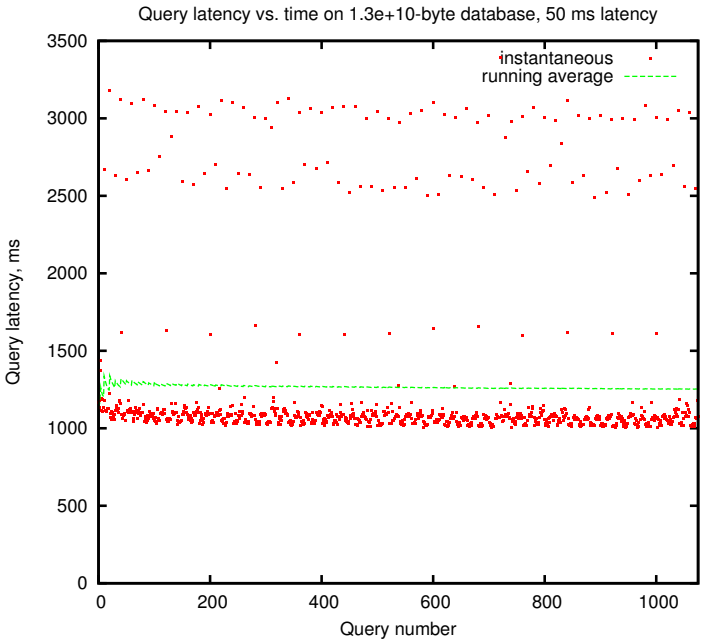

Figure 6.9: Individual query timings. Query latencies of a trial using a single client are plotted, along with the running average. With perfect de-amortization, all queries would require the same amount of time.

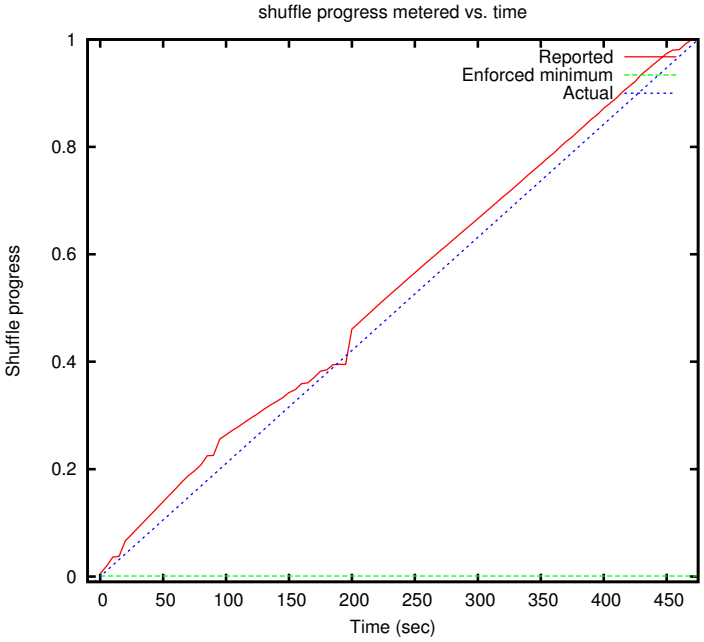

Figure 6.10: Level shuffle progress over time: The linear nature of the plot indicates the extrapolation based on partial shuffling is well done.

this performance bottleneck. It remains to be seen however, whether the sustained random write performance degradation plaguing current SSDs does not constitute a bigger bottleneck in itself, as preliminary throughput experiments on several recent 128GB Samsung SSDs with 2011 firmware updates seem to suggest.

6.5 An Oblivious File System

ORAM lends itself naturally to the creation of a block device. Due to existing results' impractical performance overhead this has not been previously possible. A Linux-based deployment of PD-ORAM is used here to design and build *privatefs*, a fully-functional oblivious network file system in which files can be accessed on a remote server with computational access privacy and data confidentiality.

An initial implementation was built on top of the Linux Network Block Device (NBD) driver, which is the simplest and most natural approach, since PD-ORAM already provides a block interface. However, NBD supports only serial, synchronous requests. To take advantage of the parallel nature of PD-ORAM, privatefs is instead built on FUSE (Filesystem in Userspace [56]).

A second attempt used *ext2fuse*, a FUSE-based ext2 implementation [112], by rerouting block access through PD-ORAM. However, thread safety difficulties prevented us from modifying it to support parallel writes or reads. Additionally, because of its nature as a block device file system, *ext2fuse* requires mechanisms for allocating blocks for files, such as block groups, free block bitmaps and indirect file block pointers inside inodes. These mechanisms are not all thread-safe and pose a challenge to synchronize. Moreover, locking the code using synchronization primitives would not result in a sufficient degree of parallelization.

Instead, with the help of Alin Tomescu, we implemented our own *privatefs* using the FUSE libraries in C++. It fully leverages the parallelism of PD-ORAM. Moreover, it takes advantage of the non-contiguous block labeling of PD-ORAM in a way that block-device file systems cannot.

Following the Linux file system model, in *privatefs* files are represented by inodes. Directories are inodes containing a list of directory entries; each directory entry is the name of a file or subdirectory along with its inode number. Inodes are numbered using 256-bit values and are mapped directly to ORAM blocks, such that inode x is stored in ORAM block x. Inodes hold metadata such as type, size and permissions. Both *privatefs* and (this instance of) PD-ORAM use 256-bit block identifiers and 4096-byte blocks.

Because the ORAM provides random access to 256-bit addressable blocks, a block can be allocated simply by generating a random 256-bit number. We take advantage of this in two ways. First, to read or write the i^{th} block of file with inode number x, the pair (x, i) is hashed with the collision-resistant SHA256 hash, yielding the 256-bit ORAM block ID for that file block. Second, when a new file is created, a 256-bit inode number is randomly generated, as opposed to maintaining and synchronizing access to an inode counter.

Our design eliminates the complexity of contiguous block device file systems and minimizes the need for locking when writing or reading files. As opposed to *ext2fuse*, *privatefs*

File System	Write speed	Read speed
NBD (ext2)	14.80KB/s	10.45KB/s
ext2fuse	7.11KB/s	9.62KB/s
privatefs	83KB/s	61.69KB/s

Figure 6.11: Measured file system performance in a zero latency environment for our two trial implementations, plus our *privatefs* implementation.

does not incur the overhead of maintaining free block or inode bitmaps, grouping blocks into block groups, or traversing indirect block pointers to read files. The potential drawback is that sequential blocks of a given file will not be stored contiguously in the file system. However, this is harmless when using an ORAM, since there is no notion of sequential block numbers (which would compromise access privacy).

privatefs employs exclusive locks when reading and writing directories. In addition, an LRU cache is implemented to quickly retrieve an inode's data given its inode number and also for file path to inode number translation, which helps avoid long directory traversals (and associated locking). *privatefs* communicates with the ORAM server by means of a proxy (written in Java), which receives block requests from the file system and satisfies them using parallel connections to the ORAM server. This design choice affords us a higher degree of modularity, enabling us to connect *privatefs* to other ORAM schemes in the future. This arrangement is illustrated in Figure 6.13.

We benchmarked privatefs along with our previous file system attempts, using a parallel workload writing ten, 512KB files simultaneously to the file system, then checking their integrity (also done simultaneously) using the `sha256sum` utility. The set of tests in Figure 6.11 considered a 0-latency environment. The performance results for *privatefs* indicate a major improvement relative to our previous implementations using NBD and *ext2fuse*.

Our benchmarks (Figure 6.12) indicate that *privatefs* is benefiting from the high degree of parallelism in high-latency environments. We ran five trials for each point, varying the number of ORAM clients used by the proxy to satisfy file system block requests and performance increased proportionally with the degree of parallelism.

Reads are less expensive than writes in the 40ms and 160ms trials, because the individual writes are performed synchronously, while the reads can be parallel, resulting in a higher overall degree of parallelism. On the other hand, the low latency in the 0ms trial prevents this parallelism from having an impact. The reads are, in turn, more expensive, due to inefficiencies in the de-amortization method resulting in slightly slower queries at later points in the process.

Overall, *privatefs* features a throughput modest when compared to unsecured file systems. However, this is the inherent cost of achieving privacy. *privatefs* is the first file system to provide access pattern privacy, and is fully functional and immediately usable.

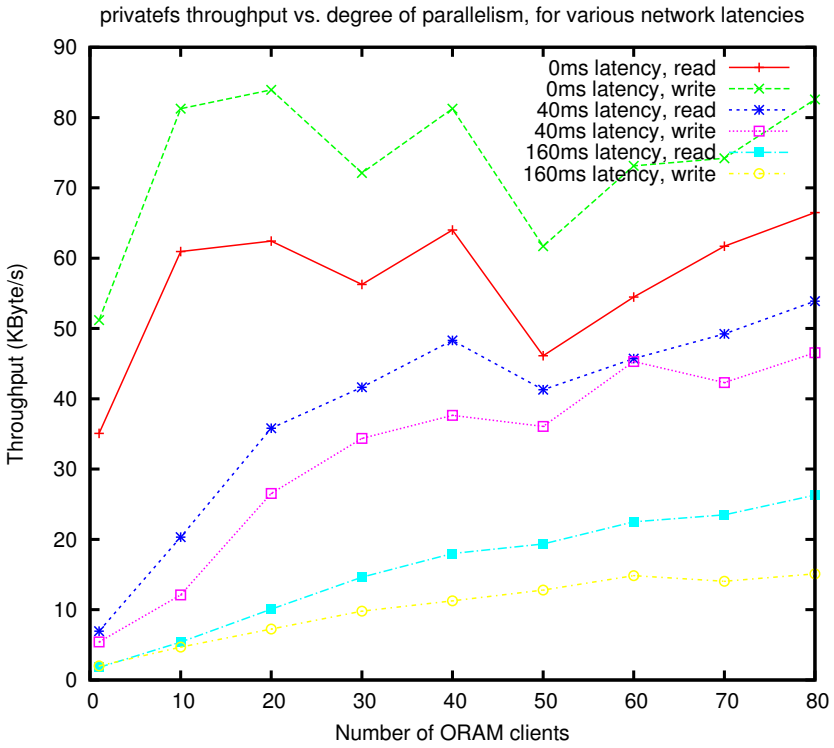

Figure 6.12: *privatefs* read and write performance vs. the number of ORAM Clients, for various network latencies. The zero latency environment is mostly unaffected by the degree of parallelism, as expected. In higher latency environments, the throughput grows with the number of clients; having parallel clients offsets the effect of the latency.

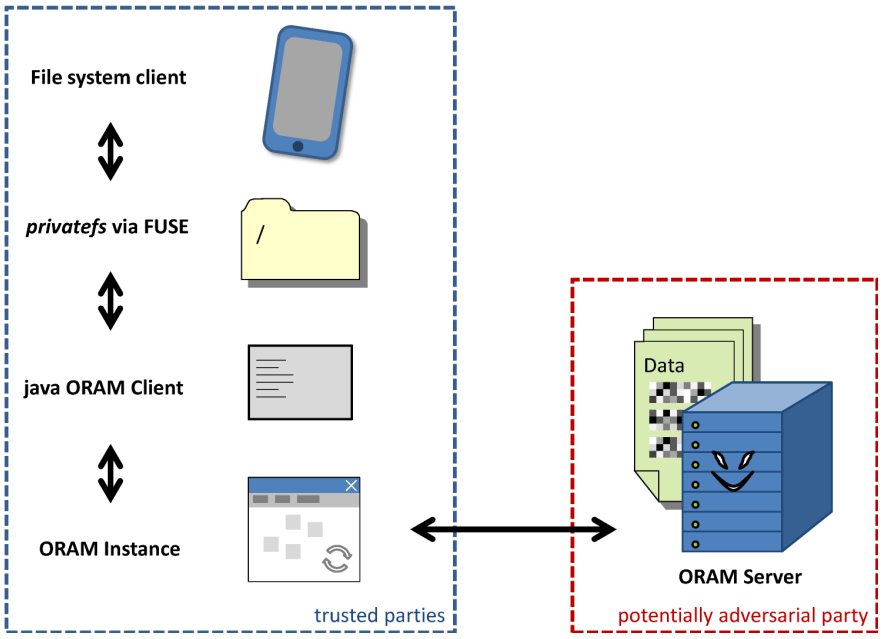

Figure 6.13: Implementing an Oblivious File System: the ORAM Client provides a virtual block interface. A FUSE-based file system converts reads and writes into sequences of block requests. The result is a parallel-access oblivious file system, wherein the data is stored remotely but accessed with access privacy and data confidentiality.

6.6 Conclusions

The contributions of this chapter include mechanisms for secure parallel querying of existing ORAMs to increase throughput, a generalization of ORAM de-amortization, a logarithmic-storage construction of a Bloom filter-based ORAM, and implementation of an efficient ORAM based on these techniques, performing a transaction per second on a 1TB database in an average-latency network, a first. Finally, an implementation of *privatefs*, the first oblivious networked file system, is provided.

Chapter 7

ORAM in a Single Round Trip

I now present SR-ORAM, the first single-round-trip poly-logarithmic time Oblivious RAM requiring only logarithmic client storage. Taking only a single round trip to perform a query, SR-ORAM has a communication/computation cost of $O(\log n)$, with $O(\log^2 n \log\log n)$, and under 2 round trips, overall amortized per-query communication requirements. The trusted client folds an entire interactive sequence of ORAM requests into a single query object that the server can unlock incrementally, to satisfy a query without learning its success status. SR-ORAM additionally defends against an actively malicious polynomially bounded adversary.

7.1 Introduction

One of the most significant challenges to providing practical ORAM is that these interactive protocols require a large number of client-server round trips, resulting in large, often impractical, online query latencies. For example, a recent construction by Goodrich et al. [45] requires $\log_2 n$ round trips, translating to an online cost alone of over 1200-1500ms per query on a 1 terabyte database (e.g., for 10KB blocks), assuming a network link with a latency of just 50ms.

This chapter provides a simple and direct solution to the challenge: SR-ORAM, a non-interactive, single-round-trip ORAM. SR-ORAM requires a single message to be sent from the client to the server and thus incurs a single round-trip (for a total online cost of 50ms in the example above). Moreover, SR-ORAM does not greatly affect the offline, amortized cost.

The basic idea behind SR-ORAM is to fold the interactive queries into a single non-interactive request without sacrificing privacy. The client constructs a set of values (a "query object") that allows the server to selectively decrypt pieces, depending on new values obtained during its traversal of the database. Critically, each component of the query object unlocks only a specific single new component—which allows server database traversal progress while preventing it from learning anything about the overall success of the query.

My construction is based on the Bloom filter ORAM from Chapter 5, since it lends itself

conveniently to use of a non-interactive query object and provides defenses against actively malicious adversaries (not only curious). We also make use of the randomized shell sort [46], since it allows the more stringent client storage requirements of SR-ORAM (when compared to the sorts I present in 3).

Other non-interactive ORAMs exist; Chapter 2 reviews recent solutions. However, SR-ORAM is the first to provide a non-interactive *poly-logarithmic time* construction that assumes only *logarithmic client storage*.

Bloom filters. As in the other Bloom-filter based ORAMs I present (Chapters 5 and 6), the SR-ORAM Bloom filters are constrained by important considerations. First, we need to minimize the number of hash functions h, since this determines directly the number of disk reads required per lookup. Second, we need to guarantee that with high probability, there will be no false positives, since a false positive reveals a lookup failure to the curious server. Third, as will be shown, in SR-ORAM, instead of storing an encrypted bit for each position of the Bloom filter, we now store part of a decryption *key*. Since the server cannot distinguish between the keys for bit-values of *1* and keys for bit-values of *0*, we retain the property that the server does not learn the success of the Bloom filter lookup.

7.1.1 Model

Unlike previous chapters, we assume here that the client has only enough local non-volatile storage to manage keys and certificates, plus enough volatile RAM to run the ORAM client software (logarithmic in the size of the outsourced data).

We will assume in this chapter, and defend against, an actively malicious, curious polynomially bounded adversary in the random oracle model. The actively malicious defense is inherited from the underlying ORAM.

Additional Notation. The client secret key is sk. The number of times a given level has been shuffled (i.e. reconstructed) is called the "generation," and is abbreviated as *gen*. Key size and hash function output size are both assumed to be c_0; c_1 is the Bloom filter security parameter.

7.1.2 Other Single-Round-Trip ORAMs

While Chapter 2 reviews recent ORAM related work, they are not examined in the context of the number of round trips required. We will revisit in this section the work that is relevant to reducing the number of round trips.

Indeed, other recent approaches provide ways around the penalty of highly interactive protocols, at the cost of additional hardware or overwhelming requirements of client storage. The main issue in constructing a single round trip ORAM is that a request for an item depends on how recently an item was accessed. Maintaining this information at the client requires storage at least linear in the size of the outsourced database. Moreover, retrieving this information privately from the server is almost as difficult as providing ORAM. (With

the difference that this recursive ORAM only requires storing $O(\log \log n)$ bits per item, which is enough location information about the item to build the query).

Secure Hardware. Secure Hardware modules such as the IBM 4764 [59] can be placed server-side, while using remote attestation to retain some level of security guarantees for clients [5]. Secure hardware, however, is typically an order of magnitude more expensive than standard processors, and due to heat dissipation difficulties, is typically also an order of magnitude slower. Moreover, the necessity of physical security in order to provide any guarantees makes such solutions vulnerable to a whole class of new attacks.

Non-interactive protocols using client storage. A new construction by Stefanov et al. [120] maintains item location information at the client. Although at the outset, $n \log_2 n$ bits of client storage seems like a big assumption, the authors argue this is reasonable in some situations, since the block size is typically larger than $\log n$. They show that in practice, the local required client storage in practice is only a small fraction of the total database size. The recursive construction, using a second ORAM to store this level membership information, however, is interactive. SR-ORAM requires only $O(\log_2 n)$ bits of client storage (Section 7.3).

A non-interactive cache-based ORAM by Wang et al. [125] relies on k client storage to provide an amortized overhead of $O(n/k)$. The idea is to take add previously unseen items to a cache which is shuffled back into the remote database when it fills. Again, the high client storage requirements (and poor storage/performance trade-off) make it unsuitable for my model. This idea is revisited under different assumptions by Boneh et al. [17], with better security formalization, but still requiring this client storage.

7.2 A First Pass

This strawman construction modifies the Bloom filter ORAM of Chapter 5. This strawman construction has the structure, but not yet the performance, of the SR-ORAM construction. As detailed in Section 5.1, that Bloom filter ORAM uses encrypted Bloom filters to store level membership of items. To seek an item, the querying client must request a known fake item from each level, except from the level containing this item: the item is requested here instead. Which level the item is at depends only on how recently this item was last accessed. Since the client does not have storage to keep track of that, it checks the Bloom filters one at a time to learn if the item is at each level.

Moreover, since the main principle of level-based ORAMs requires *each item be sought once per level instance*, it is unsafe to query the Bloom filters past the level where this item is present. This explains why the checks must be interactive: once the item is found at level i, further accesses at the levels below ($i + 1$ through $\log_2 n$) entail only random Bloom filter queries corresponding to fake item requests. Then, putting the found item back at the top of the pyramid guarantees that later, it will be sought and found elsewhere, since the only way it gets back down to the lower levels is by riding a wave of level reshuffles.

I now describe how to safely turn this into a non-interactive process. Observe that in an interactive ORAM, if the client is requesting a recently accessed item j that happens to be in level 2, the access sequence will proceed as follows. This example is also illustrated in Figure 2.1.

1. The client checks the level 1 Bloom filter for the item: reading the positions generated by Hash($sk \mid level = 1 \mid gen \mid j$)

2. Upon seeing Encrypt(*0*) at one or more of those positions in the Bloom filter, the client learns the item is not at level 1. So it asks for a fake item instead, that is labeled as Hash($sk \mid level = 1 \mid gen \mid$ "fake" $\mid accesscount$)

3. The client now checks the level 2 Bloom filter for the item: reading the positions indicated by Hash($sk \mid level = 2 \mid gen \mid j$)

4. Seeing Encrypt(*1*) at all of those positions, the client learns the item is at this level. This means it is safe to request the item here; the client asks for the block labeled Hash($sk \mid level = 2 \mid gen \mid$ "block" $\mid accesscount$)

5. Having already found the item, to maintain appearances and not reveal this fact to the server, the client continues to issue random Bloom filter lookups at each level i below. At each level it requests the fake blocks labeled Hash($sk \mid level \mid gen \mid$ "fake" $\mid accesscount$)

Note that there are only $\log_2 n$ possible such access sequences, based on which level the item is found at (Figure 7.1). Each path starts with a real query. Real queries continue until an item is found, at which point only fake queries are issued from there on down. This limited number of possible sequences makes non-interactive querying possible.

Since we have a finite number of these paths, our goal is to follow one of these paths non-interactively, not knowing ahead of time which level the item is at (and thus which of the $\log_2 n$ paths will be followed).

To achieve this, I propose to have the Bloom filter results themselves be used in unlocking one of the two possible edges leading to the next query. A successful lookup will unlock the edge leading to a "finished" set, under which only fake queries will follow. Conversely, failure must unlock the edge continuing down the "active" search set. Once in the "finished" set, it is impossible to return back to the "active" set. Most importantly, the server must not gain any ability at identifying which path it is currently on.

One strawman idea, exponential in the number of Bloom filter hashes k, is to *make each bit in the Bloom filter a piece of a decryption key* unlocking an edge to the next node. For each level, the client prepares 2^k results, corresponding to each possible state of the Bloom filter. The Bloom filter keys are generated deterministically by the client using a cryptographic hash, so that the client can efficiently keep track of them with only logarithmic storage. That is, a bit set to *1* at position *pos* in the Bloom filter is represented by T_{pos} = Hash($sk \mid pos \mid$ level $\mid$ gen $\mid$ 1), and a bit set to *0* by F_{pos} = Hash($sk \mid pos \mid$ level $\mid$ gen $\mid$ 0). The server learns only one of the two (*never both*).

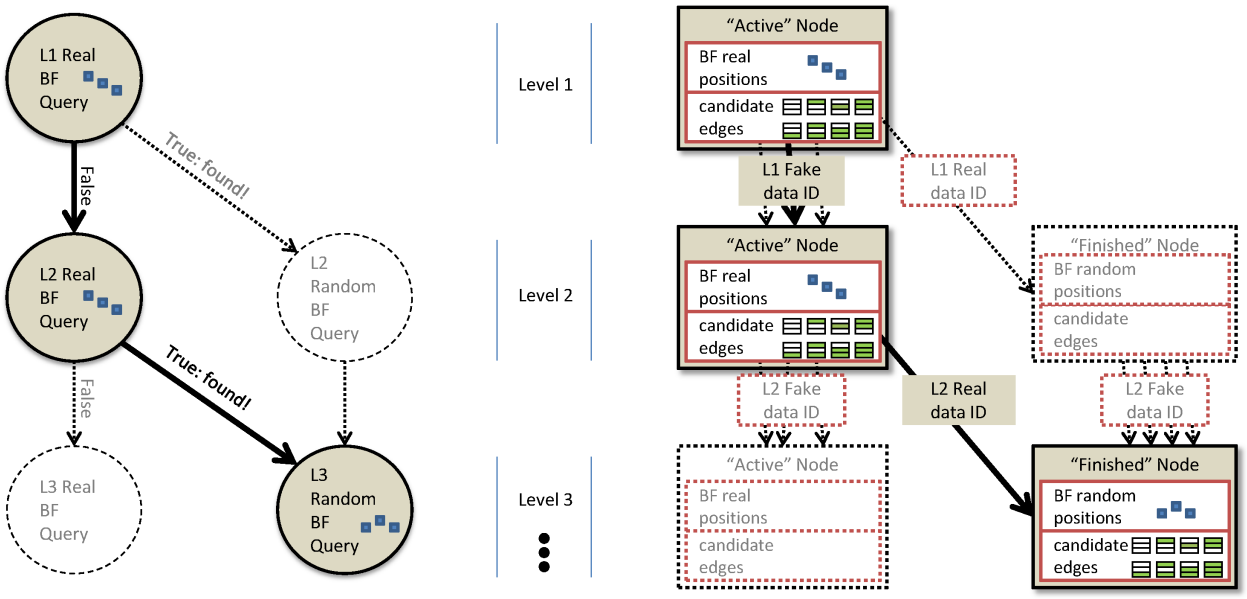

Figure 7.1: Left: potential query paths in unmodified BF-based ORAM. The client does not know at the time of querying which of the $\log n$ possible paths will be taken; it depends on where the data is ultimately found. In this example, the level 1 BF lookup returns false, indicating a fake should be retrieved from level 1. The level 2 BF lookup returns true, indicating the item should be retrieved from level 2. A fake BF query is run for level 3 since the item has already been found. Right: the query object in SR-ORAM. The server learns the edges corresponding to exactly one path. The server will be able to decrypt one such edge at each level, revealing the data ID, to retrieve and include in the response to the client, and decrypting a node of the query object in the level below.

A Bloom filter lookup involves k bit positions (k is the number of underlying Bloom filter hash functions). For each new level it traverses, the server needs to know the k associated Bloom filter bit positions to retrieve, constituting this level's query. *For the first level, these are provided by the client.* For each successive level, the server will get this information by incrementally decrypting portions of a client-provided "query object" data structure.

Illustrated in Figure 7.1 (right), the "query object" is composed of $\log n$ levels and is traversed by the server top-down synchronized with the traditional ORAM traversal. The query object allows the server to progress in its database traversal without learning anything.

Each level in the query object (with the exception of the root), contains two nodes: a "finished" node and an "active" node. Each node contains the k positions defining the current level Bloom filter query. The nodes also contain a "keying" set of 2^k elements.[1]

After performing the Bloom filter lookup, the server will be able to decrypt one of these elements (only). Once decrypted, this element contains a key to decrypt one of the query object's next level two nodes; it also contains the identifier for a current level item to return to the client. To prevent leaks, the server will be asked to return one item for each level, since we do not want to reveal when and where we found the sought-after real item.

In effect this tells the server where to look next in the query object—i.e., which of the query object's next level two nodes ("finished" or "active") to proceed with. This guides the server obliviously through either the "finished" or the "active" set, as follows:

- If the current level contains the sought-after item, the server's work is in fact done. However, the server cannot be made aware of this. Hence, it is made to continue its traversal down the ORAM database, via a sequence of fake queries. The "finished" node of the next query object level allows the server to do just that, by providing the traversal information down the "active" set.

- If, however, the current level *does not* contain the sought-after item, the server must be enabled to further query "real" data in its traversal down the ORAM database—it will thus receive access to "active" node of the next query object level.

To prevent the server from decrypting more than one element from a node's "keying" set, a special encryption setup is deployed. Each of the 2^k elements of the "keying" set is encrypted with a special query object element key (QOEK), only one of which the server will be able to reconstruct correctly after its Bloom filter query.

More specifically, for a Bloom filter lookup resulting in k bit representations (i.e., bit_i is the representation of the bit at position i – either T_i or F_i [2]), the QOEK is defined as QOEK = Hash($bit_1 \mid bit_2 \mid bit_3 \mid ... \mid bit_k$).

The encryption setup of the "keying" set ensures that this key decrypts exactly one of its elements. The element corresponding to a Bloom filter "hit" (the sought-after element

[1] After encryption, these elements are sent in a random order to prevent the server from learning any information.

[2] Recall that bit_i does not reveal to the server anything about the actual underlying Bloom filter bit since the server does not know the hash key sk.

