Oblivious RAM: From Theory to Practice

by

Christopher W. Fletcher

B.S. in Electrical Engineering and Computer Science, University of California, Berkeley (2010)
S.M. in Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Oblivious RAM: From Theory to Practice

by

Christopher W. Fletcher

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Privacy of data storage has long been a central problem in computer security, having direct implications for many Internet-era applications such as storage/computation outsourcing and the Internet of Things (IoT). Yet, the prevailing way we protect our data — through encryption techniques — doesn't protect *where* we read or write in our data. This additional information, the access pattern, can be used to reverse-engineer proprietary programs as they run, reveal a user's physical location or health information, and more, *even if data is correctly encrypted*.

This thesis studies a cryptographic primitive called Oblivious RAM (ORAM) which provably hides a client's access pattern as seen by untrusted storage. While ORAM is very compelling from a privacy standpoint, it incurs a large performance overhead and can require a large amount of client (trusted) storage. In particular, ORAM schemes require the client to continuously shuffle the data stored in the untrusted storage, using the trusted storage. Early work on ORAM proves that this operation must incur a client-storage bandwidth blowup that is logarithmic in the dataset size, which can translate to $> 100\times$ in practice.

We address this challenge by developing new tools for constructing ORAMs that allow us to achieve *constant bandwidth blowup* while requiring only *small client storage*. A re-occurring theme is to grant untrusted storage the ability to perform *untrusted* computation on behalf of the client, thereby circumventing lower bound results from prior work. Using these tools, we construct a new ORAM called Ring ORAM, the first small client storage ORAM to achieve constant online bandwidth blowup. At the same time, Ring ORAM matches or improves upon the overall bandwidth of all prior ORAM schemes (given equal client storage), up to constant factors. Next, we more heavily exploit computation at the storage to construct Onion ORAM, the first scheme with constant worst-case and overall bandwidth blowup that does not require heavy weight cryptographic primitives such as fully homomorphic encryption (FHE). Instead, Onion ORAM relies on more efficient additively or somewhat homomorphic encryption schemes.

Finally, we demonstrate a working ORAM prototype, built into hardware and taped-out in 32 nm silicon. We have deployed the design as the on-chip memory controller for a 25 core processor. This proves the viability of a single-chip secure processor that can prevent software IP or data theft through a program's access pattern to main memory (having applications to computation outsourcing and IoT). From a technical perspective, this work represents the first ORAM client built into silicon and the first hardware ORAM with small client storage, integrity verification, or encryption units. We propose a number of additional optimizations to improve performance in the hardware setting.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I would first like to thank my advisor Srini Devadas, Marten van Dijk (my de facto co-advisor), and Ling Ren (my closest collaborator).

First, to Srini. Srini is that rare individual who is not only strong technically, but also incredibly optimistic and kind (finally, high energy!). Srini is well known for his technical breadth. Case in point, he bootstrapped the security project by recognizing the security application in one of my early computer architecture projects. I had no background in security, and this was the biggest turning point in my graduate career. I have always been impressed by the quality of Srini's ideas. For example, one of Srini's early ideas in the security project (back from 2012) survived three years to become the foundation for the Onion ORAM scheme in this thesis. Finally, I have always appreciated how much freedom Srini has given me to pursue whatever I was interested in. He let us pursue ORAM to its conclusion and this was the reason for any of our success. Srini, for allowing us the freedom to try things that would never work, I thank you. On a personal note, I consider Srini to be a good friend.

Next, to Marten. To those who don't know Marten, I recommend having him brainstorm with you for five minutes on a problem you have been stuck on. What will happen is that, in the first five minutes, he will find improvements to your scheme. Next, over the course of the following five hours, he will develop a completely new and more elegant approach. Some of the coolest ideas in the security project come from these discussions. Marten, thank you so much for your mentorship; I feel very fortunate to be able to work with you.

Finally, to my collaborator Ling. Ling has been my closest collaborator all throughout the ORAM work (we even co-founded a company together). Working with Ling is like having a wizard equally strong in systems and theory available for long brainstorming sessions every day, all day. Ling, it's been a real privilege working with you, and I know we will collaborate again in the future.

Next, I would like to acknowledge my collaborators. The work presented in this thesis is shared with an amazing group of co-authors, namely: Ling Ren, Albert Kwon, Xiangyao Yu, Marten van Dijk, Srinivas Devadas, Emil Stefanov, Elaine Shi, Daniel Wichs and Dimitrios Serpanos. Also, I thank Omer Khan who I collaborated with on multiple projects over the years. Special thanks to Emil Stefanov and Elaine Shi, whose work provided the foundation for this thesis. Finally, I would like to thank the Princeton team, who made the chip tape-out possible: Dave Wentzlaff, Mike McKeown, Tri Nguyen, Jonathan Balkind, Alexey Lavrov, Yaosheng Fu and others.

Next, I would like to thank my colleagues at MIT CSAIL, and especially to members of the Hornet group. First, I thank Nickolai Zeldovich, Daniel Sanchez and Vinod Vaikuntanathan for sitting on my various qualifying committees. Special thanks to Vinod, for the discussions on Onion ORAM and Homomorphic Encryption all the way back to the beginning of that project. To my colleagues in the Hornet group and Floor 8: Jean Yang,

Charles Herder, Alin Tomescu, Ilia Lebedev, Victor Costan, Rachael Harding, Keun sup Shim, Charles O'Donnell, Michel Kinsy, Mieszko Lis, Nirav Dave, Sang Woo and others. To everyone in the Hornet group, for putting up with security talks and offering feedback from the perspectives of no less than three areas.

I feel very privileged to have been mentored by some amazing people outside of CSAIL. I would like to thank Joe Pasquale, Charles Leiserson, John Wawrzynek, Garry Nolan and Greg Gibeling. In particular, Joe Pasquale has been one of my closest mentors ever since high school: taking my endless calls to discuss all angles of academics and life, taking me on trips to Italy for the summer, the list goes on. Joe, the number of insights you have given me all throughout my academic career is staggering. Thank you for your mentorship and, above all, your friendship. I would also like to thank Charles for being my academic advisor and for giving me the opportunity to T.A. 6.172 (Performance Engineering). Charles is without doubt *the best teacher* I have ever seen. Charles, it was a real treat to watch you run 6.172, and I will work to uphold your standard in my future courses and mentorship.

Finally, I would like to thank my girlfriend Cassie, my brother and my parents. Cassie, you and I have been long-distance for the past five years yet it's felt like you have been at my side the whole time. You are the most amazing girlfriend, and my closest confidant. To our outings in Chicago, cheers! To my brother, Sam: you are my best friend, and the only person who understands (in particular) that the meteor tattoo is not a football. Once I turn in this thesis, let's go to Vegas. To Mom and Dad: words don't do you guys justice. You have always loved and supported me unconditionally, and nothing else is possible without that. Thank you for being my strongest advocates, and best friends.

For my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Security of data storage is a huge problem in nearly all aspects of the Internet connected world. Consider several ubiquitous settings: outsourced storage, computation outsourcing and the Internet of Things (IoT).

In outsourced storage, users outsource private data storage from their private infrastructure to remote cloud servers. Data can now be stolen at any point in the cloud infrastructure; for instance, at the server itself (e.g., by insiders [38]), at the internet-server boundary (based on the Snowden revelations [67]) or in transit (e.g., [82]).

Further, in computation outsourcing and IoT, sensitive information is stored on cloud servers, or other potentially hostile environments, as it is being computed upon. Despite the promise of tamper-resistant systems (e.g., [43]) and bootstrapping trust from a known CPU state (e.g., [97]), which protect data *while* it resides on-chip (or on-package), data can still be stolen via software or physical attacks when it is stored *off-chip* (e.g., in main memory or disk). For instance, it has been shown how memory can be accessed by exploiting cloud resource sharing [36] and vulnerable firmware [94, 2]. In IoT, the attacker may have physical access to devices, which it can use to extract data using (for example) test cards [23], bus probing [16] or technology-specific techniques [35].

A natural starting point to address this issue is to encrypt all data written to untrusted storage. For example, consider client-side encryption which defines two parties: a trusted *client* and untrusted *server* (storage). When data passes to/from the server, it is encrypted/decrypted by the client. Only the client holds the secret key. Thus, the server cannot decrypt the data it stores unless it is able to break the encryption scheme. Client-side encryption is used today. For example, it is implemented at the chip boundary in remote processors to protect main memory (e.g., Intel SGX [62]) and at the client boundary to protect outsourced storage applications (e.g., [30]).

Program 1 An example program that runs on a remote processor and leaks the integer a through the program address pattern.

```
1: function LEAKY(integer a, array M)
2:     return M[a]
3: end function
```

A big problem with client-side encryption (and other systems that protect only the data itself) is that it does not protect all aspects of how the client interacts with the server's storage. *Where* storage is accessed, the *access pattern*, can also reveal secret information. For a distilled example, consider the program LEAKY (Program 1). This program may run

on a remote 'secure' processor where the access pattern to memory is visible to an attacker. In this case, even if the inputs a and M are loaded into memory encrypted, a must be decrypted during program execution to complete the memory access $\mathsf{M}[a]$. Depending on where the encrypted array M is stored (e.g., on a remote disk/server, main memory, etc), a may be revealed to a variety of system components (e.g., the Internet, the network to disk, the processor memory bus, etc). This example exhibits the root cause of the problem, which applies to a variety of realistic settings. For example:

- Suppose a patient stores his/her genome on a remote server and wishes to check if he/she has an allele/SNP (i.e., which is located at a specific point on the genome) which corresponds to cancer. If an observer (e.g., an insurance company) learns *where* that patient is looking in its genome, the observer can infer that the patient was concerned about cancer. Similar examples can be drawn from users requesting geo-location, financial and database queries over other sensitive information (e.g., [53, 78]).

- A common task in personal and cloud computing is to run a proprietary program on a remote processor. One of the open challenges with this deployment is to prevent *software IP theft*: the program distributor wants to avoid malicious parties from being able to reverse-engineer the program as it runs. Unfortunately, an observer capable of monitoring how a program accesses main memory can, in fact, reverse engineer the program's conditional and loop structure, simply by monitoring address requests to main memory [27, 33, 78].

- In the inverse of the software IP theft setting, a user may wish to outsource *private data* to a remote processor to compute some result. In this case, the program may be selected by the server hosting the processor (e.g., a cloud service which cannot be attested by the user) and is therefore untrustworthy [48, 49]. Untrusted programs running on sensitive data are a serious concern: the program may directly or inadvertently leak the user's data. For example, the Leaky program (Program 1) may perform a legitimate task, but none-the-less leaks privacy.

1.1 Challenges In Protecting Access Pattern

The underlying problem in the above examples is inherent in how we write programs today: *to be performant, program control flow and and memory access behavior depends on the sensitive information we wish to hide.* Indeed, a strawman solution to eliminate all access pattern leakage is to perform the same amount of work, regardless of the program's sensitive inputs. In the worst case, this requires that the program *scan* all of memory on every access – e.g., download the entire genome to analyze a single allele – incurring huge performance overheads.

A natural question to ask is: can encryption solve this problem? Generally, the answer is no, considering practical constraints. Encrypting an address makes that address unusable by the memory unless the remote memory has the corresponding decryption key or the system is using certain cryptographic schemes. First, distributing decryption keys has serious limitations. In particular, the trust boundary now includes all of remote storage and the burden is on memory manufacturers to re-design their products to (safely) perform key exchange. Second, certain encryption schemes (e.g., private information retrieval [14] or homomorphic encryption [6]) can securely search over encrypted data. However, these schemes

have an inherent problem: the scheme must compute over every element in the database. Otherwise, an observer trivially knows what elements were not selected. This has even worse overheads than scanning memory due to these schemes' computational complexities. To summarize, we desire address pattern protection that doesn't make assumptions on the untrusted storage, and is asymptotically more efficient than scanning memory.

Another question is: can we get away with incomplete protection? Incomplete access pattern protection is implicit in the state of the art hardware extensions from Intel, called Intel SGX [62, 94]. In that system, the access pattern may be called 'partially hidden' because the subset of memory accesses that cause page faults are directly revealed to the untrusted operating system. Recently, however, researchers showed how even this amount of leakage can be used to reconstruct the outline of medical images in medical applications [92]. Another example in this vein is the HIDE framework, by Zhuang et al. [27]. HIDE provides access pattern protection assuming constraints on the spatial locality in the program access pattern.[1] But HIDE makes no guarantees for programs with arbitrary access patterns and, in particular, leaks non-negligible information for even a single access if there are no restrictions on where that access may occur (such as in the examples from Section 1). To summarize, we desire a general solution that doesn't make assumptions about the program access pattern, or about how much privacy is leaked on a particular memory access.

Another way to see the danger in the above attacks is to look at society's move from deterministic to randomized encryption schemes. In the University and Industry, we teach our students not to use deterministic encryption because it is "insecure," in particular it is subject to *frequency analysis attacks*. The access pattern can be viewed in a similar light: as client-side encryption becomes ubiquitous, frequency attacks on the remaining un-encrypted information (the access pattern) can become the new low-hanging fruit.

1.2 The Case for Oblivious RAM

To address the above problems, this thesis studies a cryptographic primitive called Oblivious RAM (ORAM), which *provably* eliminates all information leakage in memory access patterns [9, 13].

As with client-side encryption, an ORAM scheme is made up of a client and server with data blocks residing on the server. Consider two sequences of storage requests A and A' made to the server, where each sequence is made up of read $(\mathsf{read}, addr)$ and write $(\mathsf{write}, addr, data)$ tuples. ORAM guarantees that from the server's perspective: if $|A| = |A'|$, then A is computationally indistinguishable from A'. Informally, this hides all information in A and A': whether the client is reading/writing to the storage, where the client is accessing, and the underlying data that the client is accessing.

ORAM addresses all the weaknesses discussed in the previous section. First, ORAM is asymptotically efficient: for a database of size N, modern ORAM schemes only need to download/re-upload $O(\mathsf{polylog}\ N)$ data blocks from untrusted memory, per access (as opposed to the $O(N)$ cost of scanning memory). Second, ORAM makes no assumptions on the external memory. Memory is considered untrusted, or actively malicious, and need not manage private keys. Finally, ORAM provides the same level of protection regardless of the access pattern and assumes all memory accesses are visible to the adversary.

Since its proposal by Goldreich and Ostrovsky [9, 13], ORAM has become an important

[1] In particular, HIDE guarantees access pattern privacy as long as $\max_{i,j} |a_i - a_j|$, for memory requests i, j with addresses a_i, a_j, never exceeds a threshold (e.g., 4 KBytes of address space).

part of the cryptographic "swiss army" knife, and has been proposed to secure numerous settings, both practical and theoretical. On the practice side, ORAM has been proposed to secure outsourced storage (e.g., [58]), hide secure processor behavior to external memory (e.g., [49, 66]) and implement searchable encryption with small leakage (e.g., [63]). Additionally on the cryptography side, ORAM has become an important building block in constructing efficient secure multi-party computation protocols (e.g., [52]), proofs of retrievability [59], and Garbled RAM [65].

Despite recent advancements and numerous potential applications, however, the primary impedance to ORAM adoption continues to be its *practical efficiency*. To achieve privacy as advertised, ORAM schemes require that the client continuously *shuffle* (i.e., physically re-locate) data as it is stored on the server. This shuffling has incurred $\Omega(\log N)$ bandwidth blowup between client and server in all ORAM proposals — which translates to $25\times$ to $> 100\times$ overhead in practice. In fact, the seminal work by Goldreich and Ostrovsky [9, 13] proved that the shuffling bandwidth must be *at least* logarithmic in N for an ORAM scheme to be secure.

To confound the problem, the shuffling requires a potentially large amount of trusted storage on the client side, and the most performant schemes require more storage. It is especially challenging to reduce bandwidth overhead *while* maintaining small client storage. This is obviously desirable: ORAM exists to securely *outsource storage*. Indeed, the most performant ORAM schemes (e.g., [73]) require GBytes (to tens of GBytes) of client storage to handle TByte-range ORAMs. This immediately rules out their applicability to settings where the client storage must be small; for example, if it must fit in the on-chip memory of a remote processor (which is the case with the software IP theft and computation outsourcing settings discussed above). On the other hand, the state of the art construction that *can* be deployed in a remote processor (i.e., requires only KBytes to MBytes of client storage) incurs $> 8\times$ the bandwidth overhead of the most performant schemes [71].

1.3 Thesis Contributions and Organization

Given the aforementioned efficiency challenges of modern ORAM schemes, this thesis considers the following question:

Can we design practical ORAM schemes
that simultaneously require only small client storage?

To address this challenge, we present new tools to design small client storage ORAM schemes for theory and practice, and construct the first small client storage ORAM prototype manufactured in hardware silicon. I have developed the results of this thesis in collaboration with the following co-authors: Ling Ren, Albert Kwon, Xiangyao Yu, Marten van Dijk, Srinivas Devadas, Emil Stefanov, Elaine Shi, Daniel Wichs and Dimitrios Serpanos.