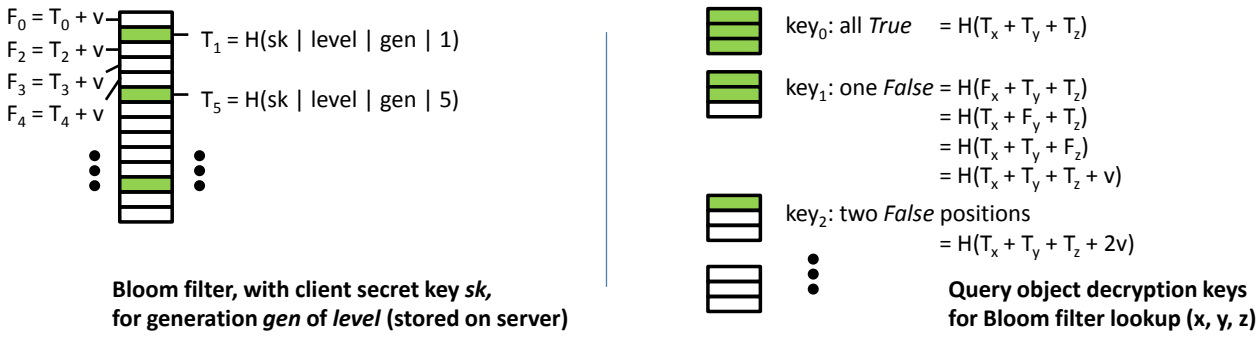

Figure 7.2: Left: Bloom filter format. A bit set to *1* in the Bloom filter is represented by a hash of the position and current level key; a bit set to *0* is represented by the same value, plus the secret value v. Right: decryption keys used with the query object. The server obtains the decryption key for a given query stage by hashing together the specified Bloom filter results. Since there are h hash functions used in a Bloom filter check, the client includes in the query object an edge corresponding to each of the $h + 1$ possible keys.

was found at this level, i.e., all the underlying Bloom filter bits are set to *1*) leads down the "finished" set, i.e., the element that QOEK decrypts now, leads down the "finished" set in the query object's next level.

7.3 Efficient Construction

I now present an efficient construction, with only $O(\log n)$ client storage, $O(\log n)$ per-query online message size, $O(\log^2 n \log\log n)$ amortized communication, and still only $O(1)$ round trips. We reduce the size of the query object of Section 7.2 from $2^h \log n$ to just $h \log n$.

The main insight is to allow compression of the 2^h decryption key possibilities into only $h+1$ possibilities. This is achieved by representing the Bloom filter bits and their combination in a commutative format. By allowing the decryption key pieces stored in the Bloom filter (described in the previous section) to be added together, rather than concatenated, the client only has to account for $h + 1$ different outcomes at each level.

To this end, let us start by first establishing a secret level-instance-specific token $v = \text{Hash}(sk \mid level \mid gen)$, not known to the server. In the Bloom filter, a bit set to *1* at position pos is represented as $T_{pos} = \text{Hash}(sk \mid level \mid gen \mid pos)$; a bit set to *0* is represented as $F_{pos} = T_{pos} + v \mod 2^{c_0}$ (Figure 7.2), where c_0 is a security parameter. In the following we will continue to operate in $\mathbb{Z}_{2^{c_0}}$. We also assume that $\text{Hash}(\cdot)$ operates in $\mathbb{Z}_{2^{c_0}}$.

Now, the query object encryption key (QOEK) is generated by combining (adding) values using modular arithmetic, instead of concatenation as in the strawman solution. This allows the client to only account for $h + 1$ possibilities, each key corresponding to the number of times v might show up among the selected Bloom filter positions.

The server sums together the values found in the Bloom filter, and performs a hash, yielding, e.g., $\text{Hash}(bit_1 + bit_2 \ldots + bit_h \mod 2^{c_0})$ as the QOEK. The commutativity of

modular addition means each permutation of bits set to *0* and bits set to *1* in a given Bloom filter yields the same key. A successful Bloom filter lookup occurs in the case that the QOEK is Hash($T_{pos_0} + T_{pos_1} + ... + T_{pos_h} \mod 2^{c_0}$), which unlocks the edge from the real to the fake path.

Further, each of the h values from Hash($T_{pos_0} + T_{pos_1} + ... + T_{pos_h} + v \mod 2^{c_0}$) through Hash($T_{pos_0} + T_{pos_1} + ... + T_{pos_h} + hv \mod 2^{c_0}$)—the result of Bloom filter lookup failures (the server does not know they are failures)—unlocks an edge in the query object that results in continuing on the current (fake or real) path (Figure 7.1). Figure 7.3 provides a summary of the query object format.

Let us now consider an example from the perspective of the server. Say the client sends a query object for an item x. This query object contains a single Level 1 node in cleartext, and two nodes for every level below. The Level 1 active node tells it to query positions $L1pos_1$ ... $L1pos_k$. The server retrieves the values stored in the Bloom filter at these locations, adds them modulus 2^{c0}, and applies the one-way hash function. This yields a decryption key. The server now tries this key on all k encrypted values included in the Level 1 node. It finds one that successfully decrypts, revealing a data ID to retrieve from Level 1, as well as the decryption key for one of the two Level 2 nodes. The server appends the retrieved Level 1 data item to its result, then decrypts the Level 2 node that it has the key for.

The server now repeats for the Level 2 node. It finds a list of Bloom filter positions, which it again retrieves, adds modulus 2^{c0}, and hashes, yielding a decryption key which it tries on the encrypted values included in the Level 2 node. Again, only one will decrypt. The server never learns which are the active or finished nodes; it simply sends back $\log_2 n$ data values to the client, of which one will be a real item, and the others fake items.

7.3.1 Obliviously Building the Bloom Filter and Levels

This section describes how to construct the Bloom filter without revealing to the server which Bloom filter positions correspond to which items. Further, it shows how to obliviously scramble the items as a level is constructed. Both processes are mostly non-interactive (only a constant small number of round trips).

In the Bloom filter construction, the key privacy requirement is that the server is unable to learn ahead of time any correlation between Bloom filter positions and data items. Note that, now, instead of storing single encrypted bits in the Bloom filter, components of decryption keys are stored. Recall that these components are computed on the client by a secure keyed hash of the position, and added to the value v if this position is intended to represent a bit value of *0*.

We use the storage-free Bloom filter construction method described in Section 5.1.7. Afterwards, the encrypted bit-storing Bloom filter needs to be converted into a key-storing Bloom filter in one final step: in a single non-oblivious pass, we read each bit and output either T_{pos} for *1*s and F_{pos} for *0*s (where *pos* is the current bit's position in the Bloom filter). Note this multiplies the size of the remotely stored object by the key size.

We do not need to otherwise modify the level construction from Chapter 5, except in replacing their storage-accelerated merge sort with the storage-free randomized shell sort.

- Level 1:
  - L1 active node, in cleartext:
    - Level 1 Bloom filter lookup index positions $L1pos_1 \dots L1pos_k$ (integer values)
    - the client computes, *but does not send*, these $k+1$ keys:
      - $key_{L1,\text{ success}} = \text{Hash}(T_{L1pos_1} + T_{L1pos_2} + ... + T_{L1pos_k})$
      - $key_{L1,1} = \text{Hash}(T_{L1pos_1} + T_{L1pos_2} + ... + T_{L1pos_k} + v)$
      - $\dots$
      - $key_{L1,k} = \text{Hash}(T_{L1pos_1} + T_{L1pos_2} + ... + T_{L1pos_k} + kv)$
    - $k+1$ encrypted values, included in a random order:
      - $E_{key_{L1,\text{ success}}}$ (L1 **real** data ID, and key2F: the key for the L2 finished node)
      - $E_{key_{L1,1}}$ (L1 fake data ID and key2A: the key for the L2 active node)
      - $\dots$
      - $E_{key_{L1,k}}$ (L1 fake data ID and key2A: the key for the L2 active node)
- Level 2: Both the L2 active and the L2 finished nodes, in a random order:
  - L2 active node, encrypted with randomly generated key2A:
    - Level 2 Bloom filter lookup index positions $L2pos_1 \dots L2pos_k$
    - the client computes, *but does not send*, these $k+1$ keys:
      - $key_{L2,\text{success}} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + ... + T_{L2pos_k})$
      - $key_{L2,1} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + ... + T_{L2pos_k} + v)$
      - $\dots$
      - $key_{L2,k} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + ... + T_{L2pos_k} + kv)$
    - $k+1$ encrypted values, included in a random order:
      - $E_{key_{L2,\text{success}}}$ (L2 **real** data ID, and key3F)
      - $E_{key_{L2,1}}$ (L2 fake data ID and key3A)
      - $\dots$
      - $E_{key_{L2,k}}$ (L2 fake data ID and key3A)
  - L2 finished node, encrypted with randomly generated key2F:
    - k random Bloom filter lookup index positions for Level 2 $L2pos_1 \dots L2pos_k$
    - the client computes, *but does not send*, these $k+1$ keys:
      - $key_{L2,0} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + ... + T_{L2pos_k})$
      - $key_{L2,1} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + ... + T_{L2pos_k} + v)$
      - $\dots$
      - $key_{L2,k} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + ... + T_{L2pos_k} + kv)$
    - $k+1$ encrypted values, included in a random order:
      - $E_{key_{L2,0}}$ (L2 fake data ID and key3F)
      - $E_{key_{L2,1}}$ (L2 fake data ID and key3F)
      - $\dots$
      - $E_{key_{L2,k}}$ (L2 fake data ID and key3F)
- Both the L3 active node (encrypted with key3A) and the L3 finished node (encrypted with key3F), in a random order.
- And so forth, for each of the $\log_2 n$ levels.

Figure 7.3: Query Object Format. For each level, the query object is composed of two possible nodes (with the exception of the root/top which only has one node), and is constructed thusly. This set constitutes a total of $2 \log n$ nodes (containing associated Bloom filter requests), and $2h \log n$ edges. Of these nodes, the server will be able to unlock $\log n$. Each of the unlocked ones provides h edges, of which the server will be able to unlock exactly one. This contains the decryption key for a single node at the next level, as well as the data item ID to retrieve etc.

Non-interactivity of sort. It is worth noting that the shell sort, like the storage-accelerated merge sort, is mostly non-interactive. Each step in both sorting algorithms requires reading two items from the server, and writing them back, possibly swapped. However, the item request pattern (of both sorting algorithms) is known ahead of time to the server, so it can send the answer back to the client without waiting for the request from the client. This process is trivial in the merge sort: the access pattern consists of simultaneous scans of two (or sometimes up to $\sqrt{n}$) arrays; the server streams these to the client. This process of non-interactive streaming of sort data to the client is not as trivial in the randomized shell sort. Once the random seed is chosen, however, both the client and server know in advance the order the items will be requested in. The server can run a simulation of this sort, for example, to know which items the client needs to read next, and avoid waiting for network round-trips throughout the construction of the level.

7.4 Security

SR-ORAM directly inherits the privacy properties of the server-side ORAM database traversal, as well as the integrity defenses from the base ORAM construction of Chapter 5. However we still need to establish the privacy of the query object construction, as well as the new Bloom filter construction.

I establish privacy of the query object construction in Theorem 25. The server learns only one set of Bloom filter positions and one item label to retrieve at each level for a given query. In other words, the server sees only what it would see in an equivalent, interactive instantiation.

Lemma 7. *The server gains no non-negligible advantage at guessing v = Hash (sk | level | gen) from observing (i) the Bloom filter contents or (ii) the hashes included in the query object.*

Proof. (outline) For each position *pos* in the Bloom filter, the server sees a single value X that is either a random number T_{pos}, or $T_{pos} + v \mod 2^{c_0}$. If T_{pos} and v are both chosen randomly from $\mathbb{Z}_{2^{c_0}}$, then seeing the entire set of values gives the server no knowledge of v.

There is one subtlety here: if the server has some external knowledge about how recently a given query was last issued, it can determine that the set of corresponding Bloom filter positions at that level are all *1*s (and thus, that the values observed at some positions are in fact $T_{pos} + v \mod 2^{c_0}$, rather than T_{pos}). However, knowing this still does not give the server any knowledge of v: all possible values of T_{pos} and v are still equally possible and likely.

The only other values observed by the server that are linked to v are the outputs of the one-way hash functions, included in the query objects. Seeing the hash outputs in the query object provides no advantage at determining the inputs, beyond using dictionary attacks which are infeasible for a computationally bounded adversary (in the security parameter c_0), since the inputs all include the random v, which is c_0 bits long. □

Theorem 25. *The server can only unlock one path down the query object, and all paths appear identical to the server.*

Proof. (outline) Each of the $2\log_2 n$ nodes in the query object, as illustrated in Figure 7.1, provides simply a list of Bloom filter indices, and a set of encrypted values unlocking edges to one of the two next nodes. The negligible Bloom filter false positive rate, established by the base Bloom filter ORAM, guarantees that the set of indices will be unique for every query. Moreover, these indices, chosen by a keyed secure hash, are indistinguishable from random.

Each edge contains an item label which is again uniquely requested, only up to once per level. The label is also determined by the keyed secure hash. Since the contents of any single edge for a given query is indistinguishable from random, and since these edges are included in the query object in a random order, unlocking the edge provides no information to the server about the path.

The contents of the Bloom filter provide the key and determine which one edge the server can unlock. Since, as shown in Lemma 7, the server has no advantage at determining the blinding value v, the server has no advantage at guessing the alternate value at any position of the Bloom filter, and it is limited to a dictionary attack against the ciphertexts of the other edges. Thus, the server can only unlock the edge corresponding to the contents of the Bloom filter. □

Theorem 26. *The server learns nothing from the Bloom filter construction.*

Proof. (outline) All Bloom filter construction processes for any particular level appear equivalent to the server, regardless of which bits are set. The server sees only the number of bits in the Bloom filter, which is known beforehand. □

Theorem 27. *An honest but curious adversary gains no non-negligible advantage at guessing the client access pattern by observing the sequence of requests.*

Proof. (outline) Because of Theorem 25, the adversary learns nothing it would not learn by observing a standard, interactive, Bloom filter-ORAM. We defer here to the security claims of previous ORAMs (e.g. Theorem 18, which shows that the probability of Bloom filter failure in each ORAM query is negligible in the considered security parameter. □

Theorem 28. *An actively malicious adversary has no advantage over the honest but curious adversary at violating query privacy.*

Proof. (outline) In the underlying Bloom filter ORAM from Chapter 5, the client detects server protocol deviation, preventing the server from learning anything new from issuing incorrect responses. The non-interactive construction creates a slight difference in the Bloom filter authenticity check: the Bloom filter correctness check is now implicit. That is, the server can only unlock the key if the stored value is the one placed by the client, whereas in previous Bloom filter ORAMs, the client had to test the authenticity of the stored Bloom filter bits before it was safe to continue the query. □

disk seek cost	0
disk throughput	400 MBytes/sec
server RAM	24GB
crypto throughput	200 MBytes/sec
net throughput	125 MBytes/sec
net round trip	50 ms

Figure 7.4: Assumed Hardware Configuration

Acceptable failure rate	2^{-128}
Item block size	10,000 bytes
Symmetric key size	256 bits
Bloom filter hashes	50

Figure 7.5: Database Parameters

7.5 Analysis

Following from the construction in Section 7.3, the query object size is $O(\log n)$. This is transmitted to the server, which performs $h \log n$ decryption attempts (of which $\log n$ are successful) before sending $\log n$ blocks back to the client. This yields the online cost of $O(\log n)$.

The amortized offline cost per query considers the time required to build each level. A level sized z is built twice every z queries. Shuffling these items using a randomized shell sort costs $O(z \log z)$. Since the Bloom filter is sized $z \log \log n$, and it must also be shuffled, the Bloom filter construction cost of $O(z \log z \log \log n)$ dominates asymptotically. Summing over the i levels sized $z = 4^i$, and amortizing over the queries for each level between rebuilding, we find a total amortized offline cost of $O(\log^2 n \log \log n)$

We also estimate the cost of querying and shuffling for a sample hardware configuration. The *online* cost is now limited by network, disk, and encryption throughput (instead of, e.g., network round trips in related work). The *offline* cost is limited, as in existing work, by network, disk, and encryption throughput.

To make the comparison between SR-ORAM and existing work as general as possible, we consider an *ideal storage-free interactive ORAM*, which includes just two core costs inextricably linked to any pyramidal storage-free ORAM, and ignores any other costs.

First is the online cost of retrieving an item from an ORAM interactively, requiring $\log_2 n$ round trips and the data transfer cost of $\log_2 n$ blocks. Second is the offline cost of obliviously constructing each level, assuming 2^i real items and 2^i fakes at each level i, using a randomized shell sort. Thus, this ideal storage-free interactive ORAM provides a lower bound for any existing ORAMs that do not assume super-logarithmic client storage. Existing work, including constructions by Goldreich and Ostrovsky [41], and Goodrich and Mitzenmacher [44], fall into the category of a storage-free interactive ORAM, but have significant other costs, pushing their complexity in fact much above this lower bound.

As discussed in Section 5.1.8, given an upper bound on acceptable failure (false positive) rate r, we can calculate the necessary Bloom filter size as a function of the number of hashes used for lookup h.

For example, for a Bloom filter false positive rate of $r = 2^{-64}$, and $h \approx 50$ (chosen to optimize a trade-off between the Bloom filter construction and online query costs—see Theorem 18), the resulting optimal Bloom filter size is under a third of the item storage size

(of $2n$ blocks), for a wide variety of database sizes ranging from a megabyte up to a petabyte. For bounding the false positive rate at 2^{-128}, as we shall assume for the performance analysis below, Bloom filter storage of less than the total item storage size still suffices.

The target hardware configuration is listed in Figure 7.4, and the target database configuration in Figure 7.5. Solid state disks are assumed, resulting in a low disk seek cost. Disk access time is modeled as a function of the disk throughput; divergence between this model and reality is examined and resolved in Section 7.5.1.

The results are illustrated in Figure 7.6. The item shuffle cost is unchanged over the ideal ORAM construction, but the Bloom filter shuffle cost adds a significant fraction to the overall level construction cost. As can be seen, however, the $\log_2 n$ round trips imposed by existing work quickly add up to exceed the SR-ORAM shuffle cost, in even moderate latency networks. In general, the additional offline cost is small compared to the savings in the online cost.

The SR-ORAM online performance cost encompasses the cost of transferring the query object across the network, reading each key part from disk, server-side decryption, and transmitting the results back to the client. The offline performance cost encompasses the disk, encryption, and network costs of shuffling the items using a 4-pass random shell sort, and constructing the Bloom filter. The step shape results from the additional level that is shuffled and queried every time the database size doubles. While this no longer adds an additional round trip, it does increase the query object size, require reading an extra set of (e.g. 50) Bloom filter positions (key components) from disk, and require additional construction time.

Discussion. Offline shuffling in both the ideal storage-free ORAM and SR-ORAM takes somewhat longer than that described in Chapter 6 (where we achieve several queries per second on a 1TB database) because we are eliminating the assumption of client storage. Instead of using a storage-based merge scramble that requires $\log \log n$ passes to sort n items, we use a randomized shell sort, that requires $\approx 24 \log n$ passes. This is not a result of my technique—storage can be applied to speed up our result equivalently—but a result of the different model here, under which we do not provide $O(\sqrt{n \log n})$ client storage.

7.5.1 Dealing with Reality

Although it is convenient to model disk data transfer costs based only on the average disk throughput, this is not a complete model, not even for solid state disks. In the solid state disks we consider above, the disk sector size must be considered: it specifies the minimum size of a read or write we can perform. That is, reads or writes of smaller amounts cost the same as reading/writing the entire sector. This is problematic for one piece of the SR-ORAM cost analysis: the random shell sort of segments during the Bloom filter construction. The Bloom filter segments (e.g., 32 bytes) are potentially much smaller than a disk sector (e.g., 4096 bytes), and are accessed in a random pattern during the random shell sort.

I now describe implementation details that avoid the expense resulting from short disk reads. First, leading to a simple but expensive potential solution, recall that the disk costs

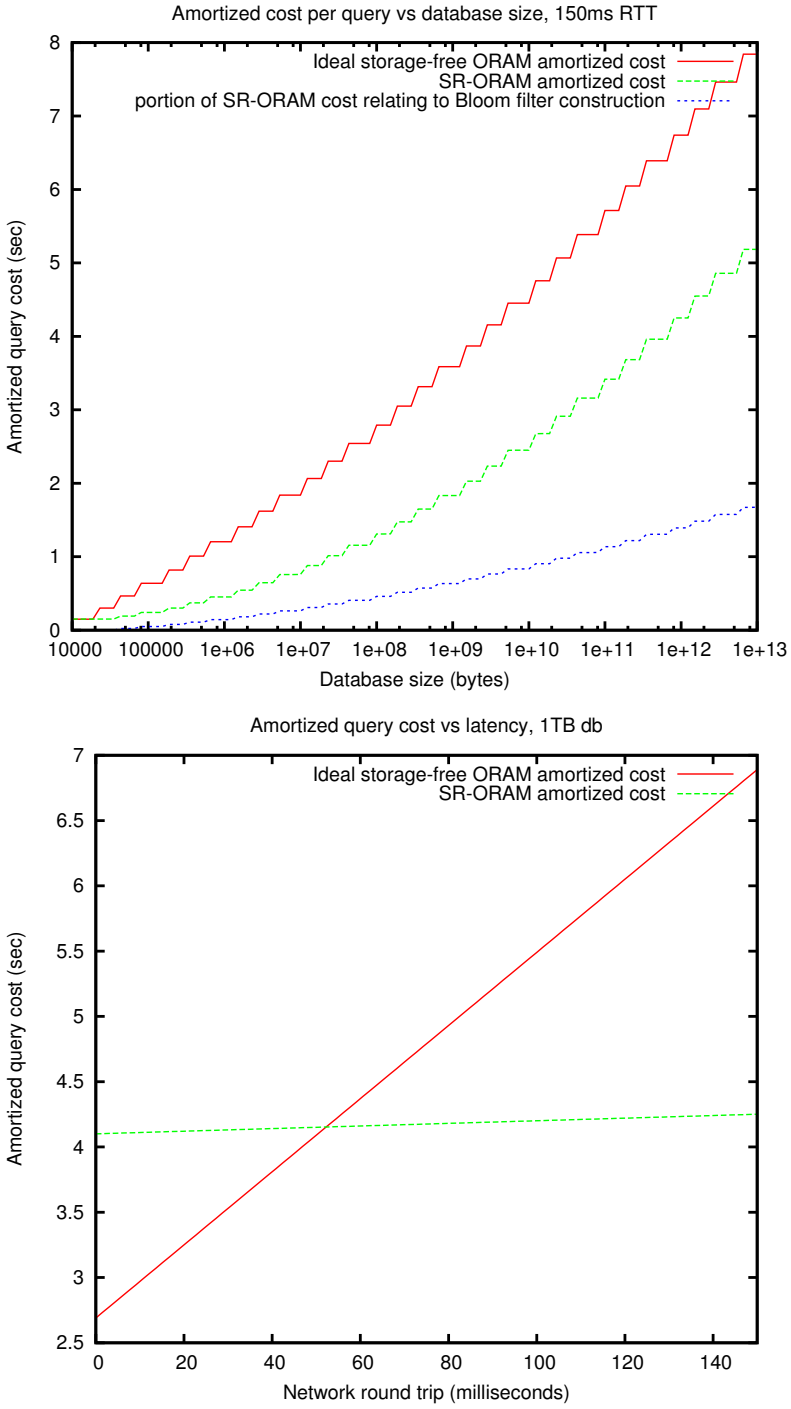

Figure 7.6: Comparison of amortized (online plus offline) cost of SR-ORAM to the ideal interactive storage-free ORAM, assuming the hardware configuration of Figure 7.4. Left: query cost plotted as the database size grows on a logarithmic X axis. Right: comparison vs. latency for a fixed database size.

are all server-side costs. This makes it plausible to solve this cost discrepancy with additional hardware. For example, a large amount of server RAM (e.g., 48 GB for a 1TB database) would make it possible to cache the *entire* Bloom filter during construction, requiring only a single contiguous write at the end. A key point is that the Bloom filter is smaller during these sort procedures than the final resulting Bloom filter will eventually be. For example, it is blown up by about a factor of 4 when converting the bits (with their, e.g., 64-bit positions) into (e.g., 256-bit) keys at the end. This conversion process requires only a single, sequential scan and write. This amount of RAM would result in a sort faster than the estimates used above (pushing the bottleneck to client crypto speeds instead of disk I/O speeds), which assume that disk transfer is required for each of the $\approx 24 \log_2 n$ passes across the data in the random shell sort. This is not necessarily an unreasonable scenario, as the RAM is only used for the duration of the sort, making it feasible to "pool" resources across many clients.

A second option is to assume enough client memory to build the Bloom filter locally. For a 1 TB database consisting of 10KB blocks, and taking the acceptable failure rate to be 2^{-128}, and 50 hash functions, the total number of Bloom filter positions to set can be under 16 billion bits. This fits in 2GB of client memory. Moreover, since this construction is now done in private client memory, the oblivious Bloom filter sort can be avoided, speeding up the Bloom filter construction significantly. This process now requires only the network transfer, and sequential disk write of the final key-storing Bloom filter (.5TB). However, the client memory requirement for that (cost-optimal) Bloom filter construction process is linear in the total database size, a trait I desire avoiding.

Fortunately, such a workaround is not necessary. I now illustrate how the randomized shell sort can be performed without incurring the overhead resulting from the minimum sector size. In exchange, we require extra server-side sequential scans of the data. Observe that a randomized shell sort consists of dividing the array to be sorted (sized s) into equal sized regions, and swapping items between the two regions (called the "compareRegions" operation). The region sizes start at $s/2$ and decrease all the way down to 1. In any compareRegions operation, one region is read sequentially, while the other region is read randomly. Likewise, the regions are written back in the order they are read. Two properties are key to eliminating disk seek and partial sector read costs. First, observe that for regions that fit entirely in the server available cache sized M, the disk seek / minimum sector cost can be avoided altogether with an aggressive read-ahead page cache. This is because any one region can be read and cached in its entirety, and small regions are accessed contiguously.

Second, when the regions are too big to fit in the server page cache, the access pattern is still predictable by the server. This means it can sort data according to the future access pattern. Moreover, this sort can be performed in only $\log_M n$ passes. This is 2 passes whenever the server has $\sqrt{n}$ blocks of RAM, which I believe to be a more than reasonable assumption. The idea is, in one pass, sort data in groups sized M. In the second pass, merge these n/M regions together, reading contiguous chunks from each region into the page cache (the default behavior of a read-ahead page cache). This way, during the shell sort, the items being read can be accessed contiguously.

For simplicity, the data is sorted back into its start location after being written. The

result is a sort that performs efficiently, even when the sort item size is much smaller than the disk sector size. The penalty is that now data is scanned from the disk up to four times in each sort pass (but this is more than made up for by the savings of not having to read an entire disk sector for every access of a 32-byte element).

This cost is reflected in the graphs above.

A similar argument comes into play when modeling the encryption/decryption throughput; my model considers the sustained throughput over large blocks of data, while when sorting the Bloom filter positions, many of the decryptions and encryptions are on short (e.g. 64-bit) blocks. Fortunately, the encryption block size (e.g., 256-bit) is much closer to the encryption size, resulting in the actual crypto throughput staying within a factor of 1/4 of the modeled crypto throughput. This analysis is also reflected in my graphs, which measure the entire cost of decrypting a cipher block even when the amount of data to be decrypted in a single operation is smaller than the block.