Notation: In this section, we differentiate between *online*, *offline* and *overall* client-server bandwidth. By online, we mean just the bandwidth needed between when the client requests a block and receives that block — this is a proxy for access latency and is important in a real system. By overall, we mean the total bandwidth overhead including the shuffling cost. Offline bandwidth is overall minus online bandwidth.

The thesis consists of two main parts.

- Chapters 3 and 4 develop two new and general tools for constructing ORAM schemes, called *bucket-size independence* and *bounded feedback*. Using these tools, we construct

the first **small client storage** ORAM schemes that achieve **constant bandwidth overhead**, while requiring only standard cryptographic assumptions, relative to an insecure system. These results improve on the $\Omega(\log N)$ bandwidth blowups in prior work.

A re-occurring theme which will be central to these results is to allow the server to perform *untrusted computation* in addition to performing memory reads/writes. We use this idea to bypass previous lower bound results [9, 13] and note that it is a realistic assumption in many settings, such as when the memory is located on a remote server with a software stack. In particular, Chapter 3 shows how to achieve constant *online* bandwidth with a small amount of untrusted computation. Chapter 4 shows how to leverage more untrusted computation to achieve constant *overall* bandwidth.

- Chapter 5 describes the design, implementation, and evaluation of the first ORAM controller manufactured into a real silicon chip. This work, taped out in 32 nm silicon, is the first hardware ORAM controller with small client storage, integrity verification, and encryption units. As a proof of concept, we have deployed our design as the on-chip memory controller for a 25 core processor. The ORAM controller passed post tape-out bring-up tests in a lab setting – thereby proving the feasibility of a single-chip secure processor capable of preventing software IP or data theft through the access pattern to main memory.

  This chapter also proposes a number of optimizations to improve ORAM's performance, area and energy consumption in the hardware setting. Most of these optimizations appear in the final silicon chip. A re-occuring theme with the optimizations was to not only improve performance, but to try and *simplify* the design (an especially important consideration when building hardware).

We now give an overview of each chapter. For a detailed narrative on the techniques used in Chapters 3 and 4, see Sections 3.4 and 4.3, respectively.

Chapter 2 – Preliminaries. We start by introducing security definitions for ORAM in several settings and efficiency metrics which will be studied later in the thesis. We then describe the usage settings for ORAM most related to the thesis and give a history of prior work in ORAM starting with the first ORAM schemes by Goldreich and Ostrovsky.

Chapter 3 – Ring ORAM. We present a new ORAM construction called *Ring ORAM*, the first small client storage ORAM to achieve constant online bandwidth. Ring ORAM simultaneously matches or improves upon the overall bandwidth of all prior ORAM schemes (given equal client storage), up to constant factors. The core building block in Ring ORAM is a series of techniques which together allow it to achieve *bucket-size independent* bandwidth, which we also use extensively in Chapter 4.

We highlight the following results, published in [91]:

- Section 3.5: We show how to achieve constant online bandwidth overhead with small client storage, if the server is able to perform perform untrusted base-2 XOR computations. This constitutes a $> 60\times$ online bandwidth improvement over prior small client storage schemes. In absolute terms for a realistic parameterization, this configuration (prior work) has online bandwidth overhead of $1.4\times$ ($80\times$) and overall bandwidth blowup of $59.7\times$ ($160\times$).

- Without server computation, Ring ORAM parameterized for small client storage improves online bandwidth relative to prior work by $4\times$ and overall bandwidth by $2-3\times$.

- Section 3.6: On the theory side, we present a simpler and tighter analysis than prior work. Further (in Section 3.7), we show how to use our analysis to optimally set parameters to minimize bandwidth overhead in practice, given a user-specified client storage budget.

- Section 3.9: Finally, we show how Ring ORAM matches the bandwidth of prior art large client storage ORAM schemes (e.g., [73]) when given comparable client storage. Thus, Ring ORAM is essentially a unified paradigm for ORAM constructions in both large and small client storage settings.

Chapter 4 – Onion ORAM. We present a new ORAM construction called *Onion ORAM*, the first ORAM scheme with constant *worst-case, overall* bandwidth blowup that does not require heavy weight cryptographic primitives such as fully homomorphic encryption (FHE). Onion ORAM circumvents the Goldreich-Ostrovsky [9, 13] lower bound without FHE by combining more efficient additively homomorphic encryption schemes with a new technique called *bounded feedback*. Since our proposal, bounded feedback has been directly adopted by subsequent third party work on constant bandwidth blowup ORAMs [90]. Due to its reliance on cheaper cryptographic techniques, we view Onion ORAM as taking an important step towards *practical* constant bandwidth blowup ORAMs.

This chapter additionally contains the following results, published in [95]:

- We show how to parameterize Onion ORAM using the Damgård-Jurik cryptosystem (Section 4.5) or a somewhat homomorphic encryption scheme such as R-LWE (Section 4.9). (These schemes rely on the Composite Residuosity and Learning with Errors assumptions, respectively.) Both configurations achieve constant worst-case bandwidth blowup, and constant client/server storage blowups — asymptotically optimal in each category.

- Section 4.6: We develop novel techniques to achieve security against a malicious server, without resorting to expensive and non-standard techniques such as SNARKs. In particular, our construction only relies on the existence of collision-resistant hash functions. Taken together, this and the above bullet give us the first constant bandwidth ORAM in the malicious model from standard assumptions.

- We propose a number of optimizations (Sections 4.7, 4.7.4 and 4.9.4) that make asymptotic and constant factor improvements.

- Section 4.7.4: We concretely evaluate the scheme over the Damgård-Jurik cryptosystem, taking into account all constant factors. Using an 8 MByte block size, Onion ORAM improves over prior work by $> 20\times$ and achieves $\sim 8\times$ bandwidth overhead in absolute terms.

Chapter 5 – Tiny ORAM, integrated with the Ascend Processor. We design and implement *Tiny ORAM*, the first ORAM controller built into silicon and the first hardware ORAM controller with small client storage, integrity verification, or encryption units. The design, written in Verilog, has been open sourced at `http://kwonalbert.github.io/oram`.

We built Tiny ORAM to be integrated into an Ascend processor, the author's prior collaborative work [49, 60], which can be used to protect processor access pattern in settings such as software IP theft prevention and secure computation outsourcing (Section 1). Thus, a major focus of this chapter is also to propose new optimizations that apply to the hardware/secure processor, and to evaluate those optimizations in the context of a real prototype. To that end, this chapter contains the following results:

- We propose techniques to securely cache (Section 5.5.1) and compress (Section 5.5.2) the ORAM *position map*, the central structure in modern ORAM schemes which constitutes the bulk on client (on-chip) storage. These techniques reduce extra bandwidth needed to maintain the position map by 95%.

- Section 5.5.3: We propose a new integrity scheme for position-based ORAM that is asymptotically optimal in its hash bandwidth requirements (a $> 68\times$ improvement over prior work). This scheme and those in the previous bullet appear in [86].

- In Section 5.6, we present techniques to avoid hardware performance bottlenecks due to DRAM architecture, memory bandwidth through the ORAM stash (a crucial client datastructure), and the ORAM's encryption units. The last technique (Section 5.6.3) also reduces necessary encryption bandwidth at the algorithm level by $\sim 3\times$. These schemes appear in [69] and [87].

- Finally, we present a thorough evaluation of our designs in simulation (Section 5.7), on an FPGA board (Section 5.8) and via an ASIC tape-out (Section 5.9-5.10). The post tape-out ASIC results appear in [96]. For completeness, we also simulate expected benefits of Ring ORAM in hardware (Section 5.7.6).

  In simulation, our hardware ORAM (Ring ORAM, projected) runs a suite of benchmarks with an average $4\times$ ($3\times$) slowdown vs. an insecure system. This corresponds to latencies measured from our ASIC prototype, which can lookup a 1 GByte non-recursive ORAM lookup for a 512 bit block in $\sim$ 1275 processor cycles (an insecure access costs $\sim$ 58 cycles). The ASIC design has a hardware area of .51 mm^2 in 32 nm technology, runs at 857 MHz, and consumes 299 mW of power – **measured on real hardware post tape-out**.

The evaluation results are encouraging. Our hardware ORAM, when integrated into the 25 core test chip, is *small*: roughly the size of a processor core or $\sim 1/72$-th the area of the test chip. At 857 MHz and the bandwidths described, the ASIC is capable of servicing 43 MB of client requests per second, with an access latency of 1.4 μs. This means running programs on hardware ORAM can have similar overheads to running a language in an interpreter.

Chapter 6 – Conclusion. We summarize the results of the thesis and discuss avenues for future work.

Chapter 2

Preliminaries

In this chapter, we give formal definitions for ORAM. We first give a strong and general security definition, which achieves simulator-based security against a malicious adversary. We then discuss ORAM metrics and how they impact practice, and give a history of prior work.

2.1 ORAM Definition

Following Apon et al. [72], we define ORAM as a reactive two-party protocol between the client and the server, and define its security in the Universal Composability model [20]. We use the notation

$$((c\_out, c\_state), (s\_out, s\_state)) \leftarrow \mathsf{protocol}((c\_in,\ c\_state), (s\_in,\ s\_state))$$

to denote a (stateful) protocol between a client and server, where *c_in* and *c_out* are the client's input and output; *s_in* and *s_out* are the server's input and output; and *c_state* and *s_state* are the client and server's states before and after the protocol.

Definition 1 (Oblivious RAM (ORAM))**.** *An ORAM scheme consists of the following interactive protocols between a client and a server.*

- $((\perp, \mathcal{C}), (\perp, \mathcal{D})) \leftarrow \mathsf{Setup}(1^\lambda, (D, \perp), (\perp, \perp))$: An interactive protocol where the client's input is a memory array $D[1..N]$ where each memory *block* has bit-length B; and the server's input is $\perp$. At the end of the Setup protocol, the client has secret state $\mathcal{C}$, and server's state is $\mathcal{D}$ (which typically encodes the memory array D).
- $((data, \mathcal{C}'), (\perp, \mathcal{D}')) \leftarrow \mathsf{Access}((\mathsf{op}, \mathcal{C}), (\perp, \mathcal{D}))$: To access data, the client starts in state $\mathcal{C}$, with an input op where $\mathsf{op} := (\mathsf{read}, addr)$ or $\mathsf{op} := (\mathsf{write}, addr, data)$; the server starts in state $\mathcal{D}$, and has no input. In a correct execution of the protocol, the client's output *data* is the current value of the memory D at location *addr* (for writes, the output is the old value of $D[addr]$ before the write takes place). The client and server also update their states to $\mathcal{C}'$ and $\mathcal{D}'$ respectively. The client outputs $data := \perp$ if the protocol execution aborted.

We say that the ORAM scheme is *correct*, if for any initial memory $D \in \{0,1\}^{BN}$, for any operation sequence $\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_m$ where $m = \mathrm{poly}(\lambda)$, an $\mathsf{op} := (\mathsf{read}, addr)$ operation would always return the last value written to the logical location *addr* (except with negligible probability).

2.2 Security Definition (Simulator)

We will adopt two security definitions throughout this thesis. The first follows a standard simulation-based definition of secure computation [18], requiring that a real-world execution "simulate" an ideal-world (reactive) functionality $\mathcal{F}$.

Ideal world. We define an ideal functionality $\mathcal{F}$ that maintains an up-to-date version of the data D on behalf of the client, and answers the client's access queries.

- *Setup.* An environment $\mathcal{Z}$ gives an initial database D to the client. The client sends D to an ideal functionality $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ of the fact that the setup operation occurred as well as the size of the database $N = |D|$, but not of the data contents D. The ideal-world adversary $\mathcal{S}$ says ok or abort to $\mathcal{F}$. $\mathcal{F}$ then says ok or $\perp$ to the client accordingly.

- *Access.* In each time step, the environment $\mathcal{Z}$ specifies an operation $\mathsf{op} := (\mathsf{read}, addr)$ or $\mathsf{op} := (\mathsf{write}, addr, data)$ as the client's input. The client sends op to $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ (without revealing to $\mathcal{S}$ the operation op). If $\mathcal{S}$ says ok to $\mathcal{F}$, $\mathcal{F}$ sends $D[addr]$ to the client, and updates $D[addr] := data$ accordingly if this is a write operation. The client then forwards $D[addr]$ to the environment $\mathcal{Z}$. If $\mathcal{S}$ says abort to $\mathcal{F}$, $\mathcal{F}$ sends $\perp$ to the client.

Real world. In the real world, an environment $\mathcal{Z}$ gives an honest client a database D. The honest client runs the Setup protocol with the server $\mathcal{A}$. Then, at each time step, $\mathcal{Z}$ specifies an input $\mathsf{op} := (\mathsf{read}, addr)$ or $\mathsf{op} := (\mathsf{write}, addr, data)$ to the client. The client then runs the Access protocol with the server. The environment $\mathcal{Z}$ gets the view of the adversary $\mathcal{A}$ after every operation. The client outputs to the environment the data fetched or $\perp$ (indicating abort).

Definition 2 (Simulation-based security: privacy + verifiability)**.** *We say that a protocol $\Pi_{\mathcal{F}}$ securely computes the ideal functionality $\mathcal{F}$ if for all probabilistic polynomial-time real-world adversaries (i.e., server) $\mathcal{A}$, there exists an ideal-world adversary $\mathcal{S}$, such that for all non-uniform, polynomial-time environments $\mathcal{Z}$, there exists a negligible function* negl *such that*

$$|\Pr\left[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1\right] - \Pr\left[\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1\right]| \leq \mathsf{negl}(\lambda)$$

At an intuitive level, our definition captures the privacy and verifiability requirements for an honest client (the client is never malicious in our setting), in the presence of a malicious server. The definition simultaneously captures *privacy* and *verifiability*. Privacy ensures that the server cannot observe the data contents or the access pattern (the contents of any op_i). Verifiability ensures that the client is guaranteed to read the correct data from the server.

2.3 Security Definition (Termination Channel Leakage)

The two outcomes from running the protocol in the previous section are: (1) the adversary deviates from the protocol, which may cause the client to prematurely abort, or (2) the adversary lets the protocol complete. In (1), which we call the *termination channel*, there is no privacy leakage since we require the existence of $\mathcal{S}$ that, for all $\mathcal{Z}$, can predict when the

termination occurs a-priori (i.e., independent of the access pattern given by $\mathcal{Z}$). In (2), there is some leakage: the adversary learns how many accesses were requested by $\mathcal{Z}$. However, we are not interested in preventing this leakage since it once again does not depend on the operations (the access pattern) submitted by $\mathcal{Z}$.

In this section, we want to relax this definition to permit some small, but access pattern-related, leakage through the termination channel. Doing so will enable several performance optimizations in Chapter 5 (hardware ORAM).[1] Informally, the change to the real world adversary's view is the following. If the adversary is semi-honest, the protocol terminates when $\mathcal{Z}$ stops submitting operations and the adversary only learns (2) above, as with the previous definition. If the adversary is malicious, the protocol may terminate before (2) occurs, but *when* it terminates will be a function of the adversary's strategy, randomness in the protocol, and importantly the access pattern specified by $\mathcal{Z}$. For the purposes of satisfying the definition, we wish to show that when/if the protocol terminates is the only new information the adversary is able to learn. More formally,

Definition 3 (Termination channel security: privacy)**.** *An ORAM scheme leaks privacy only over the termination channel if for every (malicious) adversary $\mathcal{A}$, there exists a functionality $\mathcal{F}$ such that the following two distributions are computationally indistinguishable.*

1. *(Real world). Same as the real world in Section 2.2, where the real-world distribution is the adversary $\mathcal{A}$'s view of the protocol. We denote with a bit b whether the client decided to abort (output $\perp$) at some point in the protocol. The number of accesses made by the client until it stops making requests (or aborts) is denoted m'.*

2. *(Ideal world). Same as the real world experiment, except for the following change. For a given experiment in the real world, the client's interactions with the server are replaced by the functionality $\mathcal{F}$, which is given b and m' but not the client data requests. Throughout the protocol, $\mathcal{F}$ produces a view (the ideal-world distribution) for the adversary $\mathcal{A}$ which is a function of only b and m'.*

Definition 4 (Termination channel security: verifiability/integrity)**.** *Consider the following correctness experiment. The client and $\mathcal{A}$ run m' rounds of the* Access *protocol, at which point the protocol naturally terminates or prematurely aborts as described above. Correctness requires that except with negligible probability $\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_{m'-1}$ are* correct *(if the client aborted) or $\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_{m'}$ are* correct *(otherwise). Correctness of each trace follows the definition from Section 2.1 and is from the perspective of $\mathcal{Z}$.*

2.4 Metrics

We will gauge ORAM schemes primarily on the following performance metrics. *Note regarding notation:* Metrics are in bits unless otherwise specified.