7.6 Conclusion

I introduced a new single-round-trip ORAM. I analyzed its security guarantees and demonstrated its utility. While non-interactivity is of significant theoretic interest in itself, I also showed this to be the most efficient storage-free-client ORAM to date for today's average Internet-scale network latencies.

Part II

Access Privacy in Other Scenarios

Chapter 8

Approaches Outside Oblivious RAM

8.1 Private Information Retrieval (PIR)

Private Information Retrieval (PIR). PIR addresses a similar scenario of private querying. In contrast to ORAM, PIR targets scenarios involving *multiple, unrelated clients* querying a *public database*. And, while the data in most cases is *not* confidential, PIR does not assume nor usually require client-side credentials management, in contrast to ORAM which, in principle assumes all accessing clients have access to the same credentials and the data is to be protected from the server. Unlike ORAM, PIR typically only provides read privacy, not update privacy, though notable exceptions include constructions by Lipmaa and Zhang [78] and Boneh et al. [16].

PIR. Private Information Retrieval has been proposed as a primitive for accessing outsourced data over a network, while preventing its storer from learning anything about client access patterns [23]. In initial results, it was shown [23] that in an information-theoretic setting in which queries do not reveal any information at all about the accessed data items, any solution requires $\Omega(n)$ bits of communication, for a database of size n. To avoid this overhead, if multiple non-communicating databases can hold replicated copies of the data, PIR schemes with only sublinear communication overheads are shown to exist [23]. For example, Sassaman et al. [111] applied such a scheme to protect the anonymity of email recipients. In the world of data outsourcing, in which there are only a few major storage providers, I do not believe the assumption of non-collusion among such untrusted servers is always practical, so I do not wish to rely on this assumption. Goldberg [39] introduced a construction that combines a multi-server PIR scheme with a single-server PIR scheme, to guarantee information-theoretic PIR if the servers are not colluding, but still maintain computational privacy guarantees when all servers are colluding.

It is not my intention to survey the inner workings (beyond complexity considerations) of various PIR mechanisms or of associated but unrelated research. I invite the reader to explore a multitude of existing sources, including the excellent, almost complete survey by William Gasarch [34, 35].

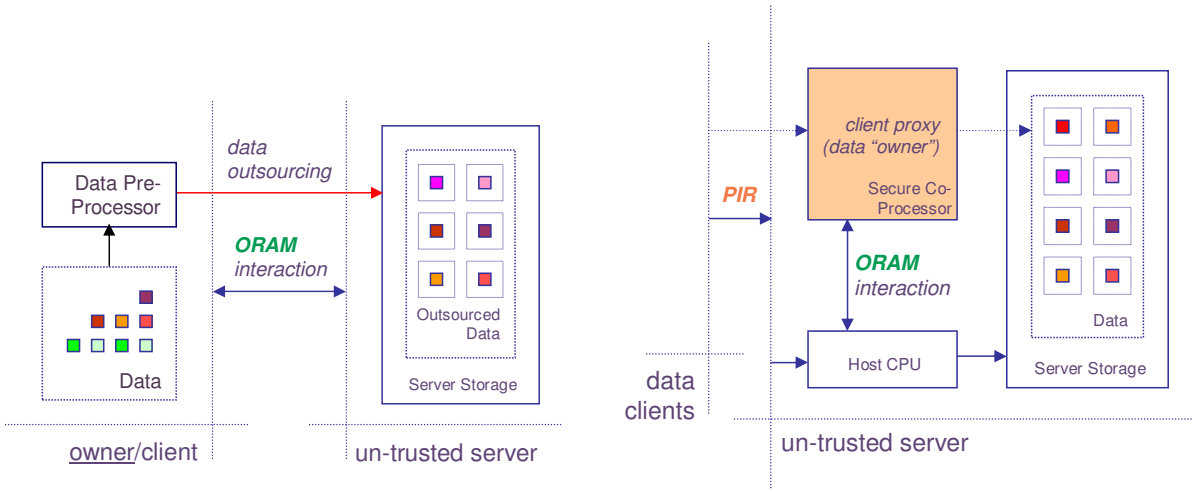

Figure 8.1: (left) Simple ORAM Protocol between a client and a server. (right) A trusted server-side client proxy can be used to build a PIR interface on top of ORAM assurances.

8.1.1 Secure Hardware-aided PIR

The recent advent of tamper-resistant, general-purpose trustworthy hardware such as the IBM 4764 Secure Co-Processor [59] has opened the door to efficiently deploying ORAM privacy primitives for PIR purposes (i.e., for arbitrary public or private data, not necessarily originated by the current client) by deploying such hardware as a trusted server-side client proxy.

Asonov was the first to introduce [5] a PIR scheme that uses a secure CPU to provide (an apparent) $O(1)$ online communication cost between the client and server. However, this requires the secure CPU on the server side to scan portions of the database on every request, indicating a computational complexity cost of $O(n)$, where n is the size of the database.

An ORAM-based PIR mechanism is introduced by Iliev and Smith [61], who deploy secure hardware to achieve a cost of $O(\sqrt{n}\log n)$. This is better than the poly-logarithmic complexity granted by ORAM for the small database sizes they consider. This work is notable as one of the first implementations of ORAM-based PIR setups. Figure 8.1 summarizes the interaction between the client and server in ORAM, and how to turn an ORAM implementation into a PIR implementation using a Secure CPU.

Achieving PIR. Part I described providing access pattern privacy for private data. A general PIR implementation requires a client to be able to download from a public server, meaning the client does not have access to prearranged secret keys. By implementing the access pattern privacy on a server-side tamper-proof trusted processing component such as a Secure Co-processor, PIR can be achieved. The secure CPU maintains the encrypted database, and never leaks any of the encryption keys to the server or clients. Clients who wish to retrieve an item privately then interact with the main data through the SCPU.

Predicted performance will be considered in Section 4.3. Taking the IBM 4764 as an example, Figure 4.5 shows that when PIR is implemented on the secure CPU, the bottleneck is no longer the network bandwidth, but the en/decryption times. Additionally of concern

is the limited SCPU storage. The 64 MB of RAM available on a 4764 can support a 10 GB database with the probability of overflowing the queue space of less than 2^{-194}, derived from the bounds established in Theorem 14.

Memory Pooling. A key advantage to my algorithm is that the working buffers are only used for a small period of time and are transient, thus requiring no backup. Therefore the high cost of client storage maintenance is avoided since no data is lost if the working memory is lost. A second advantage is that resources can be pooled between SCPUs to support larger databases. For example, if a storage provider manages 10 SCPUs for 10 customers, and if the working buffer is only in use for 10% of the time, the provider can pool the secure storage between SCPUs, allowing for an effective secure storage area of 640 MB instead of just 64 MB; this is enough allow the provider to support 1 TB databases.

The limiting factor in pooling is the percentage of time the secure CPUs are put to use. This will vary based on the actual transaction patterns of the clients. If transactions are run continuously at the maximum throughput, I expect the idle time to be around 50%. If there are idle periods, however, and the average throughput is lower, each SCPU may see a much higher idle time.

Existing PIR. Trivial PIR (transferring the entire database to the SCPU for every query) will have a bottleneck shared by the bus transfer time and the disk transfer time, of 50 MB/sec. For a 1 TB database, this will require about 22000 seconds per query.

A PIR protocol introduced by Wang et al. [125] offers an amortized complexity of $O(n/k)$ for database size n and secure storage size k. For $k = O(\sqrt{n \log n})$, this yields an overhead of $O(\sqrt{n/\log n})$ per query. This is proving to be a reasonable estimate of k, since as described earlier, database sizes and hard disk capacity are increasing much faster than secured storage capacity. As databases become larger, my superior $O(\log^2(n))$ complexity becomes increasingly necessary for obtaining practicality.

8.1.2 Related Constructions

Practicality of PIR. In initial results, Chor et al. [24] proved that in an information theoretic setting, any single-server solution requires $\Omega(n)$ bits of communication. PIR schemes with only sub-linear communication overheads, such as the seminal construction provided by Chor et al. [24], require multiple non-communicating servers to hold replicated copies of the data. The information theoretic guarantees can be relaxed to assume a computationally bounded adversary, in which case single-server solutions have been proposed.

For single-server PIR, initial results [116], seemed to suggest it is extremely difficult to provide PIR implementations that are faster than the trivial case (downloading the entire public database to the client).

Recent work [99] however, has shown that multi-server schemes as well as more recent lattice-based mechanisms are 10-1000 times faster than downloading the entire database.

PIR vs. ORAM. Unfortunately, this still leaves PIR several orders of magnitude behind

ORAM mechanisms – which are faster at the expense of requiring client-side key management. For example, for the 1TB database considered here, over a dedicated 100Mbps link, in the absolute ideal case, ignoring any other constants, round-trip times and systems issues, a lattice-based single-server PIR query would still take upwards of 3 minutes (2 orders of magnitude slower than PD-ORAM).

When considering client query encoding, network round-trips, server disk seeks[1] as well as the cost of the extremely high server CPU utilization, the ORAM-PIR difference can grow beyond 3 orders of magnitude.

8.2 Other Approaches

Anonymity. The goal of anonymous communication is to hide not the data being accessed, but the identity of the party accessing this data. In many cases this provides suitable privacy. Anonymous communication approaches are less complete, however; efficient anonymous communication is able to provide only weak security guarantees, relying on the adversary's inability to obtain any sort of global network view. Johnson et al. present a discussion of state of the art anonymous routing approaches [65]. Moreover, often anonymous communication is not sufficient. Combined with external knowledge, the access patterns can compromise anonymity, or reveal sensitive information in themselves.

Oblivious Transfer (OT). OT [63, 96, 110] is a related cryptographic primitive in which clients can request a number of database records from a server with full privacy. Further, clients are prevented from learning any additional database entries. In contrast to ORAM and PIR, OT does not aim to achieve any efficiency targets, e.g., such as sublinear communication in the database size etc.

[1] Possibly catastrophically dominant, as our results on deploying the Goodrich et al. theoretically-attractive randomized shell sorting network [46] have shown (see above discussion and analysis in Section 3.1.2)

Chapter 9

Transaction Privacy

I introduce a new paradigm for outsourcing the durability property of a multi-client transactional database to an untrusted service provider. Specifically, I enable untrusted service providers to support transaction serialization, backup and recovery for clients, with full data confidentiality and correctness. Moreover, providers learn nothing about transactions (except their size and timing), thus achieving read and write access pattern privacy.

I build a proof-of-concept implementation of this protocol for the MySQL database management system, achieving tens of transactions per second in a two-client scenario with full transaction privacy and guaranteed correctness. This shows the method is ready for production use, creating a novel class of secure database outsourcing models.

9.1 Introduction

Increasingly, data management is outsourced to third parties. This trend is driven by growth and advances in cheap, high-speed communication infrastructures.

Outsourcing has the potential to minimize client-side management overheads and benefit from a service provider's global expertise consolidation and bulk pricing. Providers such as Amazon [79–81], Google [48], and others—ranging from corporate-level services such as IBM Data Center Outsourcing Services [26] to personal level database hosting [62, 100] —are rushing to offer increasingly complex storage and computation services.

Yet, significant challenges lie in the path of a successful large-scale adoption. In business, health care and government frameworks, clients are reluctant to place sensitive data under the control of a remote, third-party provider, without practical assurances of *privacy* and *confidentiality*. Yet today's privacy guarantees of such services are at best declarative and often subject customers to unreasonable fine-print clauses—e.g., allowing the server operator (or malicious attackers gaining access to its systems) to use customer behavior and content for commercial, profiling, or governmental surveillance purposes [20, 66]. These services are thus fundamentally insecure and vulnerable to illicit behavior.

Existing research (discussed in Section 9.1.1) addresses several important outsourcing aspects, including direct data retrieval with access privacy, searches on encrypted data,

and techniques for querying remotely-hosted encrypted structured data in a unified client model [29, 92]. These efforts are based on the assumption that, to achieve confidentiality, data will need to be encrypted before outsourcing to an untrusted provider. Once encrypted however, inherent limitations in the types of primitive operations that can be performed on encrypted data by untrusted hosts lead to fundamental query expressiveness constraints. Specifically, reasonably practical mechanisms exist only for simple selection and range queries or variants thereof.

In this chapter, I introduce an orthogonal thesis spurred by the advent of cheap and fast disks and CPUs. I believe that in many deployments, individual data clients' systems are in a position to host and access local large data sets with little difficulty at no additional cost.

Holding data locally may appear to contradict the spirit of outsourcing, yet I argue that there is one essential data management aspect that cannot be hosted as such: *transaction processing for multiple concurrent clients*. In a world where client machines have become powerful enough to run local databases, I predict data management outsourcing markets will converge on supplying the transactional, network intensive services and availability assurances which are not trivially achievable locally.

In this chapter, I thus introduce a novel paradigm for solving the data management outsourcing desiderata: *a mechanism for collaborative transaction processing with durability guarantees supported by an untrusted service provider under assurances of confidentiality and access privacy*. In effect, I achieve the cost benefits of standard outsourcing techniques (durability, transaction processing, availability) while preserving the privacy guarantees of local data storage. This is accomplished by enabling data clients to collaboratively perform runtime transaction processing and interact through an untrusted service provider that offers durability and transaction serializability support.

In this context, data outsourcing becomes a setting in which all permanent data is hosted securely encrypted off-site, yet clients access it through their locally-run database effectively acting as a data cache. If local data is lost, it can be retrieved from the offsite repository. Inter-client interaction and transaction management is intermediated by the untrusted provider who also ensures durability by maintaining a client-encrypted and authenticated transaction log with full confidentiality.

In our model, each client maintains its own cache of (portions of) the database in client-local storage, allowing it to perform efficient reads with privacy, while relieving local system administrators of backup obligations. The key benefit thus becomes achieving data and transaction privacy while (1) avoiding the requirement for persistent client storage (clients are now allowed to fail or be wiped out at any time), and (2) avoiding the need to keep any single client-side machine online as a requirement for availability.

9.1.1 Related Work

Queries on Encrypted Data. The paradigm of providing a database as a service recently emerged [51] as a viable alternative, likely due in no small part to the dramatically increasing availability of fast, cheap networks. Given the global, networked, possibly hostile nature of

the operation environments, security assurances are paramount.

Hacigumus et al. [50] propose a method to execute SQL queries over partly obfuscated outsourced data. The data is divided into secret partitions and queries over the original data can be rewritten in terms of the resulting partition identifiers; the server then performs queries over the partitions. The information leaked to the server is claimed to be 1-out-of-s where s is the partition size. This balances a trade-off between client-side and server-side processing, as a function of the data segment size. At one extreme, privacy is completely compromised (small segment sizes) but client processing is minimal. At the other extreme, a high level of privacy can be attained at the expense of the client processing the queries in their entirety after retrieving the entire dataset. Moreover, Hore et al. [58] explore optimal bucket sizes for certain range queries. Similarly, data partitioning is deployed in building "almost"-private indexes on attributes considered sensitive. An untrusted server is then able to execute "obfuscated range queries with minimal information leakage". An associated privacy-utility trade-off for the index is discussed. The main drawbacks of these solutions lies in their computational impracticality and inability to provide strong confidentiality.

Recently, Ge et al. [36] discuss executing aggregation queries with confidentiality on an untrusted server. Unfortunately, due to the use of extremely expensive homomorphisms (Paillier [103, 104]) this scheme leads to impractically large processing times for any reasonable security parameter choices (e.g., for 1024 bit of security, processing would take over 12 days *per query*). Current homomorphisms are not fast enough to be usable for practical data processing.

Avoiding the trade-off between processing and computation time altogether, I allow efficient queries on encrypted data with full privacy by running queries on a client-side decrypted copy of the data. Thus, with no additional network transfer, and with a computational cost equivalent to running the query on an unencrypted database, I provide full query privacy. Since we will run the queries on a copy of the database at the client side, so there is no need for expensive homomorphisms, either.

Query Correctness. In a publisher-subscriber model, Devanbu et al. deployed Merkle trees to authenticate data published at a third party's site [29], and then explored a general model for authenticating data structures [83, 84]. Hard-to-forge verification objects are provided by publishers to prove the authenticity and provenance of query results. Mykletun et al. [92] introduce mechanisms for efficient integrity and origin authentication for simple selection predicate query results. Different signature schemes (DSA, RSA, Merkle trees [89] and BGLS [15]) are explored as potential alternatives for data authentication primitives. Mykletun et al. [94] also introduce *signature immutability* for aggregate signature schemes—the difficulty of computing new valid aggregated signatures from an existing set. Such a property is defeating a frequent querier that could eventually gather enough signatures data to answer other (un-posed) queries. The authors explore the applicability of signature-aggregation schemes for efficient data authentication and integrity of outsourced data. The considered query types are simple selection queries. Similarly, Narasimha and Tsudik [97] deploy digital signature and aggregation and chaining mechanisms to authenticate simple selection and

projection operators. While these are important to consider, nevertheless, their expressiveness is limited. A more comprehensive, query-independent approach is desirable. Moreover, the use of strong cryptography renders this approach less useful. Often simply transferring the data to the client side will be faster. Pang and Tan [106] deploy *verification objects* to authenticate simple data retrieval in "edge computing" scenarios, where application logic and data is pushed to the edge of the network, with the aim of improving availability and scalability. Lack of trust in edge servers mandates validation for their results—achieved through verification objects. Pang et al. [105] deploy Merkle tree and cryptographic hashing constructs to authenticate the result of simple range queries in a publishing scenario in which data owners delegate the role of satisfying user queries to a third-party un-trusted publisher. Additionally, Narasimha and Tsudik [98] deploy closely related mechanisms in database outsourcing scenarios. Devanbu et al. [28] propose an approach for signing XML documents allowing untrusted servers to answer certain types of path and selection queries.

Sion [115] explored query correctness by considering the query expressiveness problem where a novel method for proofs of *actual* query execution in an outsourced database framework for *arbitrary* queries is proposed. The solution is based on a mechanism of runtime query "proofs" in a challenge-response protocol built around the *ringer* concept first introduced by Golle and Mironov [43]. For each batch of client queries, the server is "challenged" to provide a *proof of query execution* that offers assurance that the queries were actually executed with completeness, over their entire target data set. This proof is verified at the client site as a prerequisite to accepting the actual query results as accurate.

While many of these efforts have produced efficient query correctness verification mechanisms for specific kinds of queries, unique approaches are required for different queries. Moreover, it has proven difficult to *simultaneously* provide correctness and privacy with these techniques, since the construction of a query verification object typically requires knowledge of the query. To simultaneously ensure query correctness, query privacy, and database privacy for fully general queries (on relational or other types of databases), we can instead verify only the *updates*. This mechanism guarantees that clients have correct views of the database, thus ensuring they also obtain correct query results.

Database Integrity and Audit Logs. In a different adversarial and deployment model, researchers have also proposed techniques for protecting critical DBMS structures against errors [13, 124]. These techniques deal with corruptions caused by software errors. In work on tamper proof audit logs by Snodgrass et al. [107, 118] introduces the idea of hashing transactional data with cryptographically strong one-way hash functions. This hash is periodically signed by a trusted external digital notary, and stored within the DBMS. A separate validator checks the database state against these signed hashes to detect any compromise of the audit log. If tampering is detected, a separate forensic analyzer springs into action, using other hashes that were computed during previous validation runs to pinpoint when the tampering occurred and roughly where in the database the data was tampered. The use of a notary prevents an adversary, even an auditor or a buggy DBMS, from silently corrupting the database.

This notion of a cryptographically protected log provides the framework for this solution.

Rather than using an audit log to identify errors, however, I propose use of an update log stored by an untrusted party as the authoritative version of the database; the cryptographic hash properties are used to prevent tampering by the untrusted party. Our log does not protect from software bugs in the DBMS, as audit logs traditionally do, since our logs store only a list of updates, not checksums on the contents.

Databases Replication. Database replication has been pursued as a means to improve query latency and throughput, and eliminate any single point of failure. Researchers have found efficient solutions guaranteeing various levels of consistency. In particular, Kemme and Alonso [68], and later Amir and Tutu [4], build replicated databases on top of existing RDBMS's using a reliable group communication system with total message ordering and (in Amir and Tutu's work) message logging.

I employ a similar model of operation, in which clients simulate transactions locally before distributing them. In place of the group communication system, however, we will use a dedicated, untrusted provider, and provide cryptographic guarantees of security and privacy. This allows us to remove the availability and write durability requirements from the replication servers in previous work, assigning them to the untrusted provider, making client system administration much simpler.

Encrypted Storage. Encryption is one of the most common techniques used to protect the confidentiality of stored data. Several existing systems encrypt data before storing it on potentially vulnerable storage devices or network nodes. Blaze's CFS [11], TCFS [19], EFS [90], StegFS [88], and NCryptfs [131] are file systems that encrypt data before writing to stable storage. NCryptfs is implemented as a layered file system [54] and is capable of being used even over network file systems such as NFS. SFS [49] and BestCrypt [64] are device driver level encryption systems. Encryption file systems are designed to protect the data at rest, yet only partially solve the outsourcing problem. They do not allow for complex retrieval queries or client access privacy.

Integrity-Assured Storage. Tripwire [69, 70] is a user level tool that verifies file integrity at scheduled intervals of time. File systems such as I^3FS [67], GFS [37], and Checksummed NCryptfs [117] perform online real-time integrity verification. Venti [109] is an archival storage system that performs integrity assurance on read-only data. Mykletun et al. [92, 93] explore the applicability of signature-aggregation schemes to provide computation- and communication efficient data authentication and integrity of outsourced data.

In integrity-protected random-access storage, significant computational overhead is typically required to prevent rollback attacks, in which an adversary replaces a portion of the data with an older version. The party intending to detect such an attack may need a proof of the latest versions identifier of all stored data, for example. We avoid this overhead since each client keeps track of this information in a locally stored copy. We are then left with only the task of ensuring a consistent version sequencing on updates, which we provide using a hash chain.

Keyword Searches on Encrypted Data. Song et al. [119] propose a scheme for performing simple keyword search on encrypted data in a scenario where a mobile, bandwidth-restricted user wishes to store data on an untrusted server. The scheme requires the user to split the data into fixed-size words and perform encryption and other transformations. Drawbacks of this scheme include fixing the size of words, the complexities of encryption and search, the inability of this approach to support access pattern privacy, or retrieval correctness. Eu-Jin Goh [38] proposes to associate indexes with documents stored on a server. A document's index is a Bloom filter [12] containing a codeword for each unique word in the document. Chang and Mitzenmacher [21] propose a similar approach, where the index associated with documents consists of a string of bits of length equal to the total number of words used (dictionary size). Boneh et al. [14] proposed an alternative for senders to encrypt e-mails with recipients' public keys, and store this email on untrusted mail servers. Golle et al. [42] extend the above idea to conjunctive keyword searches on encrypted data. The scheme requires users to specify the exact positions where the search matches have to occur, and hence is impractical. Brinkman et al. [18] deploy secret splitting of polynomial expressions to search in encrypted XML.

We will see that we can efficiently (and trivially) achieve the spirit of this goal—running searches with full privacy on "outsourced" data—by adjusting the model to redefine "outsourced". Specifically, we provide a model that achieves the bulk of the cost savings of outsourcing while retaining the performance benefits of locally managed data.

9.2 Model

I provide details of the participants in this protocol, the required transaction semantics, and the cryptographic primitives employed.

Parties

Provider/Server. The provider owns durable storage, and would like to provide use of this storage for a fee. The provider, being hosted in a well-managed data center, also has high availability. I will investigate ways that clients can make use of these attributes.

Since the provider has different motivations than the clients, I assume an actively malicious provider. However, I do not try to prevent denial of service behavior from the provider. There are techniques beyond the scope of this work, that can be employed to help clients detect denial of service behavior, such as attaching timestamps to messages to measure server latency.

Clients. In our model, the clients are a set of trusted parties who must run transactions on a shared database with full ACID guarantees. Since storage is cheap, each client has a local hard disk to use as working space; however, due to the fragile nature of hard disks, we shall not assume this storage is permanent. Additionally, the clients would like to perform read queries as efficiently as possible without wasting network bandwidth or paying network

latency costs. Each of the trusted parties would also like to be able to continue running transactions even when the others are offline, possibly making use of the provider's high availability.

The clients would like to take advantage of the durability of the provider's storage, but they do not trust the provider with the privacy or integrity of their data. Specifically, the provider should be prevented from observing any of the distributed database contents. I define a notion of consistency between the clients' database views to address integrity. It is not imperative that all clients see exactly the same data as the other clients at exactly the same time, however, they need to agree on the sequence of updates applied. I define $trace_{c,i}$ to be the series of the first i transactions applied by client c to its local database copy. Clients c and d are considered **i-trace consistent** if $trace_{c,i} = trace_{d,i}$.

In some scenarios the provider might be able to partition the set of clients, and maintain separate versions of the database for each partition. This partitioning attack has been examined in previous literature; if there are non-inter-communicating asynchronous clients, the best that can be guaranteed is fork consistency, first defined by Mazières and Shasha [85]. Any adopted solution should guarantee that the data repository is fork consistent; that is, all clients *within each partition* agree on the repository history. This guarantee is not as weak it may appear to be on the surface, as once the provider has created a partition, the provider must block all future communication between partitioned clients, or else the partition will be immediately detected.

I assume that clients do not leak information through transaction timing and transaction size. Clients in real life may vary from this with only minimal loss of privacy, but I use a timing and size side-channel free model for illustration purposes.

The first part of this chapter assumes a potentially malicious provider, but trusted clients. Section 9.5.1 relaxes this assumption to provide protection against not only a potentially malicious provider, but against malicious clients at the same time.

Transaction Semantics

I provide a general protocol that supports nearly any class of transaction. Transactions can be simple key-value pair updates, as in a block file system, or they can be full SQL transactions. Clients can even buffer many local updates over a long period of time, e.g. when the client is disconnected, and then send them as a single transaction. The only requirements for using this protocol are that the underlying transaction-generating system implement the following:

`RunAndCommitLocalTransaction`(Transaction T) applies transaction t to the local database and commits.

`DetectConflict`(TransactionHandle h, Transaction C) returns `true` if the external (visible to the program) outcome of Transaction T_h would have been different had transaction C been issued before T_h. It does not matter whether the local database contents would be different; all that matters is whether the transaction issuer would have acted differently.

`Retry`(TransactionHandle h) rolls back all changes (in the local database, and any side-effects external to the database) for uncommitted transaction T_h and re-attempts the transaction.

`RollbackLocal`(TransactionHandle h) rolls back local database changes from uncommitted transaction T_h.

I provide the following interface to the transaction-running system:

`DistributeTransaction`(Transaction T, TransactionHandle h) returns once transaction T has been successfully committed to the global database image. Implementations of this command will invoke the callbacks above.

Conflicts

In the protocols described later, multiple clients will simultaneously run transactions that ultimately end up in a different order than the clients see. Therefore, I define the notion of conflicts to indicate whether this re-ordering affects the client computation.

Let us say transactions a and b *conflict* if changing the order of these transactions affects the return value of one of the operations, or the client state that results from executing these operations. This definition allows us to provide the highest level of transactional isolation, serializability, as defined in the ANSI SQL-92 standard [3]. Client implementations may substitute a lower isolation level, such as the one defined by Adya and Liskov [1], to improve performance by reducing the number of conflicts.

For example, if this system represents a block file system, clients may abort transactions that read the value of a block that is written in a prior, pending transaction. If this system represents a relational database, clients may use row-based or table-based conflict detection. Alternatively, some transactions could be implemented as PL/SQL procedures that avoid sending any values back to the initiator, avoiding any possibility of conflicts altogether!

For correctness, there must be no false negatives returned by the `DetectConflict` command (no client decides its transaction is conflict-free based on the transaction list, when it in fact is not). Of course, the definition of conflict-free depends on the particular implementation. For optimal performance, the number of false positives returned by the `DetectConflict` command must also be low.

We shall require several cryptographic primitives with all the associated semantic security [40] properties: (i) a secure, collision-free hash function which builds a distribution from its input that is indistinguishable from a uniform random distribution (we use the notation $h(x)$), (ii) an encryption function that generates unique ciphertexts over multiple encryptions of the same item, such that a computationally bounded adversary has no non-negligible advantage at determining whether a pair of encrypted items of the same length represent the same or distinct items, (iii) a pseudo-random number generator whose output is indistinguishable from a uniform random distribution over the output space, and (iv) a recursive hash chain construction used to incrementally build a secure hash value over a sequence of items.

The first part of this chapter assumes a potentially malicious provider, but trusted clients. All transactions are encrypted by a symmetric key shared by the set of clients, and kept secret from the provider. Message authentication prevents tampering, and the use of a versioning structure guarantees database cache consistency. A strawman protocol introduces solution by providing the security guarantees trivially using a global lock (Section 9.3). My main result is a protocol providing these guarantees using an optimistic wait-free protocol (Section 9.4). I then describe several extensions to this protocol, including in Section 9.5.1 protection against not only a potentially malicious provider, but against malicious clients as well. Finally, my implementation shows how this protocol can be layered on top of existing SQL-based relation database management systems while obtaining practical performance overheads.

9.3 Strawman Protocol: Outsourced Durability with a Global Lock

I start by illustrating the main concepts through a strawman protocol that allows multiple clients to serialize their transactions through an untrusted server—transaction atomicity being guaranteed through a single global lock. Naturally, in practice, global locking is not a viable option as it would constitute a significant bottleneck. My main result, described in Section 9.4, is optimistic and lock-free.

An *encrypted transaction log* is the central data structure in all versions of this model. This log is the definitive representation of the database; the protocols described here simply allow clients to append to this log in a safe, parallel manner while preventing the potentially malicious provider from interfering with operations on the log.

At a high level, in this strawman protocol, clients maintain their own copy of the database in local temporary storage. They perform operations on this copy and keep it synchronized with other clients. Clients that go offline and come back online later, obtain a client-signed replay log from the untrusted server in charge of maintaining the log.

9.3.1 Transaction Protocol: the Lock-based `DistributeTransaction`

Informally, to run a transaction a client (1) "reserves" a slot in the permanent transaction log, (2) waits for the log to "solidify" up to its assigned slot, (3) runs the transaction, (4) "commits" that slot by sending out a description of the transaction to the untrusted server. The untrusted server then archives and distributes this encrypted transaction.

1. The client issues a "request slot" slot reservation command to server, along with a number l representing the last slot the client knows about (has seen updates for). The server assigns the next available transaction slot s to the client. The server sends back to the client this slot number s, with a list of all commits since the last update T_l received by the client, $T_{l+1} \dots Tj$. Note that $j < s - 1$ if there are clients reserving slots that have not yet committed at the instant the server issues the response to this message.

2. The client waits until all transactions $T_{j+1} \ldots T_{s-1}$ in slots before its assigned slot have committed. The client receives the updates from the server as they come in. After verifying checksums and authentication tokens (hash chain and signatures; see below) on each update, the client applies them to its local database copy (using `RunAndCommitLocalTransaction`) in order.

3. Once the client has received T_{s-1}, it has in effect obtained a global lock, since all other clients are now waiting for it to perform a transaction. The client now runs its own transaction on its local copy of the database, recording the sequence of updates.

4. The client commits (relinquishing the lock) by sending a complete encrypted description of the transaction updates T_s back to the server (which will relay it back to the other clients).

Finally, each client c applies transaction T_i (using `RunAndCommitLocalTransaction`) to its database once all the following conditions hold: (i) the contents of T_i have a valid signature from a valid client (using client-shared symmetric key K), (ii) the client has applied transactions $T_1 \ldots T_{i-1}$, and (iii) the hash chain link T_{i-1}.hashchain matches the client's own computation of link HC_{i-1}.

Each transaction T_i is encrypted and signed using a symmetric key K shared by all clients. It contains the following fields: **desc=** a transaction description, e.g., a sequence of SQL statements; **hashchain=** the signed hash chain link HC_{i-1}, verifying the sequence of transactions $T_1 \ldots T_{i-1}$. HC_i is calculated as $H_k(HC_{i-1}||T_{i-1}.desc)$, with $HC_0 = H_k(\emptyset)$.

```
hashchain := H_k(∅)
hashchain_pos := 0
k := ClientSharedKey
```

Procedure Initialize Global Variables

9.3.2 Correctness

I now show that this protocol is correct and offers fork consistency: all clients are trace consistent (their transaction traces are identical) as long as the server has not partitioned the clients. If the server *has* partitioned the clients, then all clients within a partition will be trace consistent.

Theorem 29. *If client c applies an update T_i of client d, then clients c and d are i-trace consistent.*

Proof. Assume that client c has applied T_i from client d, but suppose that $trace_{c,i}$ differs from $trace_{d,i}$. W.l.o.g. assume that at position a, $trace_{c,i}$ contains transaction T_a while $trace_{d,i}$ has a different value T'_a. Client c's computation of HC_a therefore differs from client d's computation of HC_a, since it involves a collision-free hash function. Additionally, the

Result: Retrieves and runs the next waiting transaction
t := server_getNextTransaction()
if *!verifySignature(t)* **then**
  return $\perp$
end
T := decrypt(t)
if *T.hashchain != hashchain* **then**
  return $\perp$
end
RunAndCommitLocaltransaction(T.desc)
hashchain := H_k(hashchain || T.desc)
hashchain_pos ++

Procedure GetIncomingTransaction

Result: Initiates a transaction
s := server_requestSlot()
while *hashchain_pos < s-1* **do**
  GetIncomingTransaction()
end
RunAndCommitLocalTransaction(d)
T := new Transaction
T.desc := d
T.hashchain := hashchain
server_commit(Enc(k,T))

Procedure DistributeTransaction(d)

collision-free property guarantees that any later link in these hash chains will also differ, and client c's computation of HC_{i-1} differs from client d's computation of HC_{i-1}. Since client c has applied T_i, it had to have successfully verified that T_i did indeed originate from client d. However, as a pre-condition to client c applying T_i, the hash chain link T_{i-1}.hashchain, which is client d's computation of HC_{i-1}, would have to match client c's own calculation of HC_{i-1}, giving us a contradiction. □

9.3.3 Privacy

Without knowledge of the shared symmetric key K, the server is unable to obtain any information from the encrypted transaction descriptions, aside from timing and size. Transaction slot requests contain no additional information.

9.4 Lock-free Outsourced Serialization and Durability

The obvious disadvantage to the above protocol is that it requires a global lock, restricting transaction processing as only one client may be active at a time. I now remove all locking from the protocol described above, and replace it with an optimistic conflict-detection mechanism. This allows clients to run transactions simultaneously, but adds the requirement that transactions are rolled back and reissued in the case of conflicts.

At an overview level, this protocol works as follows. Clients first issue an (encrypted) notification of their pending transaction, relayed to the other clients through the untrusted server. This contains enough information to allow other clients to determine whether it might cause a conflict with their own pending transactions. After this notification ("pre-commit"), clients then check to see if their pending transaction might conflict with any transactions scheduled to run before theirs. If not, they issue the commit; otherwise they retry with a new request. As in the previous protocol, clients maintain a transaction hash chain to guarantee consistency for misbehaving servers.

9.4.1 Transaction Protocol: the Lock-free `DistributeTransaction`

In this solution, running a transaction entails the following steps, outlined in figure 9.1:

1. The client simulates the intended transaction on its local database copy, then undoes this transaction on its own database copy. (Issuing the `RollbackLocal` client command defined in the Model section). It will properly apply the transaction only once it has applied the pending transactions first.

2. Once ready to commit, the client issues the "Request slot" command to the server, attaching an encrypted pre-commit transaction description P of its intended transaction, and the slot number l which is the latest the client knows about.

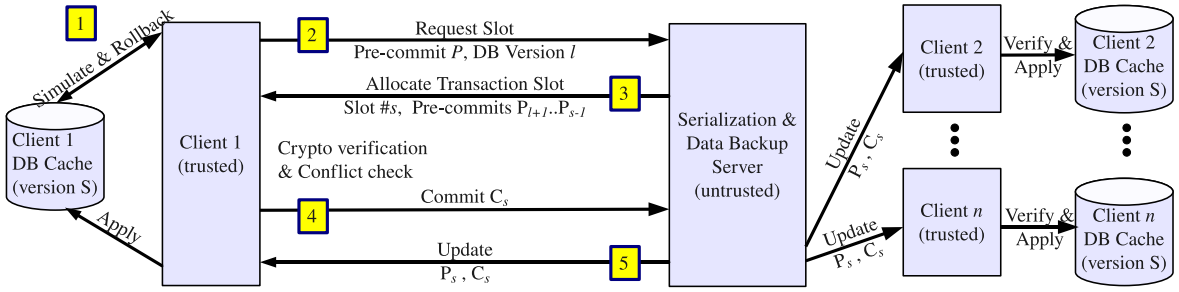

Figure 9.1: Performing an update: overview of `DistributeTransaction` execution, providing lock-free outsourced serialization and durability

3. The server allocates a slot s, and sends back a list of all new pre-commit descriptions $P_l \ldots P_{s-1}$ up to s. The server may choose to also send any previously committed transactions that the client has not seen yet at this point (e.g., this is the case if the client just joined or has been offline for a while).

4. The client verifies the signatures on each pre-commit, and checks whether its transaction conflicts with these pre-committed transactions (conflict semantics were discussed in Section 9.2). E.g., a conflict occurs with pre-commit $P_j, l < j < s$ if the *external* state would be different depending on which of P_j or P_s is run first (the `DetectConflict` command identifies these conflicts). If there are no conflicts, the client commits by sending a final encrypted transaction commit C_s. If there are conflicts, the client still sends the commit C_s, but sets its abort flag first (see below). In the case of a conflict, the client also rolls back the external effects of running the transaction locally (using the `Retry` command).

5. The server commits by logging the encrypted transaction to permanent storage. It informs all other clients about the new transaction by sending the final encrypted transaction C_s.

The pre-commit transaction description P_i contains the following information, encrypted and signed with the symmetric key K shared by all clients: **desc=** a transaction description, e.g., a sequence of SQL statements

The final encrypted, signed transaction C_i contains the following information: **commit=** a single bit indicating whether this is a commit or an abort; **prehashchain=** hash chain link $HCPre_i$, verifying the sequence of pre-commits $P_1 \ldots P_i$.

Note that when issuing commit i, the client has seen all *pre-commits* up through i, because the pre-commits up to i are returned when the client is assigned slot i. However, the client may not have yet seen all *commits* up to i when issuing this commit C_i.

Client c applies transaction i (invoking client command `RunAndCommitLocalTransaction`, originating from client d, once the following conditions hold: (i) the contents of P_i and C_i have

a valid signature from a valid client (using client-shared symmetric key K), (ii) C_i.commit indicates this is a committed transaction (not aborted), (iii) the client has applied transactions $1 \ldots i-1$, and (iv) the hash chain link C_{i-1}.pre-hashchain matches the client's own computation of link HC_{i-1} A pseudo-code approximation of these steps is included for reference.

```
prehashchain[] := [H_k(∅)]
links := 0
```

Procedure Initialize Global Variables

```
Result: Verifies and runs the transaction, reported by the server
if !verifySignature(p) or !verifySignature(c) then
    return ⊥
end
P := decrypt(p)
C := decrypt(c)
if prehashchain[id] != T.prehashchain then
    return ⊥
end
if id == links+1 then
    prehashchain.append(H_k(prehashchain[links] || T.desc))
    links++
end
if C.commit then
    RunAndCommitLocaltransaction(T.desc)
end
```

Procedure IncomingTransaction(p,c,id)

Note that this protocol is *wait-free*: if a client reserves a slot, sends its pre-commit but never completes, other clients can still perform transactions as long as they never access uncommitted data.

9.4.2 Correctness

We can again prove "fork consistency" for clients, as in the locking version. The difference here is that clients must agree on both C_i and P_i (the commit/abort status). Fortunately, only the client that issued C_i is allowed to issue P_i, so the proof follows.

Theorem 30. *If client c ever applies an update T_i of client d, clients c and d are i-trace consistent.*

```
Result: Runs a transaction initiated locally
RollbackLocal(h)
P := new Pretransaction
P.desc := T
s,pending := server_requestSlot(Enc_K(P),prehashchain.length)
conflict := false
foreach p,id ∈ pending do
    if !verifySignature(p) then
        return ⊥
    end
    p_id := decrypt(p)
    prehashchain.append(H_k(prehashchain[links] || T.desc)
    links++
    if DetectConflicts(h, p_id.desc) then
        conflict := true
    end
end
c := new Commit
c.commit := !conflict
c.prehashchain := prehashchain
server_commit(Enc_K(c))
if conflict then
    return Retry(h)
end
```

Procedure DistributeTransaction(T, h)

Proof. Assume that client c has applied transaction i from client d, but suppose that $trace_{c,i}$ differs from $trace_{d,i}$ at position a. There are two possible ways in which $trace_{c,i}$ could differ from $trace_{d,i}$. Either the transaction description P_a.desc differs between clients, or the commit status C_a.commit differs. If the transaction description differs, then this reduces to logic in Theorem 29: there is a hash function collision. Now consider the second case, that the commit status C_a.commit differs between two clients. The C_a.commit is obtained in a signed message from the client originating transaction a; therefore, since we assume clients follow the protocol, the originating client has signed two separate commit statements for the same transaction. Thus, the client has misbehaved, which contradicts our assumption of correct client behavior. □

In summary, we can guarantee fork consistency in the presence of a potentially malicious server using a shared hash chain. If two views of the ordered list of transactions ever differ between two clients, this will result in the shared hash chain diverging at the point of the inconsistency. This proof assumes correct clients. Section 9.5.1 shows how to prevent misbehaving clients from causing inconsistencies.

9.4.3 Privacy

In the absence of timing attacks, content, read/write access patterns and transaction dependencies are hidden from the server. This follows by construction as the contents of all messages are encrypted.

9.4.4 Forward Progress

Clients running this protocol will never deadlock, as long as they are not blocking for any external resources, since there is always the ability to make progress. This is evident since each transaction depends only on the transactions preceding it; the serialization numbers ensure there can never be any circular dependencies. At any point in time, there is always at least one transaction, at the front of the list, without any pending transactions to interfere.

For clients that are waiting on external resources, we can guarantee they will avoid deadlock as long as they (directly or indirectly) hold only external resources unneeded by prior, pending transactions. That is, my serialization technique assigns all transactions an ordering that makes it easy to prevent external resource deadlock as well.

"Livelock" and starvation are relevant concerns, however, and their applicability will depend on particular implementations. If a client detects it is being continually starved (i.e., there are always pending conflicting transactions), one solution is to block while waiting for the transaction chain to solidify up to a particular slot, since forward progress is guaranteed for the pending transactions. Conversely, a client must never block for a transaction past its slot. This forward independence prevents deadlock, and it also gives flexibility to client implementations; if a client needs a lock on a set of records, for example, it can request a transaction slot, then block until all transactions prior to the slot are committed. The client can then perform reads and writes with the equivalent of a lock. Meanwhile, other clients

can prepare transactions to run in the future, under the restriction that their transactions do not conflict with the pending transaction. Random backoff is an alternate solution. This will let clients escape from livelock, but requires active participation of both the starved and the healthy parties.

9.4.5 Client Initialization

When a new client comes online, there may be a long list of updates it must apply from the transaction log. To the time required for client initialization, I recommend clients create and sign periodic database snapshots, up to any particular transaction number. The untrusted server hosts these database snapshots, which can be used by clients to recover a particular version of the database. The remaining uncovered portion of the log is then used to get fully up to date.

Database snapshots can similarly be used to reduce the storage requirements of the untrusted provider. Once a snapshot of version i exists, the transaction log entries from 0 to i can be discarded. Using database snapshots in combination with the transaction log to allow faster recovery is a traditional DBMS method in common use.

9.4.6 Privilege Revocation

Depending on the particular implementation, it may also be useful for the decision to revoke access from a client to be made externally, by a trusted party/system administrator, or internally, by a quorum of clients. Once the remaining clients agree to revoke access, they choose a new symmetric encryption key. Additionally, clients agree on a slot at which the client is considered terminated. This termination point can be determined by a system administrator, or by a quorum of clients. Modifications to the database after this slot by the terminated client are all rejected (ignored) by everyone else. Incomplete transactions are easy to discard once the remaining clients can come to an agreement about which are incomplete.

After revocation, the only abilities retained by the terminated client from its former access is read access on the database for transactions before the termination point, and potentially the ability to cause denial of service. To remove the advantage the revoked client has in performing a denial of service attack on the database, the service provider should be notified. This operation is not strictly necessary, since we still provide correctness even when the revoked client and the storage provider are colluding.

9.5 Protocol Extensions

I now briefly describe several protocol extensions.

9.5.1 Malicious Clients

I describe here a simple extension to the lock-free protocol that allows us to prevent a malicious client from bringing the rest of the system into an inconsistent state. Preventing data overwriting by a malicious client is a separate concern, which is most appropriately addressed at the database access control level.

To detect malicious client behavior before the system becomes inconsistent, we will require two modifications. First, we shall extend the transaction integrity checks to include non-repudiation, so that each message is traced back to the issuing client. Specifically, we can replace the symmetric MAC function with a public key signing system. This prevents one client from impersonating another, and is required both to enforce the access control policy, and to gain accountability if incorrect behavior is detected.

The pre-commit hash chain already ensures that all clients agree on the pending transaction; to additionally protect from misbehaving clients colluding with the server, we simply need to ensure that all clients also agree on the commit/abort status of each transaction.

Since the protocol is lock-free and wait-free, clients will not necessarily know the commit/abort status of every transaction prior to their own as they issue a commit. Therefore, we can employ a delayed-verification mechanism: as part of commit C_i, the client includes the following items: (i) **commit-hashchain-position=** the position $j < i$ of the last element in the chain this client can build (i.e., this client has received $C_1 \ldots C_j$ but has not yet received C_{j+1}), and (ii) **commit-hashchain=** The value of hash chain element j.

Clients must also cache some prior hash links in order to verify the link included with commit C_i, since C_i. commit-hashchain-position might be less than $i-1$. This cache size can be configured, and for most transaction scenarios it is likely safe to keep only a few entries.

Note this provides a slightly weaker guarantee concerning the status of commit/abort. In the presence of a malicious client colluding with the server, clients c and d will not necessarily be k-trace consistent. However, after both c and d have applied the inconsistent transaction j, the next update issued by client d (which must contain a commit-hashchain-position $\geq j$) will reveal the inconsistency to client c.

Theorem 31. *If clients c and d behave correctly, and client c has applied transaction j and issued a transaction for slot $k > j$, and client d applies the update at k from client c, then c and d are j-trace consistent.*

Proof. Assume that client d has applied update k from client c. Thus, client d's computation of $HC(k)$ agrees with C_k.pre-hashchain, and the contents of the transactions are consistent; the inconsistency is therefore in the commit/abort status of transaction j. Since client c applied transaction j before issuing update k, C_k. commit-hashchain-position $\geq j$. Similarly, since client d has applied transaction i, client d verifies that C_k. commit-hashchain matches d's own computation of that chain link. However, since these two chain links have different values as inputs (one indicating that transaction j committed, and one indicating it did not), there is a hash function collision. □

In summary, we can prevent malicious clients from causing inconsistency using an access control policy framework to limit data damage, non-repudiability of messages to prevent

cross-client impersonation, and an additional hash chain to ensure clients agree on transaction commit/abort status.

9.5.2 Lowering Transaction Latency

We can reduce the number of network round trips required for a transaction commit from two to one by eliminating the commit messages C_i, as long as all clients have identical conflict detection logic. If we add another field to P_i indicating the last transaction the submitter has applied to its local database copy before this attempted transaction, other clients have enough information to determine the conflict status of this transaction! Thus, the commit flag in C_i.commit is redundant, at the expense of performing the conflict detection across all clients instead of just one.

The hash chain confirmation C_i.pre-hashchain will need to be placed in P_i, while adding another field C_i.pre-hashchain-location, since the submitter does not have enough information to build the entire pre-commit hash chain at this point. Thus, inconsistency checking is delayed slightly, and it requires longer to detect malicious behavior. Specifically, an inconsistency introduced due to server misbehavior will be detected only once a client that has applied the inconsistent transaction has sent a later update out to other clients who have seen a different transaction in that slot. This guarantee about detecting server misbehavior is similar to the guarantee about detecting client misbehavior in Section 9.5.1.

In conclusion, a simple modification to this protocol improves transaction latency by eliminating the commit message, at the expense of slightly more client computation time and slightly weaker consistency guarantees.

9.5.3 Large Databases

So far I have not discussed the issue of local space limitations. We assumed up to now that clients can fit the entire database in local (volatile) storage, so that they can run queries without any help from other parties. If this is not the case, protocol extensions are necessary to allow clients to run queries. I discuss several mechanisms below.

On-demand data Clients can use a separate query protocol to pull pieces of recent database snapshots from other clients, or authenticated database snapshots directly from the provider. This work-around has two drawbacks: first, access pattern privacy is forfeited if clients query the provider directly for only portions of the database. Second, performance suffers since sections of the database must travel the network multiple times.

Large object references If clients can fit the database except for a set of large objects, the client can fetch these encrypted objects from the provider using a separate protocol. Access pattern privacy to these objects is lost, however access pattern privacy to the database indexes is preserved. In practice, privacy to the indexes is the most important part of privacy, since the indexes are subject to the largest semantic leaks, since positions within the index are correlated to contents of the database itself. Thus, this technique offers a useful privacy/storage trade-off.

Performance will be mostly unharmed by the large object references work-around, as long as the bulk of the transaction processing work concerns only index data. The client's available storage is well suited for caching some of the most popular items, so most objects will traverse the network only a small number of times under most usage patterns. This technique suggest a modification to the transaction protocol to improve performance, to surpass in some scenarios even the performance of the original model: for operations on sets of large objects, clients announce the writes in the transaction log, but include only a hash of the large object content. This way, since the large object content is excluded from the transaction log, clients will not download the large objects at all, unless they are specifically needed for a query.

The only modification to the transaction protocol necessary to perform this operation is that clients include the object ID and a hash of the object content, as the content in the transaction field. Thus, the link (with a checksum and version) to the object is the stored content in the log and databases, and the object itself is an external entity. Clients treat the external object as if the updates occur when the link occurs in the transaction log, with naturally following semantics for transaction aborts and so forth.

Large object references with PIR A Private Information Retrieval algorithm can be used to retrieve these large objects without revealing which objects are being retrieved, provided that the PIR algorithm does not reveal the size of the object, or that the size of the object is not unique enough to allow an access pattern privacy-defeating correlation between the objects. The advantage of the overall scheme in this context is that access pattern privacy is preserved efficiently for the bulk of the computation; when large objects are retrieved (presumably less frequently), the more expensive PIR is employed to preserve access pattern privacy.

The key to the practicality of all of these alternatives is that all the database indexes required to satisfy a particular query can fit simultaneously on a client, and that the client has enough working memory to perform the necessary joins efficiently. In practice I believe many databases are of a suitable form, with the bulk of the space consumed by large objects that do not need to be retrieved to compute joins.

9.5.4 Expiring Slots

There is a potential denial of service behavior if a client reserves a transaction slot but never commits; no transactions past this slot will be applied. A potential solution is using "mortal locks" that expire.

The following scenario outlines a method by which clients can safely delete expired locks: a pre-transaction reserved slot is only useful for a predetermined amount of time, specified by the client as it reserves its slot (or set as parameter). Clients timestamp the pre-transaction.

If this time has expired and the transaction is still in the pre-transaction phase, any client is now allowed to abort this transaction. The client desiring to abort the transaction simply issues to the storage provider an abort entry for this slot, which is then appended to the transaction log. The provider ensures that only the abort or the commit are appended to the log. The provider decides race conditions, and one of the operations will fail if both the abort

and commit are issued. Clients can guarantee consistent provider behavior in fulfilling these new obligations, by using transaction hash chains as before: if the (untrusted) provider ever accepts both the abort message and the commit for a particular transaction, it will be obvious from the conflicting hash chains once the provider sends updates out (thus maintaining fork consistency).

9.5.5 Vague Pre-commit

I describe an extension here that allows clients to issue vague pre-commits, determining the final transaction contents only after their request slot has been reserved. This technique allows improved performance in certain conflict-heavy scenarios, by giving clients the flexibility to choose their transaction *after* they are informed of current operations. Clients might choose to modify their transaction to avoid conflicts, as an example.

In the above described lock-free protocol, clients submit a pre-commit indicating their pending transaction, then issue a commit or abort on this transaction after checking for conflicts. With an extension, we can allow the commit version of the transaction to differ from the pre-commit version, adding the following field to the commit message C_i: **description=** The actual transaction to run (instead of the P_i.description.

The only requirement added is that C_i.description be a "subset" of P_i.description. That is, any conflict that the final commit C_i might cause with future transactions would also be caused by the pre-commit P_i.description. With this requirement enforced, all client behavior is identical to what it would have been if the original P_i.description was C_i.description, with the exception that there might be more aborts than otherwise. This requirement is thus sufficient to ensure consistency when clients are honest. In the malicious client scenario, it is additionally required that all clients can determine whether any commit C_i.description is indeed a subset of the pre-commit P_i.description, as they don't trust the issuer to make that declaration.

9.6 Implementation and Experiments

Strawman Implementation (ODP). I built a proof-of-concept strawman implementation of the Outsourced Durability Protocol (ODP) using different components in Java, Python and C. The implementation handles SQL queries and relational data sets and runs on top of MySQL 5.0 [95], though with minor modifications it can support other RDBMS's. The protocol enables parties with low uptime to keep databases synchronized through a single, untrusted third party that has high uptime. Thus we can allow safe outsourcing of both data backups and data synchronization through an untrusted provider.

In my particular setup I aimed towards simplicity rather than performance, giving each client application its own connection to a single database in the client's cluster. These connections are filtered through a proxy, which captures queries for the protocol to ensure proper propagation and conflict avoidance. Each cluster runs a single process that communicates with an untrusted service provider conduit through symmetric XML-RPC channels.

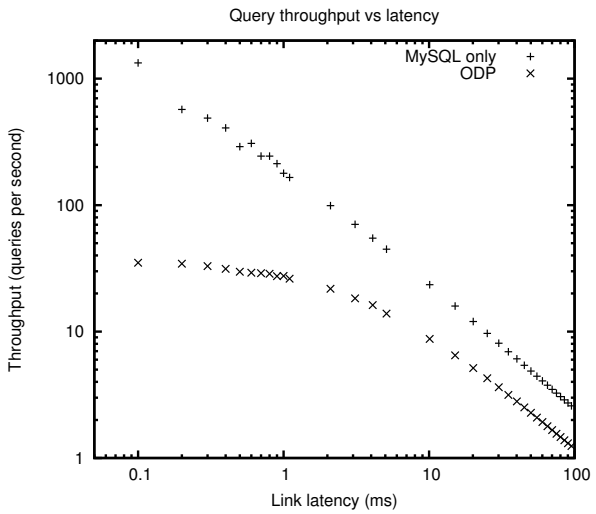

Figure 9.2: Query throughput in transactions per second vs. link latency, with log scale axes. Both MySQL and ODP quickly converge to a relationship inversely proportional to link latency.