Client/server storage. The client/server's storage, given by $|\mathcal{C}|$ and $|\mathcal{D}|$ in the above definitions, refers to the number of blocks held by the client and server at the start of an Access operation. In all the schemes we describe, the client/server storage after and during each call to Access will be the same asymptotically as the starting size, so we will

[1] Note that this change will leak non-negligible privacy. We will be explicit as to where it is used in Chapter 5.

not distinguish these cases. Following conventions from prior work, we say client storage is *small* if it is $O(B \cdot \mathsf{polylog}\ N)$ and *large* if it is $\Omega(B\sqrt{N})$ — where B is the data block size in bits. We consider an insecure block storage system to require $O(BN)$ server storage and $O(B)$ client storage, thus this is optimal for an ORAM as well.

Client-server bandwidth. Client-server bandwidth (bandwidth for short) refers to the number of blocks sent between the client and server to serve all Access operations, over the number of accesses made (i.e., is amortized). Insecure block storage systems require $O(B)$ bandwidth. When we say an ORAM requires $O(B \log N)$ bandwidth, this may also be interpreted as $O(\log N)$ bandwidth *blowup/overhead* relative to the insecure system. Note that some ORAM schemes only achieve their best bandwidth given large-enough blocks. If the allowed block size is larger than the client application's desired block size, the bandwidth blowup increases proportionally.

In addition to the primary metrics, we will analyze the following as they become relavent to different constructions.

Online bandwidth. The online bandwidth during each access refers to the blocks transferred before the access is completed from the client's point of view. By the "client's point of view," we are mainly interested in the case when the access type is read: i.e., online bandwidth represents the critical-path operation, the time between when the client requests a block and receives that block. To hide whether the operation type is read or write, however, ORAM schemes typically make Access perform the same operations from the server's perspective, regardless of operation type. So, for the rest of the thesis online bandwidth will refer to the blocks transferred before data is returned to the client, as if every client operation was a read.[2] After the online phase of Access, more block transfers may be required before the access is complete, which we call the *offline* phase.

Worst-case bandwidth. The worst-case bandwidth refers to the per-Access bandwidth if amortization is not possible. For certain ORAM schemes, the bandwidth per call to Access is naturally the same for every call (in which case worst-case equals bandwidth). In other schemes (including ones that we present), offline bandwidth can be pushed to future calls to Access to improve the online bandwidth of multiple consecutive requests. This is to improve performance of "bursty workloads:" if the client must make two read requests before proceeding in its computation, the effective online bandwidth is the online bandwidth of both calls to Access and the offline bandwidth of the first call to Access.

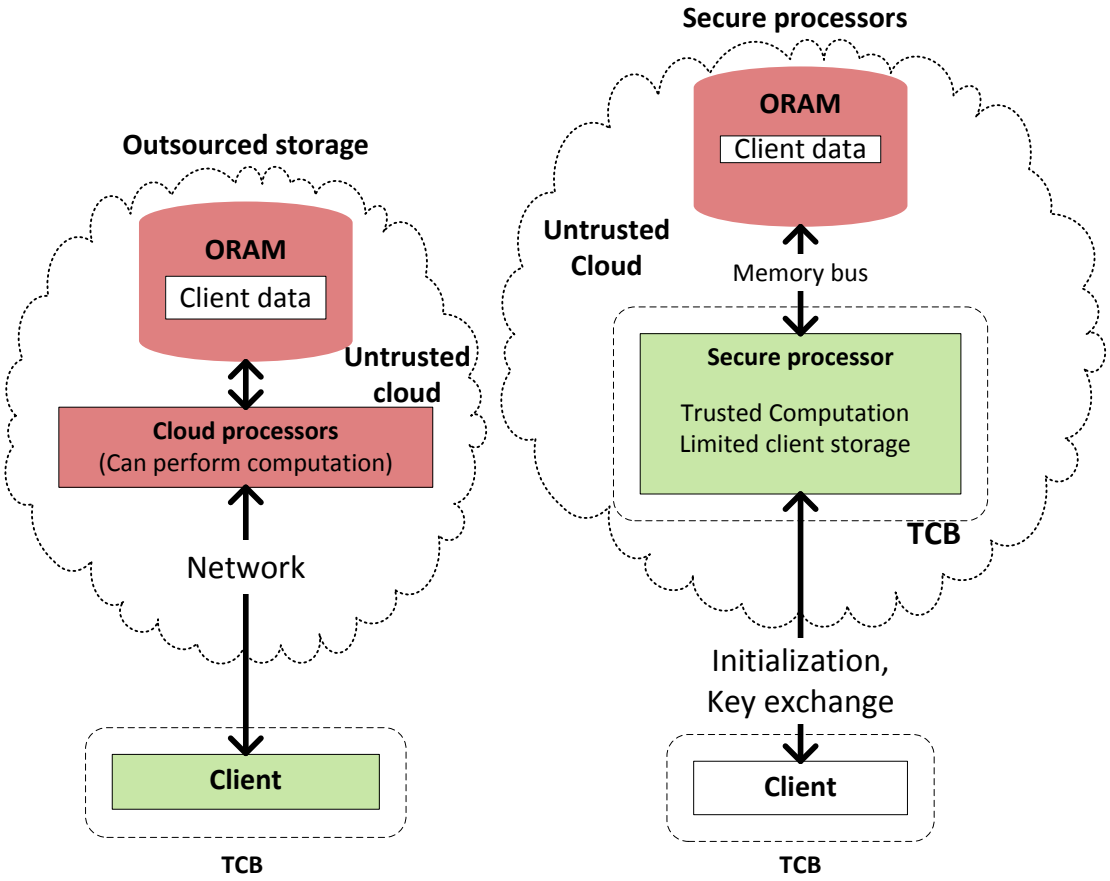
Server computation. The server computation is the amount of *untrusted, local* computation performed by the server, in addition to performing simple memory read/write operations. In the first ORAM papers [9, 13], the server is assumed to only perform read and write operations to untrusted storage. In practice, many recent constructions have implicitly assumed the server is able to perform some amount of computation on data to reduce client-server bandwidth. Depending on the amount of computation, the computation may become the system bottleneck.

[2]The ORAM literature sometimes makes this distinction by saying logical write operations return $\perp$ to the client.

Other metrics: Number of round-trips. The number of round-trips refers to the round-trip block traffic between client and server during each call to Access. As in regular systems design, more roundtrips means worse performance since future operations must wait for the interconnect latency between client and server.

2.5 Settings

Many of the techniques presented in the thesis are general and help improve any ORAM deployment. We mention two settings in later chapters, described below (Figure 2-1).



(a) File server setting (b) Secure processor setting

Figure 2-1: Outsourced storage and secure processor ORAM usage settings. Green is trusted.

Outsourced storage. Here, a client (e.g., a mobile device or in-house data management system) wishes to securely store data on a remote storage provider. We assume the storage provider acts as block storage (e.g., Amazon S3): the operations exposed to the client are to read/write blocks of data [83]. The trusted computing base (TCB) is the client machine: *we wish to eliminate access pattern leakage, given a potentially malicious adversary, at all points beyond the client (e.g., the network, server, etc).*

Secure processor. Here, a client wishes to outsource computation to a server or to obfuscate its execution in an Internet-of-Things (IoT) environment. In the outsourcing setting, there is a remote client, a secure processor on the server, and the rest of the server state (e.g., its DRAM / disk hierarchy). In a setup phase, the client loads data and (possibly) a program into the secure processor using conventional secure channels and attestation techniques. Once setup, the secure processor computes the result of running the program on the privided data, and sends it back to the client (also using secure channels). In the IoT setting, a processor collects and computes on data in a hostile environment where the adversary may have physical access to the device. The TCB is the secure processor and the remote client (if one exists).

As the program runs, *we wish to eliminate access pattern leakage to main memory, given a potentially malicious adversary, when last-level cache (LLC) misses occur.* Several possible attacks include cold boot [35], intercepting data on the memory bus [23, 16], BIOS flashing [2, 37], and in general multiple processors (or helper modules such as the Intel Management Engine [94]) sharing main memory in space or time.

2.6 Related Work

We now review prior ORAM work more generally, as well as in the contexts described in the previous section.

2.6.1 ORAM History

We start by describing the three main families of ORAM schemes. The goal is to show the progression of ideas over time, and to introduce several core techniques that will be used later in the thesis. The three schemes detailed below all require $O(B)$ client storage (asymptotically optimal).

Square root ORAM (1987)

The study of ORAM was initiated by Goldreich [9], who sought to address the problem of software IP theft. This problem is similar to our secure processor setting: for a program running on a remote secure processor, one wishes to hide a program's control flow as determined by the address pattern to main memory. The trivial solution is to scan all of memory on each access, which has $O(BN)$ online/overall bandwidth.