To filter queries we use MySQL Proxy [71], an open source scriptable tool built by the creators of MySQL, allowing capture and insertion of SQL queries and database responses. This simple setup shows that we can deploy quickly on existing systems while obtaining reasonable performance; a tailored solution would improve overhead by eliminating the numerous process forks, file writes, and TCP connections initializations in every transaction in the simple strawman implementation.

Strawman: Throughput Experiments. I performed experiments aimed at understanding the throughput behavior of our mechanisms. Given their network-dependent nature we focused on understanding how network characteristics impact performance.

The experimental setup consists of an (untrusted) "server" and several "clients" connected directly through a 1Gbps router. The server is a Dell PowerEdge 2850 running CentOS 4.1 with 4 Dual core Xeons and 4GB RAM, The clients were Lenovo Thinkpads with an Intel Pentium Core 2 Duo 1.8GHz CPU running Red Hat Fedora 9, and Pentium 4 Red Hat Fedora 8 desktop machines. I measured overall throughput in a setting where the two clients simultaneously issued transactions to the server running my ODP software, connecting to a MySQL database through MySQL Proxy [71]. As a baseline control setup I ran the same clients connected directly to the server-hosted MySQL database.

I soon discovered that in this setup the 1Gbps network bandwidth is easily surpassing the processing ability of the baseline, thus we focused mainly on understanding the behavior of ODP vs. baseline MySQL as a function of network latency. To this end I modulated network latency at the kernel level using the NetEm [55] network emulation tool, which delays packets in the outgoing network queue[1]

[1]Effective bandwidth was also slightly decreased by the latency, since the TCP window sizes are fixed.

Figure 9.2 shows the throughput in queries per second obtained using a remote MySQL database with no server guarantees, and the throughput obtained in a strawman ODP implementation with full privacy and correctness assurances. I vary link latency from 0.1ms to 100ms, sampling at growing intervals to suit the log scale X axis.

From Strawman to Efficient Prototype. The strawman ODP implementation could support over 30 queries per second with full assurances. I believe this throughput can be increased by at least one order of magnitude by an industry level prototype which would consider the following bottlenecks of the strawman solution.

Multiple process forks. I used Java to manage all the communication aspects, as its pre-existing constructs reduce coding and debugging time. Additionally, a C-based Lex/Yacc parser was the most natural mechanism to detect conflicts between SQL transactions. To obtain the most functionality in the shortest amount of time, I decided to launch a new Lex/Yacc based conflict detection process from Java for every SQL statement. The result is that I incur several process forks for each processed transaction, launching both a shell and the parser once for each statement in each transaction on each client. Additionally, the conflict detection operates as a separate C-based executable. While process forks themselves are relatively cheap, incurring several in succession while the client waits for the commit creates a low performance cap. I profiled the time required to launch a shell and application at approximately 2ms—this accounts for a large portion of our overhead.

Synchronous client. The MySQL command line and stdin piping was used as our application client. This incurs the full latency of each transaction as a transaction throughput cap. Having two concurrent clients alleviates this slightly, but issuing multiple simultaneous transactions from each client would decrease the impact of latency on throughput. Additionally, part of this benefit can be received by continuing each single-threaded client before the commit has been applied—even at the risk of causing more conflicts, e.g., by creating the possibility for client conflicts with itself.

Multiple TCP connection setups. Instead of reusing client-server TCP connections, the strawman creates a new connection on each request. Multiple requests are constructed per transaction. This design choice is a result of the Java XMLRPC server I am building on; this slowdown can be eliminating by choosing a different XMLRPC implementation.

Java VM overhead. Choosing Java as the implementation language allowed quick prototyping. The disadvantage is that the Java VM (even after just-in-time compilation) can run many (I/O) tasks slower than a streamlined implementation compiled to machine byte code. MySQL, by comparison, is implemented in C.

Lua scripting overhead. MySQL Proxy [71] allows the capture of sessions without building a custom MySQL listener. This allows easy integration with MySQL-enabled applications. The interface to MySQL Proxy is a Lua script parsed at runtime. Application logic in this Lua script runs considerably slower than it would if implemented directly.

Also, a set of costs *cannot* be eliminated due to the core nature of the protocol, including:

- Symmetric key encryption: two symmetric key encryptions, one approximately the size of the transaction description, and the fixed size commit of around 100 bytes. Additionally, this ciphertext must be decrypted on each client.

- Hash function computations, for MACs and hash chains: the overall quantity of data hashed per transactions per client is the size of the transaction description, plus a small amount around 100 bytes.
- Two asynchronous TCP round trips: latency will not affect the throughput of asynchronous (conflict-free) clients. The network bandwidth consumed per transaction is slightly higher than in MySQL, since this we send transactions back out to all clients.
- Proxy costs: organizing hash chains, ordering and transmitting incoming transactions. This program overhead however, is likely always smaller then the actual transaction application times.
- Conflict detection cost: the time required to determine if the order of two transactions affects the outcome. This is an application specific cost—a function of the conflict definition.
- Clients running each individual transaction description: This cost is incurred at the server in a MySQL-only scenario.

Ultimately, in an industry-level prototype, I estimate throughputs of roughly the same order of magnitude as an un-secured MySQL server.

9.7 Conclusions

In this chapter, I introduced a novel paradigm for secure outsourcing of data management primitives, specifically durability and availability with assurances of data confidentiality and access privacy. I designed, implemented, and evaluated a strawman implementation that validates the feasibility of the new paradigm, running at tens of queries per second. I identified key efficiency bottlenecks that can be eliminated in an industry-level prototype to achieve orders of magnitude higher throughputs.

Chapter 10

Untrusted clients

This chapter looks at the question of providing access privacy in a shared database, when not all clients are trusted. I provide only brief excerpts here, referring the reader instead to external work. The first paper shows how to provide the notion of delegation through temporary access privileges. The idea is to provide another party with enough information to perform a single query, at point in the future they choose.

The second paper proposes a model under which mutually non-trusting clients share a single database with access controls. Moreover, it shows how to hide even the sets of clients that are sharing data.

10.1 Oblivious Outsourced Storage with Delegation

The following is an excerpt from the paper "Oblivious Outsourced Storage with Delegation" I co-authored with Franz et al., presented in 2011 [33].

Abstract. In the past few years, outsourcing private data to untrusted servers has become an important challenge. This raises severe questions concerning the security and privacy of the data on the external storage. In this paper we consider a scenario where multiple clients want to share data on a server, while hiding all access patterns. We propose here a first solution to this problem based on Oblivious RAM (ORAM) techniques. Data owners can delegate rights to external new clients enabling them to privately access portions of the outsourced data served by a curious server. Our solution is as efficient as the underlying ORAM constructs and allows for delegated read or write access while ensuring strong guarantees for the privacy of the outsourced data. The server does not learn anything about client access patterns while clients do not learn anything more than what their delegated rights permit.

Introduction

As data management is increasingly being outsourced to third party "cloud" providers such as Google, Amazon or Microsoft, enabling secure, distributed access to outsourced data becomes essential. This raises new requirements concerning the privacy of the outsourced

data with respect to the external storage, network traffic observers or even collaborators who might have access to parts of the outsourced database. In this scenario, a data owner O outsources his data items to a server S. At a later time, he wishes to delegate read- or write-access to individual data items to third party clients C_1 , ..., C_n . Since the data is potentially privacy sensitive, strong confidentiality and privacy guarantees should be in place. Clients should only be able to access those items they are given access to. Moreover, potential adversaries should be unable to derive information from the observed access patterns to the outsourced database. This is necessary, as even the observation that one item is accessed more frequently than others or the fact that one item is accessed by multiple clients, might leak sensitive information about this particular item. In the particular case where the owner is the sole client accessing the data stored on the server, the problem can be solved by applying techniques from Oblivious RAMs. An ORAM structure preserves not only data confidentiality but also provides privacy for client data accesses. So far, the problem of hiding access patterns in outsourcing database scenarios containing multiple (distrusted) clients is open. In this paper we show that ORAM techniques can be adapted to this scenario as well. To this end, we introduce a new ORAM feature: delegated access. Data owners can delegate controlled access to their outsourced database to third parties, while preserving full access privacy and data confidentiality. Achieving this turns out to be non-trivial: in addition to preserving the owners access privacy, we also need to ensure that (i) the server is unable to learn the access patterns of any of the clients, (ii) no client is able to learn or modify any information of items she cannot access and (iii) no client can learn about the access patterns to items which she cannot access herself.

Applications. We now describe several applications that can be built on the constructions we propose in this paper. Anonymous Banking: The numbered accounts supported by several banks claim to provide user privacy. However, by allowing banks to trace the currency flow and build access pattern statistics, they can be used to learn undesired information, ultimately compromising privacy. The solution proposed in this paper can be used toward preventing such leaks: Account numbers and details are stored as records by the bank and account owners can delegate access rights to other clients as desired. Since the data is accessed obliviously, the bank can learn neither which records are being accessed nor the access rights associated with users. Oblivious Document Sharing: Document sharing applications such as Google Docs suffer from obvious security and privacy shortcomings. Not only is the central storage able to access the cleartext documents but it can also learn access privileges as well as access patterns and exact contributions from individual users. Our solution is the perfect fit for this problem as users encrypt stored documents, privately outsource read and write privileges and obliviously and efficiently access desired documents as allowed by their permissions. Rating Agency Access: Privacy is of paramount importance in financial markets. Public knowledge of investor interest can influence the ratings and prices of company shares in undesired ways. The natural question to ask is, can an investor privately obtain desired information about companies of interest? The solution we provide in this paper answers this question affirmatively. A rating agency maintains an ORAM with records containing ratings and general information for individual companies. Each company

owns its own records and can delegate write access to specialized rating assessing companies and on-demand, read-only access to clients that pay to privately access them.

Contributions. We devise delegated ORAM privacy and security properties, expressing the fact that clients cannot learn any information about items which they are not allowed to access. We provide a first construction of an ORAM with delegation that satisfies this property while preserving the original ORAM privacy properties. The construction relies on a new type of read, write and insert capabilities issued by data owners for items that clients should be able to access. We also show how data owners can efficiently revoke access rights.

10.2 Oblivious Databases without Central Authority

The following is an excerpt from the unpublished paper "Oblivious Databases without Central Authority" I co-authored with Martin Franz, Stefan Katzenbeisser, and Radu Sion.

Abstract. It has been shown both feasible and desirable to obtain access privacy, the ability to perform reads or writes without revealing to the storage provider any correlation of accesses. Existing work typically assumes one party reading and writing (Oblivious RAM), or multiple parties reading shared data (Private Information Retrieval).

We motivate the scenario where any number of clients can update a shared database according to an access policy, obtaining read- and write-access privacy not just from the provider but from each other. The server sees only a sequence of indistinguishable accesses, and the clients—even colluding with each other—remain oblivious to any accesses outside their permitted access list. While these desired privacy guarantees seem almost contradictory, we find (perhaps surprisingly) a construction with sublinear complexity that achieves just this.

Our novel construction enables sharing of data between clients with strong privacy guarantees previously impossible (or prohibitively expensive) to enforce, opening up the possibility of novel applications through this new data sharing model. We show how to construct an oblivious database with no central authority, i.e. where no party is allowed to see all items stored in the database, or the access patterns of other clients.

Introduction

Databases form the backbone of modern networked applications; collaborating parties frequently outsource private data to a service provider and give other parties selective access (social networks constitute important examples). Given the global utility of such applications, and the often sensitive nature of the stored content, there is the critical need for technologies that protect outsourced data against the database. In currently deployed systems, this demand for privacy is (insufficiently) filled with encryption, such that that the database operator is unable to access the data items in the clear.

However, encryption of database content is not enough to fully protect the privacy of the involved parties: in particular, the sequence of accesses to items in the database (i.e.,

knowledge which client accessed which item at what time) can reveal critical information. For instance, despite encryption a database server can collect statistics on the most "valuable" data items in its possession. Any ability to link this with external information greatly increases the magnitude of the leak; in general, it is difficult to place a satisfactory upper bound on what is revealed to the server about stored data through this side channel. In particular, the server still sees which clients collaborate and how much data they share. Thus, to fully protect the privacy of the involved parties, *access patterns* must be hidden in addition to the actual database contents and collaboration patterns. This holds for *both* the server and the clients accessing the database.

Our goal is to provide a scheme that allows construction of a database, in which multiple clients can *privately* (contents of items are hidden) and *obliviously* (access patterns are hidden) outsource their data to a central database server, while granting each other selective access and preserving privacy of the collaboration patterns. While desirable, these guarantees prove non-trivial to provide at reasonable cost.

Scenario Description. We target a scenario as follows: client parties (e.g., financial analysts, agents and investors) jointly work on some documents containing highly sensitive data (e.g., a company's balance records, countries' national debts, or financial standings). Even the mere interest of an investor or analyst in this data might trigger enormous movements in the markets. Thus, the documents which are stored in a central database need to be protected, and it is indispensable to hide who accesses which documents, even from the owner of the database.

In this paper we propose an efficient solution in which k clients share a single database on a single untrusted server. Our solution protects the documents as well as the access patterns of the clients, even when there is no central authority among the clients that runs the database. They operate jointly on their documents, hiding both the access patterns and their contents from unauthorized accesses. Borrowing a theme from Oblivious RAM, the database undergoes periodic reshuffling to maintain indistinguishability between queries. Rather than requiring trust in any single client to be perform the reshuffles, however, our construction shares the trust among a subset of the clients using a well known concept from threshold cryptography.

Chapter 11

Guarantees from Trusted Hardware

Chapter 8 briefly explored the use of trusted hardware in the context of access privacy, specifically as a mechanism of converting an Oblivious RAM solution into a Private Information Retrieval (PIR) solution. This chapter examines trusted hardware from a different perspective: how to design high-performance trusted hardware, with strong privacy and integrity assurances to software. After analyzing what we need from hardware to obtain these security guarantees, we shall look at access privacy. In particular, we explore in Section 11.7.4 how this trusted hardware design can guarantee access pattern privacy to software.

11.1 Introduction

To protect software and data against vulnerabilities and malware, we describe simple extensions to the Power Architecture for running Secure Executables. By using a combination of cryptographic techniques and context labeling in the CPU, these Secure Executables are protected on disk, in memory, and through all stages of execution against malicious or compromised software, and other hardware. Moreover, I show that this can be done efficiently, without significant performance penalty. I take a transparency-focused approach, emphasizing ease of software deployment. Secure Executables can run simultaneously with unprotected executables; existing binaries can be transformed directly into Secure Executables by re-linking. Moreover, Secure Executables can safely make use system calls provided an untrusted operating system. In sum, I show that a simple set of processor modifications suffices to provide secure execution in an untrusted environment, without significant changes to the executable.

Almost every week, we hear of incidents in which systems are compromised and sensitive information is stolen in an Internet-based attack, with victims suffering significant financial harm. There have been many startling examples recently [10, 53, 82, 121]. While I do not believe a single solution can solve the entire problem, part of the comprehensive approach required is to focus on protecting the sensitive information itself. This protection should come in the form of a simple, robust barrier against potential attackers. Moreover, I believe that to be broadly applicable to today's software industry, such a solution needs to work

transparently with existing software, given the prevalence of legacy software and the expense of developing new software.

Given the evident difficulty in creating bug-free operating systems that reliably protect software, hardware support should be employed to minimize the trusted code base. In particular, to achieve a meaningful protection boundary around the sensitive information, the operating system must be outside the trusted base. On the other hand, I believe a small set of architecture extensions is acceptable if they realize guarantees of software protection while avoiding significant performance penalty. Existing work (described below) has looked into this approach; I will identify and attempt to solve the set of issues still preventing these approaches from revolutionizing the security industry.

In summary, a dire need exists for systems and methods to prevent the theft of information from a computer system in an Internet-based attack. In particular, this includes the need to protect sensitive data and software on a computer system from other software, including software that an attacker may be able to introduce into a targeted computer system. I take a fresh look at hardware-based isolation techniques in this chapter, providing a developer-transparent solution for achieving strong protection of sensitive software.

I describe an architecture that protects the confidentiality and integrity of information in an application so that other software or hardware cannot access that information or undetectably tamper with it. In addition to protecting a secured application from attacks from outside the application, the architecture also protects against attempts to introduce malware "inside" the application via attacks such as buffer overflow or stack overflow attacks. Finally, a strong root of trust is established to guarantee confidentiality and attest to the integrity of results. In sum, this architecture provides a foundation for strong end-to-end security in a network or cloud environment.

The architecture, called SecureBlue++, builds upon the IBM SecureBlue [60] secure processor technology. SecureBlue has already been used in tens of millions of CPU chips to protect sensitive information from physical attacks. In a SecureBlue system, information is "in the clear" when it is inside the CPU chip but encrypted whenever it is outside the chip. This encryption protects the confidentiality and integrity of code and data from physical probing or physical tampering.

SecureBlue++ builds upon SecureBlue. Like SecureBlue, we can continue to protect against physical attacks. SecureBlue++, though, uses "fine-grained" SecureBlue-like cryptographic protection that also protects the confidentiality and integrity of information in an application from the *other software* on a system. Moreover, it does this in a way that is transparent to applications.

11.1.1 Overview

SecureBlue++ provides a trusted execution environment for individual applications, protecting an application from other software, *including privileged software* such as the operating system and malware that manages to obtain root privileges. The set of hardware changes described below provides this strong isolation in a way that is nearly transparent. In particular, we can run Secure Executables side-by-side with unprotected software, with only

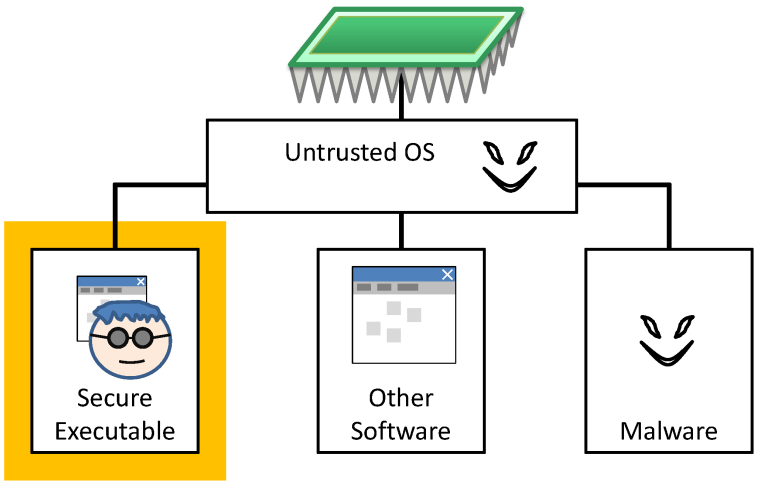

Figure 11.1: Overview: the Secure CPU provides a boundary of protection around the Secure Executable, protecting its confidentiality and integrity from other software, malware, or a compromised operating system.

a small performance impact to the Secure Executable, and no performance impact while running unprotected software. The software source code requires no changes to build a Secure Executable; we simply link with new initialization code and system call handlers. The idea is that the CPU uses encryption and integrity checks to prevent the OS from reading or tampering with application memory. At the same time, it enforces some basic semantics of system calls that allow applications to safely make use of them, even when they are implemented by an untrusted OS.

I show the full set of required hardware changes is limited in scope, and convenient for integration into mainstream CPUs. I begin now with a brief summary to provide the intuition behind how sensitive applications are protected at all stages—on disk, in memory, in cache lines, and in registers. A "Secure Executable" is an executable file, encrypted (aside from the loader) so that only a specific target Secure-Executable-enabled CPU can access it. The included cleartext loader sets up the memory space and enters secure mode, enabling the CPU to decrypt the rest of the executable. Confidentiality and integrity of the encrypted portion of the Secure Executable in memory are protected by keys not available to software. Cache lines are stored in cleartext, but associated with a Secure Executable ID that restricts access of the cache line. Reads and writes to locations in the CPU cache incur no new overhead aside from verifying the cache label. Finally, on interrupts, the contents of the registers are protected from the operating system by moving them to protected memory. Figure 11.1 provides an overview of this approach.

11.1.2 Related Work

There is a long history of using hardware features to isolate software from a compromised operating system, starting with XOM [76]. Figure 11.2 compares Secure Executables to

	SCPU (IBM 4758)	Flicker	TrustVisor	SP Secret-Protecting Architecture	XOM	Aegis	Overshadow	Secure Executables
Requirements								
Works without Hardware Changes	N	✓	✓	N	N	N	✓	N
No OS in Trusted Code Base	Card OS in TCB	✓	✓	✓	✓	✓	Host OS in TCB	✓
Works without OS Support	N	✓	✓	N	N	N	✓	N
Transparent to Developers	N	N	N	N	N	N	✓	✓
Protection								
Code privacy (transparently)	✓	✓	✓	N	N	✓	N	✓
Resilient to Memory Replay Attacks	✓	✓	✓	N	N	✓	✓	✓
Protection from Physical Attacks	✓	N	N	✓	✓	✓	N	✓
Features								
Multiple Simultaneous Instances	✓	N	N	N	✓	✓	✓	✓
Multi-threading / Multi-core support	✓	N	✓	?	N	N	✓	✓
Support Shared Memory Regions	✓	N	N	N	N	N	N	✓
Virtualization Support	N	N	✓	N	N	N	✓	✓

Figure 11.2: Comparison with existing work

related work.

XOM. XOM [76] is a set of architecture changes that extends the notion of program isolation and separate memory spaces to provide separation from the OS as well. They introduce "secure load" and "secure store" instructions, which tell the CPU to perform integrity verification on the values loaded and stored. For this reason, the *application transparency* is limited: developers must tailor a particular application to this architecture.

Aegis. Aegis [122] fixes an attack in XOM by providing protection against replay attacks. This requires using an integrity tree to protect memory. The authors also offer optimizations specific to hash trees in this environment that greatly reduce the resulting performance overhead. These optimizations include batching multiple operations together for verification, and delaying the verification until the time where the values affect external state. Again, the approach is not transparent to developers: the set of sensitive code areas must be explicitly specified by the developer. Additionally, the developer must identify the specific calculations requiring expensive integrity checkpoints, which are the only places tamper checking is performed. In contrast, SecureBlue++ maintains integrity guarantees across all code sections and computations within the Secure Executable.

Secret-Protecting Architecture. Secret-Protecting Architecture [74] provides a mechanism to run code in a tamper-protected environment. Dwoskin and Lee [30] design a mechanism extending a root of trust to supporting devices. However, this technique does not have the transparency of other techniques. For example, there can be only one Secret-Protected application installed and running at a given point in time. The ability of protected applications to make use of system calls is likewise limited.

TPMs. The Trusted Platform Module, widely deployed on existing consumer machines, can be used to provide guarantees of the integrity of the operating system. It does not exclude the operating system from the trusted code base, however. Instead, the TPM signs the current memory and register state, and it is up to other parties to decide if this signed value constitutes a valid operating system state.

There is a separate class of approaches, including Flicker [87] and TrustVisor [86], using the TPM late-launch feature to protect software from a compromised operating system.

Flicker. Flicker [87] provides a hardware-based protection mechanism that functions on existing, commonly deployed hardware (using mechanisms available in the TPM). As with other solutions, this incurs only minimal performance overhead. It provides a protected execution environment, but without requiring new hardware changes. The trade-off is that software has to be specifically developed to run within this environment. The protected environment is created by locking down the CPU using TPM late-load capabilities, so that the protected software is guaranteed to be the only software running at this point. System calls and multi-threading in particular do not work for this reason. Moreover, hardware interrupts are disabled, so the OS is suspended while the protected software is running. Thus, software targeted for Flicker must be written to spend only short durations inside the protected environment. The advantage of Flicker is the reduced hardware requirements: it is supported by existing TPM-capable processors.

TrustVisor. TrustVisor [86] guarantees software isolation using a trusted hypervisor. A root of trust is established using hardware features of recent Intel and AMD processors to attest to the integrity of the hypervisor. Simultaneously, TrustVisor dramatically improves performance over Flicker by using the hypervisor—which has the ability to make physical pages inaccessible by the OS—to do the loading and measuring of security-sensitive regions of the application. For comparison, Flicker required TPM measurements at every security-sensitive region entrance and exit. Speeding up this task is critical since the system operates with interrupts disabled while running the security-sensitive application regions. TrustVisor does not consider physical attacks, but it does protect against DMA (Direct Memory Access) attacks by compromised devices.

Overshadow. Overshadow [22] provides guarantees of integrity and privacy protection similar to AEGIS, but implemented in a virtual machine monitor instead of in hardware. This approach has several other advantages as well; making the implementation transparent to software developers—for example, software shims are added to protected processes to handle system calls seamlessly. This means, however, that there is no protection provided against a malicious host OS or against physical attacks. Nonetheless, the authors provide an updated approach that incorporates techniques we can adapt to our hardware-based-protection model. One example is the use of system call wrappers, which transparently manage system calls from a secured application, otherwise unmodified.

11.1.3 New Contributions

I detail several novel contributions in this chapter, addressing components missing from much of the related work. Moreover, I believe these features need to be provided by any secure architecture that will achieve widespread adoption. See Figure 11.2 for a comparison with existing work.

- minimal size of trusted computing base (TCB). In SecureBlue++, the TCB consists of just the secured application and the hardware. Relying on the security of any other components makes it much more difficult to obtain reasonable security guarantees.
- minimal changes to hardware and software, including minimal changes to OS. Instead of defining a new hardware architecture altogether, I provide a small set of changes that can be applied to existing architectures, without affecting other software running on the system.
- transparency to applications. Developers do not need to design programs any differently in order to make Secure Executables. An existing program can quickly and easily be rebuilt into a Secure Executable (not even requiring recompilation from source code). I believe this is key to enabling widespread adoption.
- transparency with respect to build tools. There are no language changes required to take advantage of Secure Executables, and we remain compatible with existing compilers and link editors. I do introduce a new linking step that links system call wrapper code in with an existing application, and a final step to encrypt the application and enable Secure Mode.
- protecting confidentiality and integrity of sensitive information. We will encrypt the Secure Executable binary, so sensitive information is protected even in the file system before execution. The architecture establishes a root of trust guaranteeing the integrity of a Secure Executable, ensuring its sensitive information is readable only by the target CPU, and only during execution of that Secure Executable.
- shared memory. SecureBlue++ provides a mechanism allowing seamless sharing of memory regions between Secure Executables. Furthermore, as long as values stay in the cache, this sharing avoids the need for any cryptographic operations.
- multi-threading. SecureBlue++ supports multiple threads (again, seamlessly) for a single Secure Executable. This is a critical feature if the architecture is to be used by modern day applications.
- virtual machine support. Our architecture is compatible with both type-1 and type-2 virtual machines, and can be used to establish a root of trust all the way up to the physical CPU.

I now describe the precise security guarantees I wish to achieve, and the architecture changes necessary to achieve them.

11.1.4 Model

A user wishes to run a particular application (the **Secure Executable**) with integrity and confidentiality guarantees. The target system is simultaneously executing untrusted software, including a potentially untrusted or compromised operating system. This chapter develops a set of extensions to the CPU running on this untrusted system that make these integrity and confidentiality guarantees achievable. The specific guarantees are revisited again at the end of this section.

We will employ standard cryptographic constructions, using hashes, symmetric encryption, and public-key encryption to maintain the integrity and privacy of information. I do not specify the particular constructions to use at this point, however; potential candidates include AES, and RSA with 1024 bit keys. Since the security of our mechanisms rely on these cryptographic mechanisms, I assume our adversaries are unable to defeat the employed cryptographic encryption primitives.

We shall consider an attacker who has full control of the software running on the system, aside from the Secure Executable. The protection of memory regions established at launch leads naturally to enforcement of no-write and no-execute regions, which is useful for discouraging buffer overflow and stack writing. However, we do not otherwise protect the Secure Executable from itself: vulnerabilities in the Secure Executable will remain exploitable. We significantly reduce the size of the trusted code base—instead of trusting the entire OS, developers merely have to trust the code that they have control over. I believe a Secure Executable can be much smaller, and consequently more easily verifiable, than an entire operating system.

The attacker may also have physical access to the system, and we protect against attacks to this memory, disk, buses, and devices. However, I assume that it is beyond the abilities of the attacker to successfully penetrate the CPU boundary. That is, the gates and registers inside the CPU are not considered subject to tampering or leak. I believe this is a reasonable assumption for most scenarios and adversaries. However, in situations where it is not, there are existing physical protection techniques, such as those employed on the IBM 4764 [59] that can be used to increase the cost of physical attacks against the CPU. These techniques typically result in increased CPU cost and/or decreased performance due to limited heat transfer capacity.

It is beyond the scope of this work to consider side channel attacks such as power analysis and timing attacks. We will, however, briefly consider access pattern privacy in Section 11.7.4. This section explores ways of efficiently hiding the addresses of the accessed memory.

The mechanism I provide consists of a set of changes to a CPU architecture (in this chapter I address the POWER architecture specifically, but the techniques can be adapted to most modern CPU architectures). Four new instructions enable execution of a Secure Executable, under which decryption and execution of protected code is possible, while outside tampering or snooping is simultaneously prevented.

11.1.5 Integrity and Confidentiality Guarantees

We consider here informal definitions of the desired security guarantees.

Application Integrity. A polynomially-bounded adversary has control over all values read from external memory and all software on the system. An architecture provides *application integrity* if the adversary's ability to tamper with the code or data of a secure executable (that is, to get the Secure Executable to operate out of the range of behaviors allowed by its source code), taken over the possible choices of Secure Executable secret keys, is negligible.

Defining confidentiality is more difficult. It might be desirable to provide a confidentiality guarantee such as that a fully malicious adversary gains no advantage at guessing the code or data of a secure executable through physical access to the system, and through specifying all software running on the system besides the secure executable. This could be captured with a game in which the adversary specifies two Secure Executables, observes the system, and guesses which of the two is run.

This is definition is not helpful, however, since any secure executable performing useful computation trivially reveals itself to the adversary if allowed to even output a result or interact with its environment. Moreover, the operating system in any practical construction still has the ability to observe system calls (the inputs and outputs to the Secure Executables), and the pattern of memory locations accessed. A narrower definition is needed, closer to the model we are working within, to capture the notion that an adversary gains no advantage from compromising the operating system.

Application Data Confidentiality. An architecture provides *application data confidentiality* if the advantage of any polynomially-bounded adversary, with control over all values read from external memory and all software on the system, to answer any question about the observed Secure Executable, is negligibly better than the advantage of an adversary that observes only the sequence of system calls, memory access pattern (but not *contents*), file size, inputs, and outputs.

11.2 Approach

In our design, sensitive information is cryptographically protected whenever it is outside the CPU chip. It is available in cleartext inside the chip but only by the software that "owns" the sensitive information. The sensitive information is decrypted as it moves from external memory into the chip, and encrypted as it moves from the chip into external memory. Integrity values are also checked when information moves into the chip and updated when information moves back out. This approach is illustrated in Figure 11.3.

A public/private key pair is installed in the CPU at manufacture time, with the public portion signed by the factory. This is used to establish the root of trust, allowing Secure Executable binaries to be encrypted so that only a target CPU can access the embedded code and data.

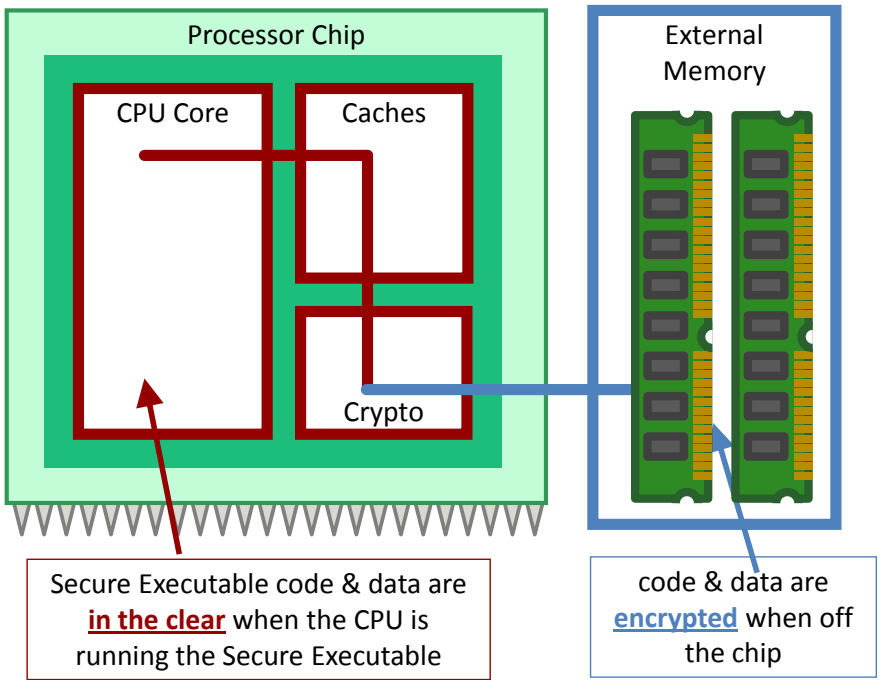

Figure 11.3: The main idea: sensitive information is cryptographically protected whenever it is outside the CPU chip, and visible as cleartext inside the chip, but only by the software that owns the sensitive information.

11.2.1 New Hardware Logic

Memory encryption. Encryption/decryption is performed between cache and main memory. Cache lines corresponding to the protected region are decrypted as they are brought into the cache, and encrypted as they are evicted.

Integrity tree verification/update. A cache line is verified using the integrity tree when data is brought into the cache. This may require performing additional fetches to make sure enough of the tree is present to perform this verification. When lines are evicted from the cache, the corresponding parts of the integrity tree are updated.

Cache line label. On loads from a protected region, the ID of the Secure Executable associated with that cache line (linked indirectly, though the Memory Region ID described later) is compared to the current Secure Executable ID. If there is a mismatch, the access is treated as a cache miss. On stores, the cache line Secure Executable ID is updated to match the currently running Secure Executable.

Register protection. The CPU ensures that application registers are not visible, or undetectably modified while the Secure Executable is suspended. The two obvious approaches are to encrypt the registers before giving control to the OS interrupt handler, or to store the registers on chip while the Secure Executable is suspended. Both approaches have drawbacks. Encryption takes time, and is potentially vulnerable to replay attacks unless a hash

is stored on chip; storing a copy of the registers on chip uses valuable space and severely limits the number of supported simultaneous suspended Secure Executables.

We choose instead to store the register set in protected memory corresponding to each Secure Executable (allocated by the loader code). The registers end up protected in the cache (and ultimately by the Secure Executable root hash value, if these values get evicted from the cache). Only the Secure Executable root hash value, plus the metadata required to use this hash value, are stored on-chip. This eliminates the performance penalty of the cryptographic operations, while greatly reducing the amount of required on-chip storage.

11.2.2 Software Changes

Because software transparency is a main goal, we allow existing applications to be transformed into Secure Executables by relinking. This relink process attaches initialization code (Section 11.2.3), reserves portions of the address space for the integrity tree (Section 11.3) and thread restore locations (Section 11.4.3), attaches system call wrapper code (Section 11.5), and reserves a portion of the application memory address space for use during launch as follows.

A metadata region is stored in the Secure Executable application memory space. It is used to register initial parameters with the CPU, as well as establish memory regions. It is encrypted with the Executable Key, and protected by a keyed hash, the initial value of which is loaded on chip with the Enter Secure Mode instruction (described below). Updates to this region require the CPU to re-compute the metadata hash before the next interrupt can be dispatched.

- Metadata region size
- Code entry point
- Signal handler entry point
- Core dump data key
- Location of the protected memory region table (within metadata region)
- Location of the thread restore list (outside metadata region)

The `metadata region size` specifies the total size of this region. The `code entry point` is a 64-bit memory location identifying the next instruction the CPU should execute after running `ESM`. The `signal handler entry point` is a 64-bit memory location identifying the first instruction of the Secure Executable signal handler. This is the only location the OS can jump to in this program; see the discussion of signal handling, under system calls, for more information on the use of this value. The `core dump data key` is an AES encryption key, that if non-zero, will be used by the CPU to encrypt the dynamically generated data keys and export to software when requested by the OS (in case of a core dump). This allows

software that has access to the core dump data key to decrypt the protected memory regions in the core dump.

Finally, two more values specify the location of data structures maintained by the helper library linked with the Secure Executable. The protected memory region table keeps track of the ranges of protected memory, and the memory region ID corresponding to each. This fixed-size list is stored on chip every time the Secure Executable is entered (e.g., with the RestoreContext instruction). This is necessary to allow quick translation from memory location to Region ID on memory accesses. The thread restore list keeps track of the re-entry points into the Secure Executable, used with the RestoreContext instruction.

11.2.3 Launching

At run time, when the Secure Executable is to be launched, the OS executes the Secure Executable binary, as a normal ELF file. The ELF file consists of a protected region and a cleartext region. The confidentiality of the protected region is protected using a symmetric encryption algorithm; the integrity of the protected region is protected by an integrity tree in the cleartext region. The cleartext region also includes loader code and an ESM instruction. The loader code copies the initial integrity tree into memory. It then issues the `ESM` instruction, which enables decryption of protected regions, and jumps to the entry point (identified in the ESM operand). Integrity and confidentiality protection of memory accesses is also enabled at this point. The ELF application binary format is illustrated in Figure 11.4.

11.2.4 Root of Trust

The root of trust is established as illustrated in Figure 11.5. In the factory, a hardware private key (1) is embedded in the CPU. [1] The public portion of this key is signed by the manufacturer, producing a public key certificate (2) asserting that this private key corresponds to a CPU supporting Secure Executables. To build a Secure Executable for this chip, the Secure Executable decryption is encrypted using the signed public key (3). Finally, the Secure Executable code itself is encrypted with this executable key. This ensures that knowledge of the hardware-installed private key is necessary to access the sensitive code and data distributed through the Secure Executable.

11.3 Integrity Tree

An integrity tree consists of a set of encrypted hash values of memory locations, with these hashes themselves protected by parent hashes. This tree is used to protect memory at runtime. The branching factor of this tree is taken to be the number of hashes that fit in a cache line. For 128-byte cache lines, and 128-bit hash values, for example, each node of the tree protects 8 nodes directly below.

[1] While I do not specify the details here, Physically Unclonable Functions (PUFs) [123] provide a convenient and inexpensive mechanism to do just this.

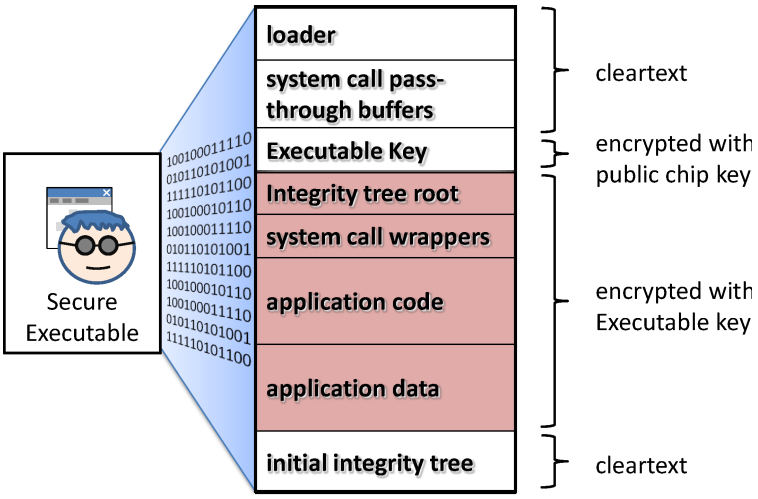

Figure 11.4: The Secure Executable as a binary file on disk: regions and parameters

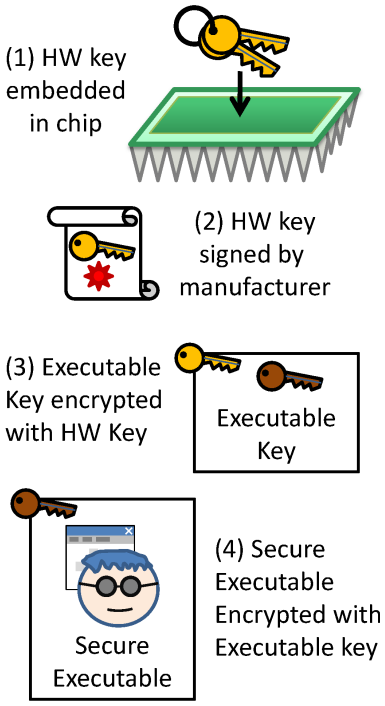

Figure 11.5: Establishing the Root of Trust

The initial value of the integrity root is signed and loaded with the Enter Secure Mode (`ESM`) instruction, used to start of the Secure Executable. This root ensures that the CPU will only use an intact version of the code; the CPU provides the guarantee that only integrity-verified words are visible to the application in regions of the protected address space.

Ancestors of a dirty cache line are pinned in the cache to avoid updating the tree all the way to the top on each write. Evicting a node from the cache requires updating its parent, which might require evicting another node. By locking all ancestors in the cache, however, it suffices to simply update the parent of a node (assuming it can be brought into the cache). Thus, every dirty line in the cache has all the ancestors pinned in with a reference counter.

This requires considering the case in which a parent cannot be brought in, because all the lines where it could go are already pinned. In this case, the tree computation must be performed immediately, as far up the tree as necessary (until a line whose ancestors all fit is found). The procedure to perform a store from a Secure Executable context is as follows. First, bring ancestors in, starting at top (if they're not already there). Lock each parent when one of its children comes in, by incrementing its reference counter. If a child cannot be brought in, compute the path up to this point, reading and writing all the children directly to/from memory. Then the parent is updated to reflect the new value, and marked as dirty in the cache.

This approach has the property that the CPU can evict a cache line with a zero reference count at any time, with one hash operation but requiring no additional loads or stores. Lines

representing instructions or data can always be evicted, as can nodes of the integrity tree whose protected lines are no longer in the cache. Flushing the cache requires starting with all the lines with a zero reference count, evicting these, and repeating as many times as necessary (which can be up to the tree height). On each iteration one level of the integrity tree is evicted.

Due to cache locking conflicts, we do not guarantee that a write requires only updating the parent; it could require up to (e.g.) $\log_8(n)$ loads/stores. But this stalls only the current running Secure Executable, and it can be paused and resumed later. This case should be rare, with high enough cache associativity.

This integrity tree needs to be correct over the whole protected address space, even those parts of the address space that have not yet been allocated. Moreover, the parts of the tree covering these unallocated regions should not require initialization, since the protected memory region is potentially much larger than the portion ever accessed. I introduce the notion of a "sparse" integrity tree: one that correctly protects the whole memory region, but is zero everywhere that it protects unallocated memory. In this sparse integrity tree, these zeros protecting unallocated memory are implicit. Memory is not allocated for these integrity values until they are accessed, at which point they are initialized (by the OS) to zero. Implementing this sparse integrity tree requires a property of the integrity hash function: the hash of an empty region must be zero.

A second requirement for this property of integrity tree correctness over uninitialized data is that the *cleartext* corresponding to uninitialized memory is also zero. To achieve this, we consider all memory locations protected by a hash value of zero to be zero themselves.

11.3.1 Overcommit

Since we are targeting a 64-bit architecture, the size of protected memory regions can be large. For this reason, we need a simple way to dynamically grow the integrity tree. The Linux overcommit mechanism allows us to reserve room in the address space corresponding to a tree protecting large memory regions, while delaying the physical memory allocation until each page is accessed. In overcommit mode, available in recent Linux kernels, application memory allocations always succeed, even if there is not enough physical memory or swap space to support the entire allocation. That is, pages are not mapped to physical memory until they are accessed. This OS mechanism is particularly useful in 64-bit mode, since the size of the address space we are protecting is potentially much larger than the available physical memory. Without overcommit, applications would need to specify a limit on the total memory use beforehand, since we must use a much smaller integrity tree (that can fit entirely in RAM or swap space).

Since the hash tree is maintained by the CPU, instead of the software, any solution must involve the CPU as little as possible; the CPU in particular should not be involved in memory management, and should not issue memory allocation requests. Overcommit provides a convenient solution: the Secure Executable loader can allocate a portion of the virtual address space large enough to contain a hash tree over the rest of the address space. With overcommit enabled, this can be much larger than the available physical memory

plus swap space. When the CPU attempts to access an address in this range, a page fault is generated, since this address has not yet been assigned to physical memory. Since the virtual address has been reserved by the Secure Executable process (with the mmap syscall), the OS handles this fault transparently by assigning a physical page to this location. This provides exactly the behavior we desire, with no changes to the OS (aside from enabling overcommit).

Downsides to using overcommit. When overcommit is enabled, processes run the risk of running out of physical memory at any time—and being targeted by the kernel's out-of-memory killer. A potential workaround described here prevents this from occurring. That is, to guarantee that when the machine is out of memory, only new allocations will fail (matching traditional behavior when physical memory is fully consumed), rather than leaving the potential for any process at all to be killed. The OS provides a new type of memory allocation, specifically for integrity trees. Processes are still allowed to allocate the large integrity tree, but the OS is now aware that the total space used by this integrity tree will stay within a fraction of the rest of the process's used memory. Thus, instead of overcommitting to the entire allocation, the OS is only committing to the normally allocated memory plus a constant fraction.

11.3.2 Integrity Tree Extensions

The integrity tree provides complete integrity protection for a reasonable performance overhead. However, in certain scenarios the entire range of integrity protection may not be necessary. It may be desirable to have a more limited form of integrity protection at a significantly reduced cost. In particular, by storing keyed checksums of the data, but no integrity tree, we can prevent all integrity attacks except for rollback attacks. This means the only modification an adversary can get away with undetected is restoring a given block of data to an earlier state. For read-only segments of memory this is not a problem. For example, the adversary cannot modify undetectably the read-only code segments since there is no other version of these to successfully roll back to. This is the level of protection that XOM [76] provides; Suh et al. [122] explain how XOM protection is vulnerable to such replay attacks.

11.3.3 Integrity Tree Optimizations

A more sophisticated layout of the integrity tree can minimize the number of memory pages needed to cover a particular region. The idea is that each page of memory is organized as a local breadth-first tree, containing only nodes under the first entry in the page. The next page begins with the first node (according to global breadth-first order) not yet included. The calculation to determine where in memory a given node is more complicated in this case; we shall favor simplicity of design by simply using the global breadth first layout.

11.4 Extensions

11.4.1 Memory Regions and Shared Memory

The ability to share protected memory regions between Secure Executables is critical. This could potentially be implemented in software using cryptography. However, to achieve transparency and avoid the need to intercept all memory writes, it is still desirable to have hardware support (even if the cryptographic performance were not a concern). There are two issues to consider with respect to sharing memory: how to provide this mechanism transparently to the application, and how to implement it efficiently.

For common cases of inter-process communication, we automatically create the protected shared memory region. One case is anonymous pipes opened before a `fork`: it is apparent these two Secure Executables need to communicate. The custom system call wrapper sets up the shared memory region over which to implement the pipe. Additionally, on `fork`, the software system call wrapper assigns an MRID (Memory region ID) to all anonymous memory regions. Memory-mapped files cannot be shared securely and transparently without new semantics, however, since it is not known ahead of time what processes and Secure Executables will be accessing this memory-mapped file.

To allow multiple protected regions per application, with various share options, these protected regions are registered at runtime by the applications, by adding an entry to the Protected Memory Region table.

First we consider the hardware implementation of shared memory. Tagging the cache line with the Secure Executable ID of all sharing processes is insufficient, since any number of Secure Executables will potentially share this location. Instead, the cache line is tagged with the Memory Region ID. This `MRID` indexes into an on-chip table containing information allowing efficient verification of the current Secure Executable's permission to access this shared memory region.

We keep only a minimal amount of mapping information stored on-chip in a shared memory region table. This mapping contains two Secure Executable IDs (SEIDs): one for the creator, and one for the first sharer. This allows handling the most common memory region cases (private, or shared among two processes) without penalty. To handle the rest of the cases, a secret is associated with the memory region. Secure Executables are allowed access to this region if the secret is installed at a registered location in the SE memory.

The MRID of the cache line simply points to an entry in the Protected Memory table that indicates both the integrity root and Memory Region secret. Access is verified by checking the `SEID1` and `SEID2` of the Memory Region. Failing this, the secret corresponding to this region is compared at the registered virtual memory location of the current process with the secret in the on-chip memory region table.

The CPU needs to determine whether an access is to a protected region, and to which region. Since this determination needs to be made on every access resulting in a cache miss, the information needs to be stored on chip. We reserve a location in the Secure Executable metadata region to serve this purpose. On every context switch into the Secure Executable, the mapping table linking memory locations to the appropriate `MRID` is loaded into this

hardware table.

To protect the integrity of these shared memory regions, Secure Executables share the integrity tree. This is supported by the OS as a standard shared memory region; the virtual pages for each Secure Executable participating in the sharing are mapped to the same physical page.

11.4.2 Memory Policies: Protecting the Secure Executable from Itself

The mechanisms used to register memory regions and support shared memory lend themselves readily to enforcement of no-write and no-execute policies over specific regions. While modern processes already support page-based access flags restricting writing and execution, they cannot prevent a compromised operating system from tampering with these flags. This, of course, is not an issue in standard processors, because without a Secure Executables approach, a compromised operating system already leads to compromised applications.

Secure Executable support for such policies (e.g. write XOR execute), to protect against attacks such as stack overflow and buffer overflow, includes new parameters to the ESM instruction. These set up the address ranges for an application. One of these will tell the CPU hardware that the address range corresponding to the application's code is, from the application's perspective, "read-only". The other will tell the hardware that the address range corresponding to the application's data, stack and heap is "no-execute". Thus if an attacker attempts to exploit a bug in an application so that the application will attempt to write into the code region or execute instructions from the data region, the attempt will fail.

11.4.3 Multi-threading

We must distinguish here between user threads and kernel threads. User threads are implemented in user space; that is, a user space dispatcher maintains and schedules the threads. This means user thread libraries typically do not support true concurrency: only one user thread runs at a time. Kernel threads are instead maintained and scheduled by the operating system. At the price of slightly increased thread switching latency (due to kernel-user context switches), kernel threads allow true concurrency on multi-threaded, multi-core, or multi-chip systems.

User threads succeed in our model without any additional changes, since the previously described protection mechanisms protect the user-space dispatcher compiled into the Secure Executable. Kernel threads require CPU awareness, since this means a given Secure Executable must support multiple simultaneous saved contexts; the OS scheduler is allowed to choose between several available contexts for a given Secure Executable.

Kernel thread creation via the `clone` system call creates a new context restore point equivalent to the current running one, except for a different stack pointer. Our system call wrapper takes care of the bulk of the work in creating a new thread, then registers a new

stack pointer with the OS and the CPU. Registration with the OS works as in existing POSIX systems, by using the `clone` system call.

Using part of the application protected memory space to store register restore sections provides a convenient method to support threads. The thread is registered with the CPU by creating a new entry in the protected memory region of this Secure Executable corresponding to the restorable register sets. Then, the OS kernel can switch to this context by providing the memory location of the thread state.

The Secure Executable meta data region contains a pointer to the thread restore table. This thread restore table contains the following rows:

- Restorable flag, to prevent the OS from reusing the restore point
- Register set (including program counter)

Since threads share a common memory image, all context restore points associated with a given SE share the same integrity root. There is only one entry in the hardware Secure Executable table, covering all threads belonging to the Secure Executable. Thus, the thread count is not subject to hardware limits, while the number of simultaneous Secure Executables is limited to the size of the hardware Secure Executable table. The CPU is not even directly aware of the number of running threads for a given Secure Executable; the OS merely provides it with the restore point, which it validates by checking that it is a valid offset into the thread restore region.

This model works for multi-core CPUs as well; of course, the cores need to maintain coherency. This can be achieved, for example, by enforcing exclusive access to a shared Secure Executable table. Running multiple threads within a SE across multiple CPUs requires a more sophisticated coherency protocol.

11.5 System Calls

Since we prevent any other process from reading values in our Secure Executable's registers or address space, we naturally need a new mechanism to support passing data in system calls. Nevertheless, we support system calls transparently to both the Secure Executable and the operating system. I describe two mechanisms here to support system calls transparently. To make both approaches transparent to the operating system, the CPU leaves the general purpose registers alone on a system call interrupt, so that these registers can still be used to pass parameters to the OS system call handler. Recall that on all other interrupts, the registers are hidden before control is giving to the interrupt handler. For this reason, both approaches move the application registers to protected memory before invoking the system call.

11.5.1 Approach: System Call Wrappers

I use this approach in my proof of concept implementation. The application is linked with a customized version of libc, that replaces system calls with wrappers. These wrappers serve

the purpose of copying the system call parameters from the protected memory region to the unprotected, OS-readable region.

On invocation, each wrapper performs the appropriate copy, depending on the parameter types. Note that in order to correctly copy the system call parameters, the wrappers have to be aware of each system-call. The wrapper then copies the general purpose registers to the stack, and zeros them, aside from the ones being used as parameters in the system call. Each wrapper then invokes the appropriate system call, and copies the result back into the protected region. Finally, it restores the registers from the stack.

Naturally we must consider the privacy and integrity of the parameters while they are in unprotected memory. We necessarily give the OS the ability to read the parameters and choose arbitrary return values. There is nothing we can do to ensure the OS is correctly handling these system calls; thus we leave it to the Secure Executable application code to make the appropriate decisions about the information it passes to the system call. For example, we do not prevent the Secure Executable from printing out sensitive information if that is what it is programmed to do.

11.5.2 Approach: New System Call Instruction

This approach allows for better application transparency, by avoiding the requirement for a modified version of libc (and thus, now supporting executables that are not even linked with libc). We still add a layer to the system call invocation, but rather than adding this layer as a set of wrappers sitting between the Secure Executable and its system call invocation, we add the layer after the invocation.

To do this we employ two system call instructions, illustrated in Figure 11.6. The first, existing PowerPC `sc` (System Call) instruction behaves as today (passing control to the OS system call handler), unless we are in Secure Mode. If we are in Secure Mode, this instruction instead returns control to a previously registered location within the application. A new PowerPC instruction, `sesc` (Secure Executable System Call) gives control to the OS system call handler, *regardless of whether we are in Secure Mode.*

Now, the Secure Executable registers a system call wrapper to the CPU, so that when `sc` is issued, the system call wrapper gets control. The system call wrapper performs the equivalent memory copies / register cleanup as the previous approach, then invokes the new system call instruction `sesc` to give control to the operating system.