To address the high online bandwidth in the trivial scheme, Goldreich proposed the square root ORAM. In this design, the server memory is into two regions: a main $O(N)$ block region and a shelter of size $O(\sqrt{N})$ blocks. The main region is filled with $O(N)$ real blocks and $O(\sqrt{N})$ dummy blocks. All blocks are encrypted using a semantically secure scheme and shuffled together. This mechanism of **permuting an array** full of real blocks and dummy blocks will be used heavily throughout the thesis. How the permutation is selected is implementation dependent; the square root ORAM uses a random oracle / hash function followed by an oblivious sort.

To make an access, the client first scans the entire shelter. If the block is found there, the client reads a random, previously unread dummy block from the main region. Otherwise, the client uses the hash function to determine the address of the block of interest in the main region. Finally, the real or dummy block is re-encrypted and appended to the shelter.

Thus, the online bandwidth is $O(B\sqrt{N})$. The intuition for security is that each read scans the shelter, and performs a read to a random, previously unread slot in the main region.

Every $O(\sqrt{N})$ accesses, the main region runs out of dummies and must be fully re-permuted by the client. [9] achieves this *eviction* step by using an oblivious sort and a new keyed hash function to re-mix the shelter into the main region and re-permute the main region. Using the sorting algorithm described in the paper, this step requires $O(B)$ client storage, and $O(BN \log N)$ bits to be transferred every $O(\sqrt{N})$ accesses, giving the scheme a $O(B\sqrt{N} \log N)$ amortized bandwidth.

Hierarchical ORAM (1996)

Goldreich and Ostrovsky proposed the hierarchical ORAM [13] to improve the online and overall bandwidth of the square root algorithm. The key idea is to, instead of having one main memory region and shelter, organize the server as a pyramid of permuted arrays where each array is geometrically (e.g., a factor of 2) larger than the previous array. Each permuted array acts as the main region in the square root ORAM, and thus is parameterized by a hash function and has space reserved for dummy blocks.

To access a block, each level in the pyramid is accessed as if it were the main region in the square root ORAM. To avoid collisions in the hash function for the smaller levels, each slot in each permuted array is treated as a bucket of size $O(\log N)$ blocks. The hash function now maps blocks to random buckets. Buckets are downloaded atomically by the client when read/written to, and the bucket size is set to make overflow probability negligible. Thus, online bandwidth is $O(\log^2 N)$: the cost to access $O(\log N)$ buckets (one per level in the pyramid) of $O(\log N)$ blocks each.

Instead of scanning a shelter, each block accessed is appended to the smallest level of the pyramid after that access. Eventually (like the shelter), the top (or *root*) of the pyramid will fill, and an eviction step must merge it into the second pyramid level. When the second level fills, it along with the first level is merged into the third level, so on to the largest level of $O(N)$ blocks. In general, merging levels 0 through i involves completely re-shuffling the contents of those levels into a new array which becomes level $i + 1$. The worst case and amortized bandwidth cost of this operation is $O(N \log^2 N)$ and $O(\log^3 N)$ blocks, respectively.

Tree ORAM (2011)

Shi et al. [46] proposed the tree ORAM to decrease the worst-case bandwidth cost of the hierarchical ORAM to be $O(\mathsf{polylog} N)$ blocks. All ORAM schemes in this thesis are tree ORAMs.

The key idea in the tree ORAM is that, instead of blocks stored in level i having complete freedom on where they will be re-shuffled into in level $i+1$ (as with the hierarchical solution), blocks may only live in a single pre-ordained bucket per level. This is accomplished by connecting the buckets in the hierarchical ORAM pyramid as if they were nodes in a **binary tree**, and associating each block to a **random path** of buckets from the top bucket (the *root*) bucket) to a leaf in the tree.

To access a block the client first looks up a **position map**, a table in client storage which tracks the path each block is currently mapped to, and then reads all the buckets on the block's assigned path. The scheme achieves access pattern privacy by **re-mapping** the accessed block to a new random path when it is accessed. Similar to [13], the tree ORAM

requires buckets to be size $O(\log N)$ for reasons that will be described below. Thus, online bandwidth is also $O(B \log^2 N)$.

Similar to the hierarchical ORAM, each block accessed is appended to the root bucket at the end of each access. To prevent the root bucket (or any other bucket) from overflowing, an eviction procedure downloads $O(1)$ buckets per level per access to try and push blocks down the tree subject to blocks needing to stay on their assigned paths. This operation has an amortized *and* worst-case bandwidth overhead of $O(B \log^2 N)$.

ORAM Recursion. As described, the position map is $\Omega(N \log N)$ bits in size which is too large. To address this issue, Shi et al. [46] also proposed ORAM recursion to reduce the client storage to $O(\log N)$ bits. The basic idea is to store the position map in a separate ORAM, and store the new ORAM's (smaller) position map on-chip. This may be performed recursively until the client storage due to the final position map is $O(\log N)$ bits. Clearly, looking up a block for the client now requires looking up all the position map ORAMs in addition to the main data ORAM. Using the parameters in [46], the amortized bandwidth (including the cost of recursion) becomes $O(B \log^3 N)$. (More detail about recursion is given in Chapter 5.)

2.6.2 State of the Art ORAMs

We now review the most efficient ORAMs, which are improvements to the above families.

The most efficient schemes asymptotically are due to Goodrich et al. [42], Kushilevitz et al. [54] and Circuit ORAM [5]. In the large client storage setting, Goodrich et al. [42] achieves a bandwidth of $O(B \log N)$ with client storage $O(BN^\epsilon)$ (for $\epsilon > 0$). With small client storage, Kushilevitz et al. [54] achieves $O(B \log^2 N / \log \log N)$ bandwidth with $O(B)$ client storage. Both of these schemes assume $B = \Omega(\log N)$ and are hierarchical ORAMs. On the other hand, Circuit ORAM is a tree ORAM that achieves $\omega(B \log N)$ bandwidth with $O(B)$ client storage, but only when $B = \Omega(\log^2 N)$.

Counting constant factors, the most bandwidth efficient schemes are the SSS ORAM [56] and Path ORAM [71]. The SSS ORAM uses $O(B\sqrt{N})$ client storage to achieve $O(B \log^2 N)$ bandwidth. In practice, the SSS ORAM is often parameterized with $c * O(BN)$ client storage (for $c \ll 1$), where it can achieve $O(B \log N)$ bandwidth with hidden constant 1. Path ORAM uses $\omega(B \log N)$ client storage to achieve $O(B \log N + \log^3 N)$ bandwidth. Depending on parameters, Path ORAM's hidden bandwidth constant is ≥ 8. SSS is a hierarchical ORAM while Path ORAM is a tree ORAM.

All of the above schemes that we described explicitly (as well as the schemes we construct in this thesis) have $O(BN)$ server storage — asymptotically optimal.

2.6.3 ORAM in Specific Settings

For the next several subsections, we give ORAM schemes and systems built that are relevant to the settings from Section 2.5.

ORAM for Outsourced Storage

Several systems have been built to enable ORAM in the outsourced storage setting, including PrivateFS [58], Shroud [64], Oblivistore [70] and Curious [83]. PrivateFS and Oblivistore follow the traditional ORAM model, and Oblivistore (based on the SSS ORAM [56])

claims the best performance (between 10× and 100× bandwidth overhead). Oblivistore can trivially be combined with another construction called Burst ORAM [73] to decrease bandwidth further. Shroud assumes multiple trusted co-processors running on the server to reduce bandwidth.

Server Computation ORAM

Many state-of-the-art ORAM schemes or implementations (especially in the outsourced storage model) make use of server computation. Two recent works, Burst ORAM [73] and Path PIR [79], use server computation to reduce online bandwidth to $O(B)$ bits. The SSS ORAM [56, 70] and Burst ORAM [73] assumed the server is able to perform matrix multiplication or XOR operations. Path-PIR [79] and subsequent work [84, 81] increased the allowed computation to additively homomorphic encryption. Apon et al. [72] and Gentry et al. [61, 3] further augmented ORAM with Fully Homomorphic Encryption (FHE). Williams et al. [57] and Bucket ORAM [85] rely on server computation to achieve a single online roundtrip.

Server Computation ORAM vs. Other Cryptographic Primitives

We remark that recent works on Garbled RAM [75, 65] can also be seen as generalizing the notion of server computation ORAM. However, existing Garbled RAM constructions incur $\mathsf{poly}(\lambda)\cdot\mathsf{polylog}(N)$ client work and bandwidth blowup, and therefore Garbled RAM does not give a server-computation RAM with a competitive bandwidth blowup. Reusable Garbled RAM [76] achieves constant client work and bandwidth blowup, but known reusable garbled RAM constructions rely on non-standard assumptions (indistinguishability obfuscation, or more) and are prohibitive in practice.