The advantage of the second approach is that we do not have to modify libc (thus eliminating part of the complexity of turning a conventional binary into a Secure Executable). The system call wrapper still needs to be located somewhere in the application address space, and registered by the Secure Executable loader as a parameter to the `ESM` operand. I believe, however, that this address space modification is more application-agnostic than modifying the linked libc. For example, this transformation can be performed directly on a statically compiled binary. Moreover, this modification to the address space and entry point are also performed in other steps as part of the Secure Executable build process.

In both approaches, the CPU needs to keep track of the valid return points; the OS is not allowed to jump to arbitrary positions within the Secure Executable. At any time, the

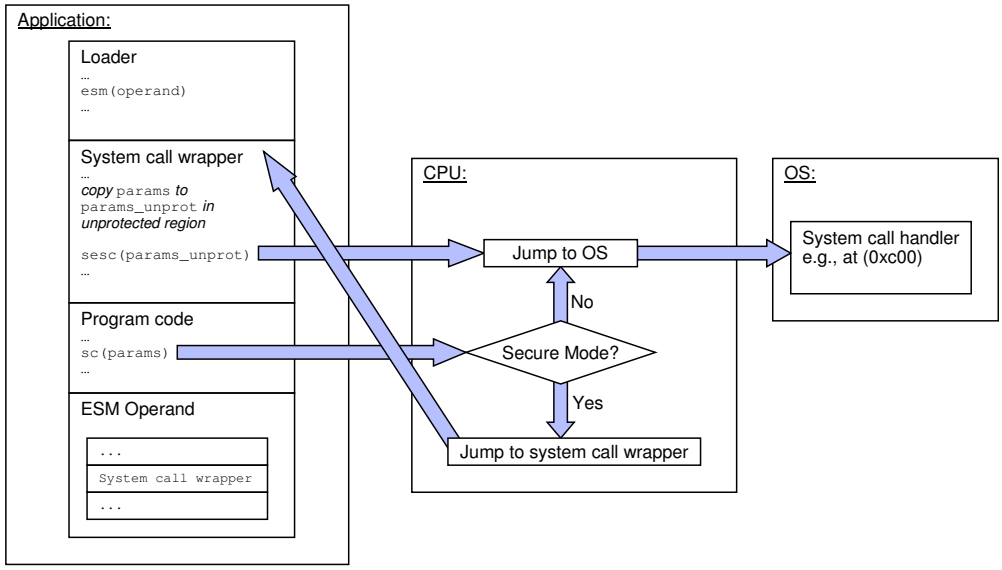

Figure 11.6: `sesc` (Secure Executable System call) handling. The idea is that a standard system call instruction (`sc`) gets intercepted by the CPU and redirected to the registered Secure Executable software system call wrapper. This wrapper, when ready, issues `sesc`. At this point, the CPU passes control on to the OS system call handler.

OS can redirect control to one of a fixed set of locations: the program counter associated with a thread restore point, or the registered signal handler (see handling the `signal` call below). In the case a system call is being run (the `InSystemCall` flag is set in the restore point), the OS is also allowed to redirect control to the restore point *plus 4*. These semantics are used in PowerPC to indicate a system call failure. The CPU sets the `InSystemCall` flag when creating the restore point as `sesc` is executed, and clears it after a `RestoreContext` to that restore point.

11.5.3 Handling Specific System Calls

The majority of system calls are handled in the straightforward manner described above—the only change required is moving the parameters to and return values from the unprotected region to enable communication with the OS. However, several system calls require special support, e.g., those with side-effect semantics that modify the application memory or registers. We now consider specific cases of system calls.

`read`, `write`, `socketcall`, `mmap`, `sbrk`, etc. No special hardware support is needed for those system calls which the OS can handle without looking inside the application. The calls listed above fall into this class. The wrappers merely move parameters and results from and to the protected region as described above. The wrapper must be aware of the size and type of each argument so that it can perform this transformation.

`signal.` The `signal` system call requires specific hardware support in our model, since the OS is no longer allowed to set the program counter to run arbitrary code in the Secure Executable. We handle the signal system call similarly to Overshadow [22]. This entails registering with the CPU a top-level signal handler as part of the `ESM` instruction operand. When a signal occurs, this application-level signal handler within the Secure Executable then dispatches the signal to the appropriate handler within the Secure Executable. The CPU then enforces that the OS can only to reset the program counter of a Secure Executable to its top-level signal handler. This gives the Secure Executable protection of the entry points into its code, so that the OS can not invoke code at arbitrary locations.

The `signal` system call wrapper stores the re-entry point in a table, and replaces the entry point in the system call to point to the top-level signal handler. This way, when the OS dispatches the signal, it gives control to the top-level signal handler. The CPU allows this since this is the registered signal handler of the Secure Executable. This top-level signal handler then dispatches the signal to the appropriate code, as indicated in its local table. This approach is transparent to the OS—it simply appears to the OS that the application is using a single signal handler for all signals.

`fork.` Supporting the `fork` system call requires defining the new semantics. When a Secure Executable process forks, we need to decide whether the new process is still considered part of the old Secure Executable, or whether it should be considered a new Secure Executable. Required considerations include how to preserve traditional `fork` behavior; in particular, the new process should begin with a memory space identical to the old process. POSIX semantics also specify several options with respect to what information should be preserved after the fork, and the system call wrapper must be aware of these options. Since we associate each process with a single Secure Executable, the most natural semantics for fork are to create a new Secure Executable on `fork`.

To obtain copy on write behavior, we use OS support. The OS sends a signal, which is handled by library code. It thusly notifies the Secure Executable when it needs to duplicate a particular page. Each copy requires un-sharing both the page and the regions of the integrity tree protecting those pages.

`clone` **(Threads).** As described in Section 11.4.3, the `clone` system call wrapper registers a new thread restore point in memory.

`exit.` To make sure the Secure Executable does not run unintended code after the `exit` system call has been issued, we wrap the `exit` call in an infinite loop in the system call handler. Thus, if control unexpectedly returns from the system call, no other code is run.

11.5.4 Return Value Validation

Being in the unique position of not trusting the OS, we need to ensure no new vulnerabilities arise out of the OS feeding unexpected return values from the system calls. The return values of the majority of system calls must be checked for sanity, matched up with the passed parameters as necessary. For example, the number of bytes written back to the Secure Executable after a `read` system call must be within the range specified by the caller.

The system call wrapper is in the right position to ensure this, since it runs within the protected region yet sits at the interface between the Secure Executable and the OS. If a violation of expected semantics such as this is detected, the system call wrapper has two options: it can put the response back into the range allowed by semantics if possible, or it can throw an integrity exception. Throwing an integrity exception is preferable to silently changing behavior since it is more visible and indicates either a bug in the system call wrapper validation or the operating system, or malicious operating system behavior.

11.6 Hardware Changes

Recall that our goal is to provide the hardware mechanisms necessary to build a boundary of protection for sensitive applications. I detail a series of architecture extensions here that enable construction of this boundary in a manner that is transparent to existing applications.

11.6.1 New Instructions

Part of the Secure Executable loading process is to issue the ESM (Enter Secure mode) instruction. The ESM instruction allocates a new Secure Executable ID, while enabling memory protection for the regions identified in the instruction operand.

Enter Secure Mode (`ESM`). The instruction itself identifies two registers, R_x and R_y. R_x points to the memory location of the operand. R_y specifies both the size of the operand, and an operand version identifier (to allow backwards compatibility in the future). The size is in the lower 32 bits of R_y.

The operand, starting at memory location R_x, contains the fields depicted in Figure 11.7. Executable Key is an AES encryption key, that has been encrypted with the public System Key. The CPU decrypts this value to obtain the Executable Key, which it then uses to decrypt the rest of the operand. The rest of the operand has been encrypted with the Executable Key.

Finally, ESM integrity value is a cryptographic checksum over the cleartext of the rest of the fields encrypted with the Executable Key (starting with the code entry point). This field is encrypted with the executable key, along with the other ones.

RestoreContext. RestoreContext is a privileged instruction issued by the OS to return control to a Secure Executable, comparable to the PowerPC `rfid` instruction. The OS sets the Secure Executable ID Save/Restore (`SEIDSR`) register, described below, to the value that was present when the Secure Executable was interrupted. Then, when the OS issues the `RestoreContext` instruction, the CPU restores the general purpose registers to the state they were in when this Secure Executable was interrupted, resuming execution. The Secure Executable ID (`SEID`) register (not accessible to software) is restored with these registers, so that cache lines owned by this Secure Executable are once again accessible.

DeleteContext. Another privileged instruction, `DeleteContext` instructs the CPU that the Secure Executable indicated in the `SEIDSR` register has finished running, and the CPU

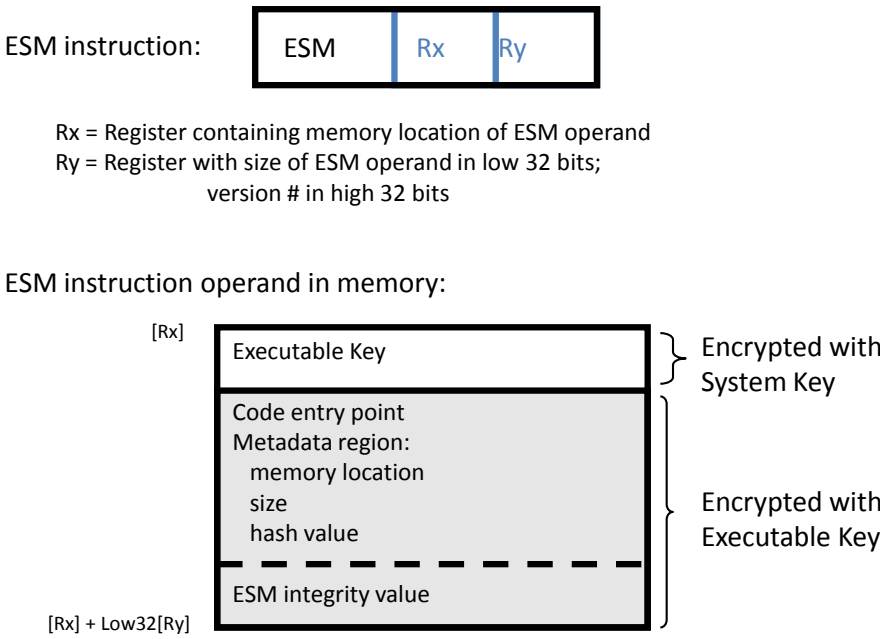

Figure 11.7: `ESM` (Enter Secure Mode) instruction

can free any CPU resources that were allocated to this Secure Executable. Additionally, the CPU clears all cache lines owned by this Secure Executable.

The OS should issue this instruction when the Secure Executable process exits; however there is no vulnerability resulting from the OS choosing to issue this instruction at other times. If the OS issues this before the Secure Executable process exits, all protected values relating to this Secure Executable become inaccessible (since the CPU will no longer allow a `RestoreContext` to this Secure Executable, and the key to access the memory is discarded). This constitutes only a Denial of Service attack, which I do not claim to prevent. If the OS fails to issue this instruction when the process exits, the CPU resources allocated to this Secure Executable remain in use; however other parties can only use this information to resume execution of the Secure Executable where it left off. This results in repeating the instruction that caused the Secure Executable to exit last time it was running. See the description of the `exit` system call in Section 11.5 for an analysis of the case that the Secure Executable is resumed by the OS after exiting.

Secure Executable System Call (`sesc`). This instruction behaves like the existing PowerPC `sc` instruction: it throws an interrupt, giving control to the OS system call handler. However, we will modify `sc` to give control instead to a Secure Executable-registered system call handler. Thus, I introduce `sesc` to obtain the original `sc` behavior when executing inside a Secure Executable. See the description in Section 11.5 for more details.

11.6.2 New Registers

Secure Executable ID Save/Restore(`SEIDSR`). This register (privileged access) is used by the OS to save and restore the ID of the current Secure Executable. It is set when a Secure Executable is interrupted, and is accessed by the `RestoreContext` and `DeleteContext` instructions.

11.6.3 New State

Current Secure Executable ID. This state is not accessible by software, and contains the ID of the currently running Secure Executable. Software can only read or write it indirectly, through the `SEIDSR` register on the `RestoreContext` instruction.

Metadata corresponding to the current Secure Executable. This information is loaded from the Secure Executable metadata region on every return from interrupt. This region is protected by a single keyed hash value, which is stored on chip at all times in the Secure Executable table.

This information includes the following:

- Core dump data key
- Signal handler
- Memory address of the thread restore list
- The memory region mapping table

The memory region mapping table corresponds to the current Secure Executable. This allows quick mapping (on cache misses) from memory location to the memory region ID. Each row in the memory region mapping table has the following fields. This table is simply a read-only cache information stored elsewhere in memory.

- logical memory location and size: to quickly test which region an access corresponds to.
- Share Secret: for quick verification of the MRID share secret on regions shared by more than two Secure Executables
- memory region ID (MRID): index into the Protected Memory Table.

Private System Key. This information also cannot be accessed by software; it is used by the CPU only to decrypt the Executable Key in the `ESM` operand.

Cache Line MRIDs. Cache lines are labeled with the ID of the associated Memory Region.

Secure Executable Table.

- `SEID` Secure Executable ID (implicit: determined by the table position)
- Metadata hash
- Metadata region location.

Protected Memory Table. For each protected memory region, the following is stored on chip:

- `MRID` Memory region ID (implicit: determined by the table position)
- `SEID1` Creator SEID: this is the primary Secure Executable corresponding to the memory region (the one that first created it)
- `SEID2` Secondary SEID: this Secure Executable also always has access; allows quick access check of regions shared two-ways.
- Read only flag (boolean)
- Encryption and integrity key
- Integrity root value
- Share Secret: knowledge of this enables read/write access to this region by other Secure Executables (e.g. other than the ones listed as SEID1 or SEID2)
- Memory Region Size

11.7 Other Considerations

11.7.1 Shared Libraries and Dynamic Linking

Our current description requires a statically linked binary, since it is unsafe to allow the host machine to load untrusted code into our memory space at runtime. Statically linking allows all code to be verified at compile time, on the build machine. There are a few downsides to statically linked executables. First, they require extra storage space, since common libraries used by multiple applications cannot be consolidated. Second, the libraries cannot be updated once the Secure Executable has been deployed—in particular, bugs in these libraries cannot be fixed without rebuilding the Secure Executable. This property is necessary for Secure Executables because they do not allow untrusted code to access their memory space. Finally, statically linking with GNU libc in particular is only allowed under the Lesser GPL (LGPL) if certain conditions are met. Specifically, the LGPL requires that customers be able to relink vendor applications to replace the libc version [32]. This poses a challenge since in the Secure Executable model, the customer does not necessarily know the Executable key. One way around this issue is to use a private non-GPL version of libc for static linking.

Another potential solution is to use signed libraries, that are loaded and verified at runtime. A Secure Executable library loader, in protected memory, verifies the integrity of the library as it loads it in protected memory, and provides dynamic linking. That is, a module can be linked at build time with the Secure Executable that will load external libraries of unresolved symbols. These external libraries will only be accepted if there is a certificate chain, with the root signed by a party trusted by the developer, attesting to the trustworthiness of the library code. This mechanism provides integrity guarantees of the shared library, while allowing the library to be patched or replaced in the future. However, it is not as simple as merely dynamically linking to the host libc: the matching certificate must be generated and transmitted to the host.

11.7.2 Virtual Machines

We consider support for two types of virtual machines (VMs) here. By "hardware" virtual machines I refer to VMs that do not require software assistance in executing non-privileged instructions (e.g., "Type 1", or "bare-metal" VMs). Non-privileged instructions in the guest are run directly on hardware. By "software" virtual machines, I refer to VMs that have a software hypervisor to assist in executing instructions (e.g., "Type 2", or most "hosted" VMs).

While the Secure Executable model can be applied to both virtualization scenarios, we consider hardware virtual machines to be the more relevant case for the high-performance server market. In a hardware virtual machine model, a Domain 0 OS typically controls switching between domains. The guest OS remains outside the TCB; Secure Executables operate as in the non-virtualized case. The Domain 0 OS is also outside the TCB, since hypervisor interrupts are treated as regular interrupts with respect to hiding the register contents. We simply require the Domain 0 OS to save and restore the `SEIDSR` register along with the rest of the registers during domain switching. The trick is that the Domain 0 interrupt handler needs to invoke the `RestoreContext` instruction to return to a Secure Executable, if one was running (the SE-ID register is set) during the hypervisor interrupt.

Software virtual machines require a different approach, since the software hypervisor needs access to the sensitive application in order to process its code and data. Thus, we wrap the VM inside the Secure Executable boundary. This is illustrated in Figure 11.8. Note that any VM that emulates instructions in software must be included in the trusted code base, since the instructions and data are considered sensitive. Any exception to allow an untrusted virtual machine to safely handle sensitive instructions and data would require expensive multi-party secure computation approaches, which we will not consider here.

In neither scenario is the host OS part of the TCB. The interesting question is whether, in software VMs, the guest OS must be part of the TCB. I show it does not have to be. Building the software VM as a Secure Executable offers an environment where the VM is protected from the host OS. With support from the software VM, constructing virtual Secure Executables protects guest applications from both the host OS *and* the guest OS. Compare this to Overshadow, which protects guest applications in software VMs from the guest OS but not the host OS. The TCB in our case is just the guest application plus the VM.

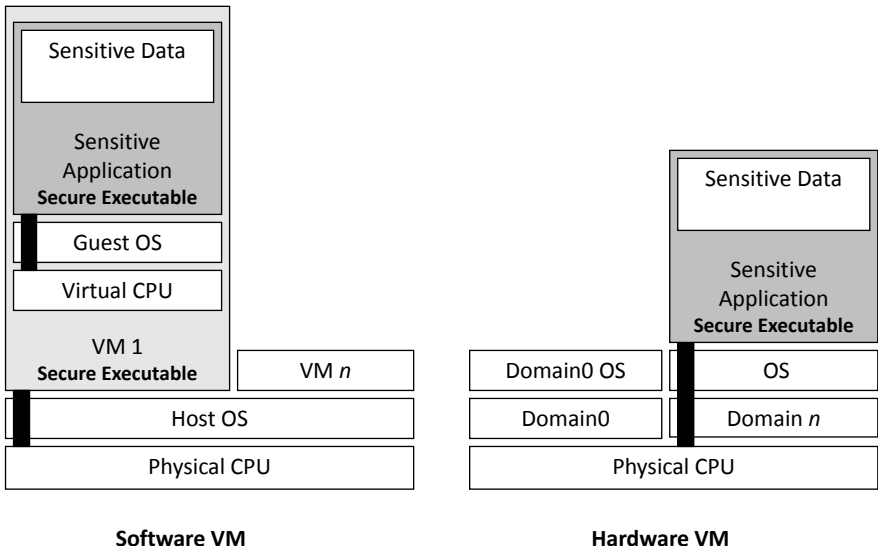

Figure 11.8: Virtual Machine Configurations

The Secure Executable uses an ESM operand encrypted for the virtual machine to decrypt. The virtual machine monitor, in turn, emulates the ESM instruction semantics, performing memory encryption and integrity checking. I assume for the sake of simplicity here that these cryptographic operations are performed in software; however, they can be hardware accelerated by a CPU with suitable cryptographic extensions. In principle, since the VM is already a Secure Executable, and thus protected by hardware, the VM does not need to encrypt the application data at all, except when those pages are accessed by the guest OS. In turn, the VM runs in its own Secure Executable on the physical CPU. A certificate signing the public key of the VM, signed by the VM developers, asserts that the private key is available only to VMs running on Secure Executable-enabled CPUs. This builds the chain of trust, establishing for the application developers that the trusted base is limited to the guest application code, the VM, and the SE-enabled CPU.

To illustrate this point, consider first that the guest OS has been compromised. The encryption and integrity protecting mechanisms built into the VM ensure that the guest OS cannot access sensitive information in the guest Secure Executable. When the host OS has been compromised, the Secure-Executable mechanisms in hardware ensure that the VM cannot be tampered with. The chain of trust is established as follows. A public-private key pair is embedded with the VM, encrypted inside the Secure Executable binary so that only the target SE-CPU can access this VM keypair. The VM developer asserts the privacy of the VM key by signing the public key embedded in the VM Secure Executable. Finally, the developer of the guest Secure Executable encrypts the guest Secure Executable so that only parties knowing the VM private key (in particular, the VM targeted for the SE-CPU) can decrypt the Secure Executable. This isolates the root of trust, guaranteeing that the application developer only need to trust the VM software and SE-CPU (and of course, the SE-CPU factory and VM developer for key management).

11.7.3 Achieving OS Transparency

I believe that OS kernel support for these chip modifications is a reasonable requirement. It is still of interest to make Secure Executable support completely transparent to the OS, however, to allow legacy operating systems to run unmodified. In related work, Overshadow [22] and Flicker [87] provide this.

There are several aspects of this approach to consider with respect to OS transparency. In particular, the interrupt handlers must be able to save and restore the application registers correctly. Moreover, when the interrupt handlers return control to a Secure Executable, the CPU must be able to identify the Secure Executable as such, without help from the OS. We can ensure that no return to an application besides a Secure Executable can accidentally be labeled as a Secure Executable return (which would crash the application).

We will use as an indicator a new instruction, `se-jump`. If on `rfid` the return is to an address containing this instruction, this indicates a return to a Secure Executable. Since these are invalid instructions in the current PowerPC architecture, no correct program would return to this point unless it is a Secure Executable.

In the case of a Secure Executable, when an interrupt occurs, recall that the CPU saves the registers to a restore point in application memory. At this time, the CPU also changes the program counter to point to a `se-jump` instruction in the restore point. When returning from the interrupt, this `se-jump` instruction restores the correct program counter from a register.

Note that we need the functionality of a jump (instead of, e.g., a `se-no-op` instruction) to handle system call returns. This is because PowerPC semantics specify that a system call success is specified by the return point. A system call returns to either the instruction following the entry point, or the following instruction. Providing the `se-jump` instruction allows the application return-from-interrupt code to determine the success value, and return to the appropriate code location.

This approach has an added complication: in PowerPC, Linux relies on the value of the program counter to resolve instruction page faults. By hiding this value from the OS, replacing it with an instruction in the restore point, an unmodified OS is unable to determine which page needs to be brought into memory. We can get around this with legacy operating systems by throwing a data page fault instead of an instruction page fault. In this case, the OS determines the location of the page fault from the data fault register instead of from the program counter.

We also require the OS to indicate to us when a Secure Executable has exited, so the processor can free the attached resources. A potential approach is to maintain a privileged process (distinct from the OS, but with hooks telling it when processes exit) that maps processes to Secure Executable IDs. This privileged process is then in charge of informing the CPU when a Secure Executable exits.

Finally, the overcommit workaround described in Section 11.3.1 requires OS support. This means overcommit must be enabled to achieve OS transparency.

11.7.4 Memory Access Pattern Privacy

Let us consider now the issue of memory access privacy in a Secure Executable. In particular, memory locations are revealed in two ways. First, accessed locations are revealed to the operating system, which is necessary to allow it to service page faults. Second, locations are written to the hardware memory bus, which is necessary for RAM to be able to satisfy read and write requests.

There are existing techniques which can hide the access patterns, such as Oblivious RAM [41]. Implementing Oblivious RAM within a secure executable, adding a layer of indirection over every memory access, would provide a provable guarantee of access privacy. However, notwithstanding the leaps towards practicality identified in Part (I) of this dissertation, existing access-pattern-hiding techniques have a significant overhead, and my goal in this chapter is to provide an inexpensive solution free of any noticeable performance overhead.

It turns out we can modify the hardware design to mitigate, to an extent, both location privacy leaks, with minimal cost overhead. We now take a cursory look at potential approaches. First, the memory regions of a Secure Executable designated as access-privacy-sensitive must be paged in at all times. Effectively, we need to adjust the page size of the protected region (and its integrity tree region), so that either the whole region is paged in or out. Page faults, in particular, must not be delivered individually to the Operating System. This can be effected by clipping the last few bits off of the faulting the virtual address (to correspond to the entire access-private memory region), and modifying the OS to respond appropriately. The whole region, registered with the OS, will be paged in simultaneously, since the does not know where in this region a page fault occurred.

This can be enforced in an architecture supporting multiple page sizes. The CPU should enforce that the page size of an access-private memory region matches the size of the region.

Second, we need to address hardware attacks. I propose extending the boundary of the trusted hardware base from the CPU alone, to include the RAM chips as well. From a hardware perspective, there are several parts to this design change. Bus traffic for such regions must be encrypted. The CPU and RAM negotiate an encryption key on boot.

Since we do not want to require that the CPU and RAM chip be provided by the same vendor, we require software to perform separate authentication of the two pieces of hardware. One option, assuming a sufficiently low-cost asymmetric cryptography hardware implementation, is to use a public/private keypair installed in each RAM chip. A public key certificate installed here is used to negotiate a key with the CPU (preventing man-in-the-middle attacks on the bus), and is provided to software to verify. This attestation is only an assurance of access pattern privacy: confidentiality and integrity is already provided by in-CPU cryptography as discussed earlier.

Only accesses to protected memory regions should be subject to bus encryption. DMA regions, and memory regions not marked access-sensitive are exempted from encryption, to allow the bus controller to work unmodified. The physically protected regions can be denoted using a fixed physical memory range corresponding to protected access.

Software, when registering a memory region as access privacy protected, will be offered the RAM certificate corresponding to the bus encryption key. It is up to software to verify

this certificate and decide whether to continue. Even though this somewhat defeats the transparency goal of the Secure Executable design, it offers in exchange efficient guarantees of physical security of access patterns.

11.8 Analysis

11.8.1 Build Process

System call wrappers are linked in (or, the `sc` handler if using the `sesc` approach described in Section 11.5.2). The Secure Executable also needs unencrypted loader code that initializes the integrity tree and issues the `ESM` instruction. This code is linked in, and the entry point of the ELF is set to point here. This process is illustrated in Figure 11.9.

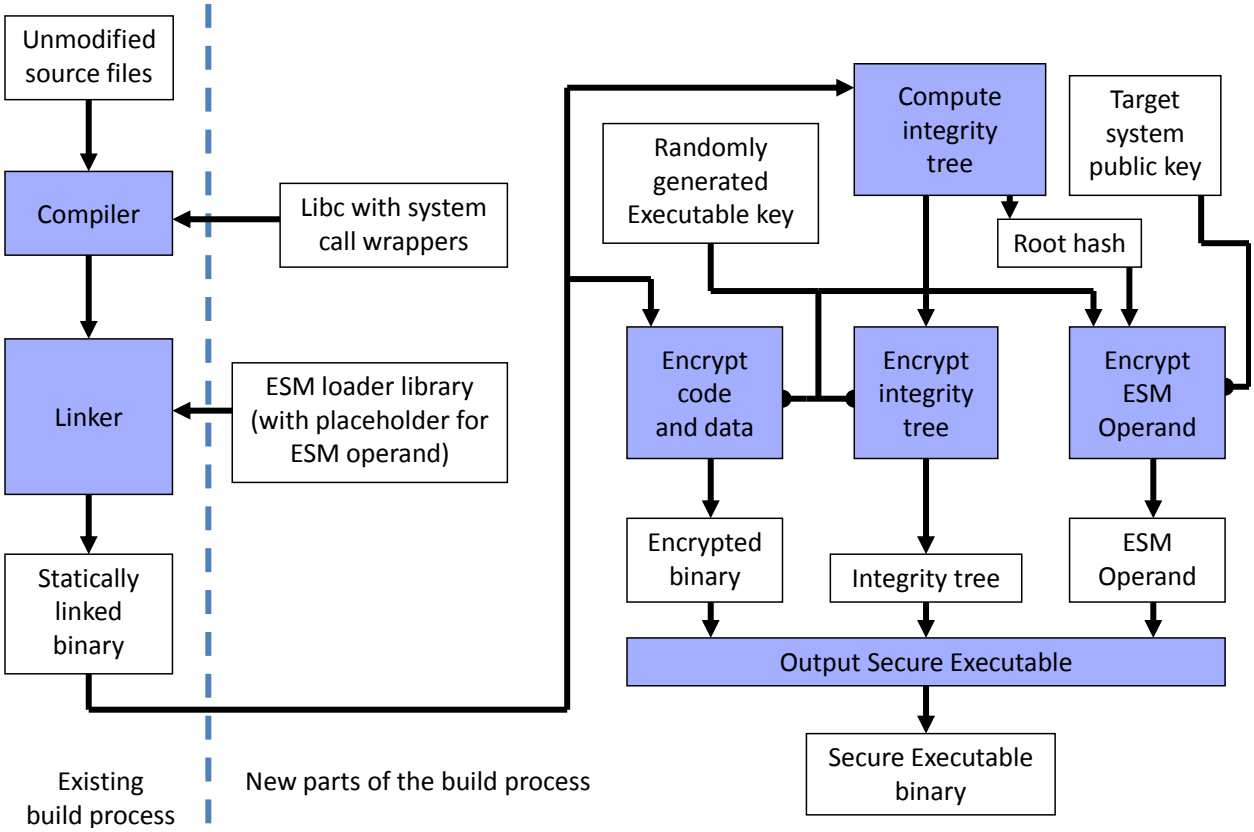

Figure 11.9: Build Process. Operations are represented with a blue background; data is represented with a white background. For encrypt operations, the encryption key applied points the side.

The Executable Key protecting this Secure Executable is randomly generated. This will be used to encrypt the static code and data sections, as well as in the computation of the integrity tree.

Let us now discuss how the ESM integrity value is computed by the build machine and how the initial integrity tree is constructed. In the secure build environment, the Secure Executable binary is statically linked with loader code and libraries. The initial integrity tree is included in the binary; the loader must insert it into the unprotected memory region. As discussed in Section 11.3.1 above, the bulk of the address space assigned to the integrity tree may be unused; therefore we use a compact representation of the tree on disk. This representation is a list of blocks of the tree as contiguous region (position, size, value) tuples. The loader simply copies these regions into the appropriate memory location (adjusting for the tree offset). Overcommit techniques (described above) ensure that only the pages that are used are allocated physical memory pages.

Finally, all sections of the ELF that overlap the protected region are encrypted with the newly generated Executable Key. The ESM operand detailed in Figure 11.7 is constructed by concatenating the various parameters, encrypting with the executable key, then encrypting the executable key with the target system key and attaching.

11.8.2 Performance Estimation

It is beyond the scope of this project to obtain performance results from a full CPU simulation. Future work includes both extending a POWER architecture system simulator to provide cycle-accurate timing results, and expanding the system wrapper library to the point where we can support existing of-the-shelf software. Ideally, measurements will look at the performance in real software.

Instead, we built a simple cache simulation to validate the memory protection model. This simulation shows that it is practical to implement the integrity trees as described in this chapter. Even though verifying a memory location potentially requires verifying several locations, up to the memory root, we see that with normal cache hit rates, the overhead due to the integrity trees is very reasonable.

Figure 11.10 (left) shows the new overhead per load, as a function of the measured cache hit rate. A sequence of 10^6 addresses sampled from an exponential distribution was chosen to simulate a normal application. The hit rate and number of RAM accesses was measured. The cache is first put through a warm-up phase of 10^6 accesses before we begin measuring, to avoid being affected by the cold-start cache misses. This is not ideal, since it fails to capture the cost overhead at the start of program execution, but is necessary for the results to be independent of the simulation length. This limitation can be properly addressed in the future by measuring real software usage patterns across a full simulator.

A fixed cache size (4096 rows with 8 associative columns of 32 words each) is used for all simulations. For a normal application, the RAM accesses measured matches the number of load instructions when the cache hit rate is near zero. As the cache hit rate increases, the number of RAM accesses decreases linearly (following from the definition). The simulation was repeated, with the same address distribution parameter, for a Secure Executable. We then vary the exponential parameter to obtain different cache hit rates. The Secure Executable cache hit rate is lower than the normal application hit rate since cache lines are occupied with integrity tree values in the Secure Executable case. To capture the

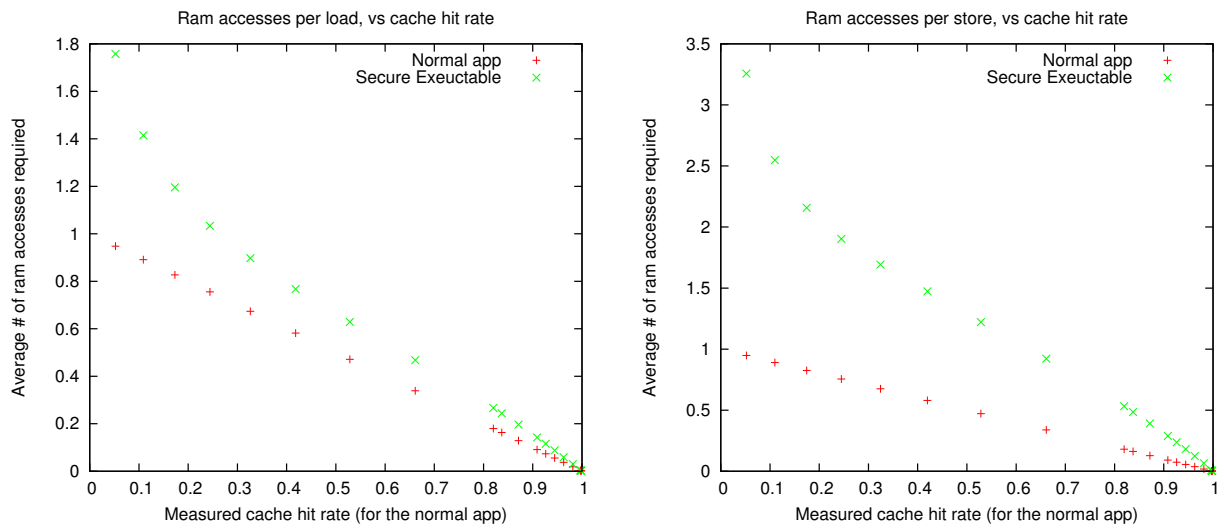

Figure 11.10: Left: RAM accesses per load. Right: RAM accesses per store

performance penalty of this aspect, the values are plotted against the normal application hit rate corresponding to this distribution.

Figure 11.10 (right) repeats this simulation, measuring stores instead of loads. While the cache logic is significantly different in the Secure Executable case, we see the measured performance exhibits similar behavior to the loads test. Note that in both figures, the integrity tree lookups dominate at low cache hit rates. Even so, the bulk of the integrity tree levels remain in cache, such that only a few locations at the lower level need to be brought in. For cache hits, the integrity tree does not even need to be consulted. This is reflected in the better performance at high cache hit rates.

These figures do not address cryptographic performance. Since the number of cryptographic operations is equivalent to the number of RAM accesses, these graphs can be trivially adjusted to reflect this performance penalty by scaling the Y axis to the ratio of the latency of a hardware cryptographic verify and decrypt operation plus RAM access vs. a RAM access alone.

11.9 Conclusion

11.9.1 Limitations of this Approach

First, we use a separate binary for every target machine. This does not require re-compilation, merely re-encryption. If we consider a Distribution Server to be the party that encrypts the code for each target CPU, the developer must trust the Distribution Server with the code and static data. This is necessary since this server decides who can read the code (and attached data). We can thus assume this Distribution Server is part of the trusted build environment; however, this may not always be a convenient assumption.

A separate useful approach to this issue is to install a single Secure Executable on the target CPU, which we will call a Deployment Server. This Deployment Server will then decrypt future binaries that are distributed under a single key, and re-encrypt them for the target CPU. This allows a kind of boot-strapping of the application distribution: once a single trusted entity exists on the target system, it can be use to safely re-target other executables for that CPU.

Next, my description is for a RISC-like architecture; this may be stretching the definition of RISC since we require multiple memory accesses in a single instruction. However, existing `POWER` instructions already have this requirement. Moreover, we still fit in the definition of a "load/store" architecture (under which each instruction can do only one or the other), since the `ESM` instruction is implemented as a "load".

Finally, it can be argued that we are minimizing the software trusted code base and optimizing application transparency at the expense of increasing hardware complexity, and thus the size of the hardware trust base. Acknowledging that we are increasing hardware complexity, I make the case that the sum of changes required is still reasonable. We shall require only a specific set of changes, which each have only a limited scope.

11.9.2 Final Notes

This chapter outlines the comprehensive set of hardware changes that together enforce the software isolation guarantees the Operating System is expected to provide, as well as protecting from physical attacks, malicious Operating Systems, and malware. This in turn minimizes the size of the software trusted code base. Notably, we can achieve this in a manner mostly transparent to the application developers, and compatible with existing software. By providing a convenient way to achieve fundamental guarantees about the security of a software execution environment, the adoption of such an architecture will address some of the most pressing security issues today.

Chapter 12

Conclusions

This dissertation described how to overcome a broad set of challenges related to access privacy. We first looked at tackling the immense shuffle cost in the original Goldreich and Ostrovsky ORAM [41] by assuming some local storage. A set of probabilistic techniques—relying on the uniformity of the randomness in generating these permutations—assured us that a small amount of local storage suffices to retain privacy in Chapter 3. These techniques were applied in Chapter 4 to both to speed up both an oblivious sort, and the process of obliviously managing a large number of remotely stored fake items. Further analysis of these techniques resulted in an asymptotically faster oblivious scramble. This was employed in Chapter 5, with a new data structure that speeds up both the querying and level construction. We made the observation that the large number of fake items in existing ORAMs are used only to hide the success of a lookup at each level. This resulted in separation of the membership from the lookup and a faster ORAM, while restoring correctness guarantees against an actively malicious adversary.

Having successfully tackled the level construction process, we moved on to other barriers to practicality. Chapter 6 showed how to bring the worst case query cost in line with the average case, generalizing an approach used in existing work, and providing an implementation of these techniques. I showed how to apply the solution to high-latency networks, despite the high degree of query interactivity. This parallel approach allowed multiple clients to query simultaneously, while privately resolving any query conflicts between clients. Measurements of a real implementation showed the results to be practical and useful, even over large databases. A first oblivious file system then improved usability. Chapter 7 provided a new different approach to the issue of query latency, compacting the whole interactive query process into a single round trip, while retaining correctness guarantees.

Part (II) showed how to extend privacy to related scenarios: efficient transaction privacy in a multi-client scenario was showed possible in Chapter 9. Chapter 11 examined the feasibility of hardware approaches, as well as the relevance of ORAM solutions for hardware.

The constructions of Part (I) proved that access privacy can be practical, despite the large number of involved challenges. In combination with the extensions in Part (II), I motivated, designed, and analyzed solutions that achieve access privacy with unprecedented speeds.

Bibliography

[1] Atul Adya and Barbara Liskov. Generalized isolation level definitions. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 67–78, 2000. 124

[2] M. Ajtai, J. Komlos, and E. Szemeredi. An O(n log n) sorting network. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 1–9, 1983. 16

[3] American National Standard for Information Systems, 1819 L Street, NW, Washington, DC 20036, USA. *ANSI X3.135-1992 – Database Language SQL*, 1992. 124

[4] Yair Amir and Ciprian Tutu. From total order to database replication. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2000. 121

[5] D. Asonov. *Querying Databases Privately: A New Approach to Private Information Retrieval*. Springer Verlag, 2004. 98, 114

[6] K.E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, page 307314, 1968. 16

[7] Paul Beame and Widad Machmouchi. Making RAMs oblivious requires superlogarithmic overhead. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:104, 2010. 10

[8] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proceedings of EuroCrypt*, 1997. 65

[9] Steven M. Bellovin and William R. Cheswick. Privacy-enhanced searches using encrypted bloom filters. Technical report, Columbia University, 2004. 54

[10] Rhys Blakely, Jonathan Richards, James Rossiter, and Richard Beeston. MI5 alert on china's cyberspace spy threat. *The Times (of London)*, December 3, 2007. 145

[11] M. Blaze. A Cryptographic File System for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, VA, 1993. ACM. 121

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. 53, 54, 122

[13] Philip Bohannon, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, and S. Sudarshan. Using Codewords to Protect Database Data from a Class of Software Errors. In *Proceedings of ICDE*, 1999. 120

[14] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt 2004*, pages 506–522. LNCS 3027, 2004. 122

[15] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EuroCrypt*, 2003. 119

[16] Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E. Skeith, III. Public key encryption that allows PIR queries. In *Proceedings of the 27th annual international cryptology conference on Advances in cryptology*, CRYPTO'07, pages 50–67, Berlin, Heidelberg, 2007. Springer-Verlag. 113

[17] Dan Boneh, David Maziéres, and Raluca Ada Popa. Remote oblivious storage: Making Oblivious RAM practical. Technical report, MIT, 2011. MIT-CSAIL-TR-2011-018 March 30, 2011. 5, 6, 7, 8, 12, 13, 98

[18] R. Brinkman, J. Doumen, and W. Jonker. Using secret sharing for searching in encrypted data. In *Secure Data Management*, 2004. 122

[19] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 245–252, Boston, MA, June 2001. 121

[20] CBS. Feds Seek Google Records On Porn. Online at `http://www.cbsnews.com/2100-205_162-1221309.html`, January 2006. 117

[21] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. Cryptology ePrint Archive, Report 2004/051, 2004. `http://eprint.iacr.org/`. 122

[22] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008. 149, 164, 171

[23] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 41–50, 1995. 113

[24] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998. 115

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001. 36

[26] IBM Corporation. 117

[27] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles, 2010. ivan@cs.au.dk, stm@cs.au.dk, jbn@cs.au.dk 14670 received 27 Feb 2010, last revised 1 Mar 2010. 10

[28] Premkumar T. Devanbu, Michael Gertz, April Kwong, Chip Martel, G. Nuckolls, and Stuart G. Stubblebine. Flexible authentication of XML documents. In *ACM Conference on Computer and Communications Security*, pages 136–145, 2001. 120

[29] Premkumar T. Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *IFIP Workshop on Database Security*, pages 101–112, 2000. 118, 119

[30] Jeffrey S. Dwoskin and Ruby B. Lee. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 389–400, New York, NY, USA, 2007. ACM. 148

[31] William Feller. *An Introduction to Probability Theory and its Applications*, volume 1. Wiley, 1967. 21, 22

[32] Free Software Foundation. GNU lesser general public license version 3. `http://www.gnu.org/licenses/lgpl.html`, June 2007. 168

[33] Martin Franz, Peter Williams, Bogdan Carbunar, Stefan Katzenbeisser, Andreas Peter, Radu Sion, and Miroslava Sotáková. Oblivious outsourced storage with delegation. In *Financial Cryptography*, pages 127–140, 2011. 141

[34] W. Gasarch. A WebPage on Private Information Retrieval. Online at `http://www.cs.umd.edu/~gasarch/pir/pir.html`, 2010. 113

[35] William Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82:72–107, 2004. 113

[36] Tingjian Ge and Stan Zdonik. Answering aggregation queries in a secure system model. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 519–530. VLDB Endowment, 2007. 119

[37] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google File System. pages 29–43. 121

[38] E. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/. 122

[39] Ian Goldberg. Improving the Robustness of Private Information Retrieval. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, May 2007. 113

[40] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2001. 4, 66, 124

[41] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on Oblivious RAM. *Journal of the ACM*, 43, Issue 3:431–473, May 1996. 3, 10, 16, 36, 43, 52, 73, 75, 81, 83, 107, 172, 177

[42] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Proceedings of ACNS*, pages 31–45. Springer-Verlag; Lecture Notes in Computer Science 3089, 2004. 122

[43] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In *Proceedings of the 2001 Conference on Topics in Cryptology*, pages 425–440. Springer-Verlag, 2001. 120

[44] Michael Goodrich and Michael Mitzenmacher. MapReduce parallel cuckoo hashing and Oblivious RAM simulations. In *38th International Colloquium on Automata, Languages and Programming (ICALP)*, 2011. 13, 15, 81, 107

[45] Michael Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *ACM Cloud Computing Security Workshop at CCS (CCSW)*, 2011. 13, 86, 96

[46] Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *In Proceedings 21st ACM-SIAM Symposium on Discrete Algorithms (SODA*, 2010. 15, 16, 17, 60, 69, 85, 97, 116

[47] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via Oblivious RAM simulation. In *ICALP (2)*, pages 576–587, 2011. 17

[48] Google.com. Google App Engine FAQs. Online at https://developers.google.com/appengine/kb/. 117

[49] P. C. Gutmann. Secure filesystem (SFS) for DOS/Windows. www.cs.auckland.ac.nz/~pgut001/sfs/index.html, 1994. 121

[50] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 216–227. ACM Press, 2002. 119

[51] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *IEEE International Conference on Data Engineering (ICDE)*, 2002. 118

[52] Torben Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Inf. Process. Lett.*, 33(6):305–308, 1990. 31

[53] Jerry Harkavy. Illicit software blamed for massive data breach: Unauthorized computer programs, secretly installed on servers in Hannaford Brothers supermarkets compromised up to 4.2 million debit and credit cards. *AP*, March 28, 2008. 145

[54] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994. 121

[55] Stephen Hemminger. Network emulation with netem (lca 2005). Online at `http://developer.osdl.org/shemminger/netem/LCA2005_paper.pdf`, April 2005. 138

[56] Csaba Henk and Miklos Szeredi. FUSE: Filesystem in Userspace. Online at `http://sourceforge.net/projects/fuse`, 2012. 92

[57] Mathias Hild and Jordan Mitchell. Free Email: Google, MSN Hotmail and Yahoo! (A). *SSRN eLibrary*, 2004. 2

[58] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proceedings of ACM SIGMOD*, 2004. 119

[59] IBM. IBM 4764 PCI-X Cryptographic Coprocessor (PCIXCC). Online at `http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`, 2006. 98, 114, 151

[60] IBM. IBM secure blue processor architecture. Online at `http://www-03.ibm.com/press/us/en/pressrelease/19527.wss`, year. 146

[61] A. Iliev and S.W. Smith. Private information storage with logarithmic-space secure hardware. In *Proceedings of i-NetSec 04: 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*, pages 201–216, 2004. 114

[62] Inetu.net. Inetu.net Managed Database Hosting. Online at `http://www.inetu.net`. 117

[63] Yuval Ishai and Eyal Kushilevitz. Private simultaneous messages protocols with applications. *Theory of Computing Systems, Israel Symposium on the*, 0:174, 1997. 116

[64] Jetico, Inc. BestCrypt software home page. `www.jetico.com`, 2002. 121

[65] Aaron Johnson, Paul Syverson, Roger Dingledine, and Nick Mathewson. Trust-based anonymous communication: Adversary models and routing algorithms. In *ACM Computer and Communications Security Conference (CCS 2011)*, 2011. 116

[66] Bobbie Johnson. Viacom lawsuit: Google told to hand over all YouTube user details. The Guardian. Online at `http://www.guardian.co.uk/technology/2008/jul/04/youtube.google`, July 2008. 117

[67] A. Kashyap, S. Patil, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004. USENIX Association. 121

[68] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, pages 134–143, 2000. 121

[69] G. Kim and E. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. In *Proceedings of the Usenix System Administration, Networking and Security (SANS III)*, 1994. 121

[70] G. Kim and E. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer Commuications and Society (CCS)*, November 1994. 121

[71] Jan Kneschke, Lenz Grimmer, Martin Brown, Giuseppe Maxia, and Kay Röpke. Mysql proxy - mysql forge wiki. Online at `http://forge.mysql.com/wiki/MySQL_Proxy`, 2008. 138, 139

[72] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based Oblivious RAM and a new balancing scheme. Cryptology ePrint Archive, Report 2011/327, 2011. 5, 13, 63

[73] De-Lei Lee and Kenneth E. Batcher. A multiway merge sorting network. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):211–215, February 1995. 17

[74] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 2–13, Washington, DC, USA, 2005. IEEE Computer Society. 148

[75] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association. 71, 73

[76] David J. Lie. *Architectural support for copy and tamper-resistant software*. PhD thesis, Stanford, CA, USA, 2004. Adviser-Horowitz, Mark. 147, 148, 158

[77] Helger Lipmaa. AES ciphers: speed. Online at `http://www.cs.ut.ee/~lipmaa/research/aes/rijndael.html`, 2006. 48, 67

[78] Helger Lipmaa and Bingsheng Zhang. Two new efficient PIR-writing protocols. In *Proceedings of the 8th international conference on Applied cryptography and network security*, ACNS'10, pages 438–455, Berlin, Heidelberg, 2010. Springer-Verlag. 113

[79] Amazon Web Services LLC. Amazon EC2 FAQs. Online at `http://aws.amazon.com/ec2/faqs`. 117

[80] Amazon Web Services LLC. Amazon Simple Storage Service FAQs. Online at `http://aws.amazon.com/s3/faqs/`. 117

[81] Amazon Web Services LLC. Amazon Web Services FAQs. Online at `http://aws.amazon.com/faqs`. 117

[82] John Markoff. Russian gang hijacking PCs in vast scheme. *NY Times*, August 6, 2008. 145

[83] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authenticated data structures. Technical report, 2001. 119

[84] 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. 119

[85] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *PODC '02: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 108–117, New York, NY, USA, 2002. ACM Press. 123

[86] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010. 149

[87] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems(EUROSYS)*, 2008. 149, 171

[88] A. D. McDonald and M. G. Kuhn. StegFS: A Steganographic File System for Linux. In *Information Hiding*, pages 462–477, 1999. 121

[89] R. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Research in Security and Privacy*, 1980. 119

[90] Microsoft Research. Encrypting File System for Windows 2000. Technical report, Microsoft Corporation, July 1999. `www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp`. 121

[91] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. 53

[92] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *ISOC Symposium on Network and Distributed Systems Security NDSS*, 2004. 118, 119, 121

[93] E. Mykletun, M. Narasimha, and G. Tsudik. Signature bouquets: Immutability for aggregated/condensed signatures. In *Computer Security - ESORICS 2004*, volume 3193 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2004. 121

[94] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Signature Bouquets: Immutability for Aggregated/Condensed Signatures. In *Proceedings of the European Symposium on Research in Computer Security ESORICS*, pages 160–176, 2004. 119

[95] MySQL. MySQL 5.0 Reference Manual. `http://dev.mysql.com/doc/refman/5.0/en/`. 137

[96] Moni Naor and Benny Pinkas. Oblivious transfer with adaptive queries. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 573–590, London, UK, 1999. Springer-Verlag. 116

[97] Maithili Narasimha and Gene Tsudik. DSAC: integrity for outsourced databases with signature aggregation and chaining. Technical report, 2005. 119

[98] Maithili Narasimha and Gene Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *Proceedings of DASFAA*, 2006. 120

[99] Femi Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In *In Financial Cryptography and Data Security 11*, 2011. 77, 115

[100] Opendb.com. Opendb.com Web Database Hosting. Online at `http://www.opendb.com`. 117

[101] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *IN PROCEEDINGS OF STOC*, pages 294–303. ACM Press, 1997. 13

[102] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51:122–144, May 2004. 13

[103] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of EuroCrypt*, 1999. 119

[104] Pascal Paillier. A trapdoor permutation equivalent to factoring. In *PKC '99: Proceedings of the Second International Workshop on Practice and Theory in Public Key Cryptography*, pages 219–222, London, UK, 1999. Springer-Verlag. 119

[105] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying Completeness of Relational Query Results in Data Publishing. In *Proceedings of ACM SIGMOD*, 2005. 120

[106] HweeHwa Pang and Kian-Lee Tan. Authenticating query results in edge computing. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 560, Washington, DC, USA, 2004. IEEE Computer Society. 120

[107] Kyriacos Pavlou and Richard T. Snodgrass. Forensic Analysis of Database Tampering. In *Proceedings of ACM SIGMOD*, 2006. 120

[108] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519, 2010. 5, 13, 15, 63, 81, 87

[109] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. pages 89–101. 121

[110] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. `http://eprint.iacr.org/`. 116

[111] Len Sassaman, Bram Cohen, and Nick Mathewson. The Pynchon gate: a secure method of pseudonymous mail retrieval. In *WPES*, pages 1–9, 2005. 113

[112] Tom Scholl. ext2fuse: ext2 filesystem in userspace. Online at `http://sourceforge.net/projects/ext2fuse/`, 2009. 92

[113] Curtis Scribner. Comment and casenote: Subpoena to Google Inc. in ACLU v. Gonzales: "Big Brother" is watching your internet searches through government subpoenas. *University of Cincinnati Law Review*, 75(1273), 2007. 2

[114] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In *ASIACRYPT*, pages 197–214, 2011. 13

[115] Radu Sion. Query execution assurance for outsourced databases. In *Proceedings of the Very Large Databases Conference VLDB*, 2005. 120

[116] Radu Sion and Bogdan Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*, 2007. 77, 115

[117] G. Sivathanu, C. P. Wright, and E. Zadok. Enhancing File System Integrity Through Checksums. Technical Report FSL-04-04, Computer Science Department, Stony Brook University, May 2004.
`www.fsl.cs.sunysb.edu/docs/nc-checksum-tr/nc-checksum.pdf`. 121

[118] Richard T. Snodgrass, Stanley Yao, and Christian Collberg. Tamper detection in audit logs. In *Proceedings of VLDB*, 2004. 120

[119] D. Xiaodong Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000)*. IEEE Computer Society, 2000. 121

[120] Emil Stefanov, Elaine Shi, and Dawn Song. Towards Practical Oblivious RAM. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012. 14, 98

[121] Jonathan Stempel. Bank of NY Mellon data breach now affects 12.5 million. *Reuters*, August 28, 2008. 145

[122] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. pages 160–171. ACM Press, 2003. 148, 158

[123] G. Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 9–14, New York, NY, USA, 2007. ACM. 155

[124] M. Sullivan and M. Stonebraker. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proceedings of VLDB*, 1991. 120

[125] Shuhong Wang, Xuhua Ding, Robert H. Deng, and Feng Bao. Private information retrieval using trusted hardware. In *Proceedings of the European Symposium on Research in Computer Security ESORICS*, pages 49–64, 2006. 12, 14, 98, 115

[126] Peter Williams and Rick Boivie. CPU Support for Secure Executables. In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST)*, 2011. 3

[127] Peter Williams and Radu Sion. Usable Private Information Retrieval. In *Proceedings of the 2008 Network and Distributed System Security (NDSS) Symposium*, 2008. 3, 13

[128] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*, pages 139–148, 2008. 3, 13

[129] Peter Williams, Radu Sion, and Dennis Shasha. The Blind Stone Tablet: Outsourcing Durability. In *Proceedings of the 2009 Network and Distributed System Security (NDSS) Symposium*, 2009. 3

[130] Peter Williams, Radu Sion, and Miroslava Sotakova. Practical oblivious outsourced storage. *ACM Trans. Inf. Syst. Secur.*, 14:20:1–20:28, September 2011. 3

[131] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. pages 197–210. 121