Secure Processors and ORAM

Academic work on single-chip (tamper-resistant) secure processors goes back to eXecute Only Memory (XOM) [24, 25, 19] and Aegis [26, 29]. In XOM, security requires applications to run in secure compartments, where both instructions and data are encrypted and from which data can escape only on explicit request from the application itself. Aegis, a single-chip secure processor, performs memory integrity verification and encryption on all data written to main memory so as to allow external memory to be untrusted. Intel SGX's memory encryption engine is very similar to this mechanism [62, 94].

Recently, a secure processor called Ascend [49, 60] was proposed to improve security over the main memory channel through hardware ORAM in addition to encryption/integrity checking. The goal of Ascend is to run *untrusted* programs and still maintain data privacy/integrity from an adversary with unrestricted access to main memory. We note that Ascend's threat model requires more protection than just the access pattern channel (e.g., it aims to protect the timing channel [74]). These additional channels are outside the scope of the thesis.

Concurrently, Maas et al. built Phantom [66], the first hardware implementation of ORAM on FPGAs. Tiny ORAM (Chapter 5) can be seen as follow-on work to Phantom, and may serve as the on-chip memory controller in the Ascend processor. Further, Chang et al. [89] built Ghostrider, a system which uses programming language techniques to decide when and how to access ORAM. Ghostrider is complementary to this thesis in the sense that Ghostrider relies on hardware ORAM as a primitive.

Notation	Meaning
N	Number of real data blocks in ORAM
B	Data block size in bits
$\mathcal{Z}$	Environment/running program
$\mathcal{A}$	Adversary (server/memory)
Introduced for Ring ORAM (Chapter 3)	
L	Depth of the ORAM tree
Z	Maximum number of real blocks per bucket
S	Number of slots reserved for dummies per bucket
A	Eviction frequency (larger means less frequent)
G	Eviction counter
T	Maximum stash occupancy (not counting the transient path)
$\mathcal{P}(l)$	Path from the root bucket to leaf l
$\mathcal{P}(l, i)$	The i-th bucket (towards the root) on $\mathcal{P}(l)$
$\mathcal{P}(l, i, j)$	The j-th slot in bucket $\mathcal{P}(l, i)$
Introduced for Onion ORAM (Chapter 4)	
C	The number of chunks in each data block
B_C	Chunk size in bits ($B = C \cdot B_C$)
$\mathcal{V}$	The set of chunk indices corresponding to verification chunks
Introduced for Tiny ORAM (Chapter 5)	
H	The number of ORAMs in the recursion
X	The number of leaves stored per PosMap block

Table 2.1: Thesis parameters and notations.

2.7 Summary of Notations

Table 2.1 overviews this thesis' commonly used notations. All logarithms (if left unspecified) are base 2.

Chapter 3

Ring ORAM:

An Efficient Small Client Storage ORAM

This chapter presents Ring ORAM, the most bandwidth-efficient ORAM scheme for the small client storage setting. At the core of the construction is an ORAM that achieves "bucket-size independent bandwidth," which unlocks numerous performance improvements. In practice, Ring ORAM's overall bandwidth is 2.3× to 3× better than the prior-art scheme for small client storage. Further, if memory can perform simple untrusted computation, Ring ORAM achieves constant online bandwidth (∼ 60× improvement over prior art for practical parameters). On the theory side, Ring ORAM features a tighter and significantly simpler analysis than prior art. Finally, we show how to parameterize Ring ORAM for the large client storage setting, where its efficiency matches the state-of-the-art schemes in that setting.

Historically, research to apply ORAM in practice has followed two divergent tracts, depending on the application domain. When the client has a large storage budget, hierarchical ORAMs [15] such as the SSS construction [56, 73] are preferred since they can achieve bandwidths as low as $1 * B \log N$. In practice, however, these constructions' client storage budgets may be tens of GBytes, which may be too large for a space-constrained client.

For example, when the client is a remote secure processor, client storage is restricted to the processor's scarce on-chip memory (KBytes to MBytes). In that setting, tree ORAMs [46] such as Path ORAM [71] are required, which has bandwidth $8 * B \log N$. Unfortunately, hierarchical ORAMs and tree ORAMs look quite different, and the above properties have had the effect of splitting the community's efforts. Ideally, one wishes for a single ORAM scheme that can be parameterized for and be performance-competitive in any setting.

Ring ORAM takes a step in this direction, by improving on Path ORAM when parameterized for small storage and matching the SSS ORAM when parameterized for large storage. Ring ORAM is also the first small client storage ORAM to achieve constant online bandwidth. The majority of this chapter focuses on the small client storage setting and Path ORAM, so we now provide a summary of that scheme. In Section 3.9, we show how to parameterize Ring ORAM for the large client storage setting.

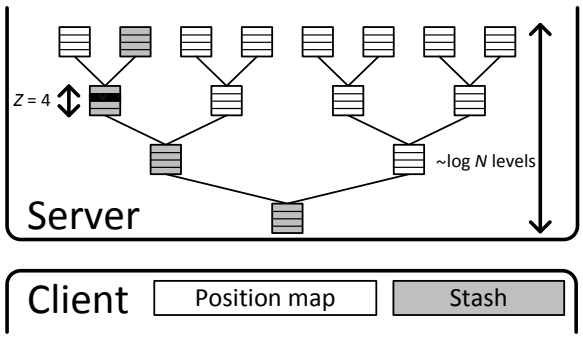


Figure 3-1: Path ORAM server and client storage. Suppose the **black** block is mapped to the shaded path. In that case, the block may reside in any slot along the path or in the stash (client storage).

3.1 Path ORAM Overview

We now give a *brief* overview of Path ORAM as it is relevant to Ring ORAM. The ORAM we later build into hardware (Chapter 5) will be based on Path ORAM, and we will describe it in more detail there.

Path ORAM (and Ring ORAM) is a tree ORAM [46] (Section 2.6.1) and server storage is structured as a binary tree of roughly $L = O(\log N)$ levels. Each node in the tree is a bucket that can hold up to a small - $O(1)$ - number Z of data blocks. Each path in the tree is defined as the sequence of buckets from the root of the tree to some leaf node. Each block is mapped to a random path, and must reside somewhere on that path.

As with the original tree ORAM: To access a block, the client first looks up a position map, a table in client storage which tracks the path each block is currently mapped to, and then reads all the ($\sim Z \log N$) blocks on that path into a client-side data structure called the stash. The requested block is then remapped to a new random path and the position map is updated accordingly. Lastly, the algorithm invokes an eviction procedure which writes the same path we just read from, percolating blocks down that path.

We remind readers not to confuse the above read/write path operation with reading/writing data blocks. In ORAM, both reads and writes to a data block are served by the read path operation, which moves the requested block into client storage to be operated upon secretly. The sole purpose of the write path operation is to evict blocks from the stash and percolate blocks down the tree.

Recursion. Recursion for Path ORAM and Ring ORAM is the same as with the tree ORAM (Section 2.6.1). In this chapter, we assume both Path ORAM and Ring ORAM use instances of themselves for recursion. Thus, for simplicity, most of the analysis in this chapter will be for a single instance ("the non-recursive" version) of both schemes.

Parameters and Costs. Path ORAM's bandwidth is $2Z \log N$ because each access reads and writes a path in the tree. To prevent blocks from accumulating in client storage, the bucket size Z has to be at least 4 (experimentally verified [71, 66]) or 5 (theoretically proven [4]).

Table 3.1: **Our contributions.** Bandwidths are given for non-recursive versions of each scheme. Ranges in constants for Ring ORAM are due to different parameter settings. XOR refers to the XOR technique from [73].

	Online Bandwidth	Overall Bandwidth
Path ORAM	$ZB \log N = 4B \log N$	$2BZ \log N = 8B \log N$
Ring ORAM	$\sim 1 * B \log N$	(3 to 3.5) $*B \log N$
Ring ORAM + XOR	$\sim B$	(2 to 2.5) $*B \log N$

3.2 Path ORAM Challenges

Despite being a huge improvement over prior schemes, Path ORAM is still plagued with several important challenges. First, the constant factor $2Z \geq 8$ is substantial, and brings Path ORAM's bandwidth overhead to $> 150\times$ for practical parameterizations. In contrast, the SSS construction does not have this bucket size parameter and can achieve close to $1 \cdot \log N$ bandwidth. (This bucket-size-dependent bandwidth is exactly why Path ORAM is dismissed in the large client storage setting.)

Second, despite the importance of overall bandwidth, online bandwidth—which determines response time—is equally, if not more, important in practice. For Path ORAM, half of the overall bandwidth must be incurred online. Again in contrast, an earlier work [73] reduced the SSS ORAM's online bandwidth to $O(1)$ blocks by granting the server the ability to perform simple XOR computations. Unfortunately, their techniques do not apply to Path ORAM, or any small client storage ORAM that has competitive practical performance.

3.3 Contributions

Ring ORAM is a careful re-design of the tree-based ORAM to achieve an online bandwidth of $O(1)$, and the amortized overall bandwidth is independent of the bucket size. We compare analytic bandwidth overhead to Path ORAM in Table 4.2. The major contributions of Ring ORAM include:

- **Small online bandwidth.** We provide the first small client storage ORAM scheme that achieves ~ 1 online bandwidth, relying only on very simple, untrusted computation logic on the server side. This represents at least $60\times$ improvement over Path ORAM for reasonable parameters.
- **Bucket-size independent overall bandwidth.** While all known tree-based ORAMs incur an overall bandwidth cost that depends on the bucket size, Ring ORAM eliminates this dependence, and improves overall bandwidth by 2-3$\times$ relative to Path ORAM in the small client storage regime.
- **Simple and tight theoretical analysis.** Using novel proof techniques based on Ring ORAM's eviction algorithm, we obtain a much simpler and tighter theoretical analysis than that of Path ORAM. As a byproduct of the analysis, we develop an analytic model for setting Ring ORAM's parameters to achieve optimal bandwidth.

Extension to larger client storage. As mentioned, as an interesting by-product, Ring ORAM can be easily extended to achieve competitive performance in the large client storage setting. This makes Ring ORAM a good candidate in oblivious cloud storage, because as

Table 3.2: **Snapshot of overheads (relative to insecure system).** Online bandwidth is the bandwidth needed to serve a request and overall bandwidth includes background work. XOR refers to the XOR technique from [73] and level comp refers to level compression from [56] (these are mutually exclusive optimizations with different trade-offs). We show Ring ORAM without the XOR technique for small client storage in case server computation is not available in that setting. The constant c in the large client storage case is very small for realistic block sizes: i.e., for this parameterization the cN term constitutes 1/16 of total client storage. Bandwidth in the small client storage case does not include the cost of recursion which is negligible in practice for the large (e.g., 4 KByte) block sizes we consider.

	Practical, Asymptotic					
	Online Bandwidth		Overall Bandwidth		Client Storage	Server Storage
Large client storage (ORAM capacity = 64 TeraBytes, Block size 256 KBytes)						
SSS ORAM [56] (level comp)	7.8×	$O(\log N)$	31.2×	$O(\log N)$	16 GBytes	$3.2N$
SSS ORAM [56] (XOR)	1×	$O(1)$	35.7×	$O(\log N)$	16 GBytes	$3.2N$
Path ORAM [71]	60×	$O(\log N)$	120×	$O(\log N)$	16 GBytes	$8N$
Ring ORAM, XOR	1×	$O(1)$	26.8×	$O(\log N)$	16 GBytes	$6N$
Small client storage (ORAM capacity = 1 TeraByte, Block size 4 KBytes)						
Path ORAM	80×	$O(\log N)$	160×	$O(\log N)$	3.1 MBytes	$8N$
Ring ORAM	20.4×	$O(\log N)$	79.3×	$O(\log N)$	3.1 MBytes	$6N$
Ring ORAM, XOR	1.4×	$O(1)$	59.7×	$O(\log N)$	3.1 MBytes	$6N$

a tree-based ORAM, Ring ORAM is easier to analyze, implement and de-amortize than hierarchical ORAMs like SSS. Therefore, Ring ORAM is essentially *a unified paradigm for ORAM constructions in both large and small client storage settings.* To give readers a sense for overheads, we compare Ring ORAM to both Path ORAM and the SSS ORAM (with all parameters set) in Table 3.2.

3.4 Overview of Techniques

We now explain our key technical insights. At a high level, our scheme also follows the tree-based ORAM paradigm [46]. Server storage is a binary tree where each node (a bucket) contains up to Z blocks and blocks percolate down the tree during ORAM evictions. We introduce the following non-trivial techniques that allow us to achieve significant savings in both online and overall bandwidth costs.

Eliminating online bandwidth's dependence on bucket size. In Path ORAM, reading a block would amount to reading and writing all Z slots in all buckets on a path. Our first goal is to *read only one block from each bucket* on the path. To do this, we randomly permute each bucket and store the permutation in each bucket as additional metadata. Then, by reading only metadata, the client can determine whether the requested block is in the present bucket or not. If so, the client relies on the stored permutation to read the block of interest from its random offset. Otherwise, the client reads a "fresh" (unread) dummy block, also from a random offset. We stress that the metadata size is typically much smaller than the block size, so the cost of reading metadata can be ignored.

For the above approach to be secure, it is imperative that each block in a bucket should be read at most once—a key idea also adopted by Goldreich and Ostrovsky in their early

ORAM constructions [13]. Notice that any real block is naturally read only once, since once a real block is read, it will be invalidated from the present bucket, and relocated somewhere else in the ORAM tree. But dummy blocks in a bucket can be exhausted if the bucket is read many times. When this happens (which is public information), Ring ORAM introduces an *early reshuffle* procedure to reshuffle the buckets that have been read too many times. Specifically, suppose that each bucket is guaranteed to have S dummy blocks, then a bucket must be reshuffled every S times it is read.

We note that the above technique also gives an additional nice property: out of the $O(\log N)$ blocks the client reads, only 1 of them is a real block (i.e., the block of interest); all the others are dummy blocks. If we allow some simple computation on the memory side, we can immediately apply the XOR trick from Burst ORAM [73] to get $O(1)$ online bandwidth. In the XOR trick, the server simply XORs these encrypted blocks and sends a single, XOR'ed block to the client. The client can reconstruct the ciphertext of all the dummy blocks, and XOR them away to get back the encrypted real block.

Eliminating overall bandwidth's dependence on bucket size. Unfortunately, naïvely applying the above strategy will dramatically increase offline and overall bandwidth. The more dummy slots we reserve in each bucket (i.e., a large S), the more expensive ORAM evictions become, since they have to read and write all the blocks in a bucket. But if we reserve too few dummy slots, we will frequently run out of dummy blocks and have to call *early reshuffle*, also increasing overall bandwidth.

We solve the above problem with several additional techniques. First, we design a new eviction procedure that improves eviction quality. At a high level, Ring ORAM performs evictions on a path in a similar fashion as Path ORAM, but eviction paths are selected based on a reverse lexicographical order [61], which evenly spreads eviction paths over the entire tree. The improved eviction quality allows us to perform evictions less frequently, only once every A ORAM accesses, where A is a new parameter. We then develop a proof that crucially shows A can approach $2Z$ while still ensuring negligible ORAM failure probability. The proof may be of independent interest as it uses novel proof techniques and is significantly simpler than Path ORAM's proof. The **amortized offline bandwidth** is now roughly $\frac{2Z}{A} \log N$, which **does not depend on the bucket size Z either**.

Second, bucket reshuffles can naturally piggyback on ORAM evictions. The balanced eviction order further ensures that every bucket will be reshuffled regularly. Therefore, we can set the reserved dummy slots S in accordance with the eviction frequency A, such that early reshuffles contribute little ($< 3\%$) to the overall bandwidth.

Putting it all Together. None of the aforementioned ideas would work alone. Our final product, Ring ORAM, stems from intricately combining these ideas in a non-trivial manner. For example, observe how our two main techniques act like two sides of a lever: (1) permuted buckets such that only 1 block is read per bucket; and (2) high quality and hence less frequent evictions. While permuted buckets make reads cheaper, they require adding dummy slots and would dramatically increase eviction overhead without the second technique. At the same time, less frequent evictions require increasing bucket size Z; without permuted buckets, ORAM reads blow up and nullify any saving on evictions. Additional techniques are needed to complete the construction. For example, early reshuffles keep the number of dummy slots small; piggyback reshuffles and load-balancing evictions keep the early reshuffle rate low. Without all of the above techniques, one can hardly get any

improvement.

3.5 Ring ORAM Protocol

3.5.1 The Basics

We first describe Ring ORAM in terms of its server and client data structures.

Server storage (like Path ORAM) is organized as a binary tree of buckets where each bucket has a small number of slots to hold blocks. Levels in the tree are numbered from 0 (the root) to L (inclusive, the leaves) where $L = O(\log N)$ and N is the number of blocks in the ORAM. Each bucket has $Z + S$ slots and a small amount of metadata. Of these slots, up to Z slots may contain real blocks and the remaining S slots are reserved for dummy blocks as described in Section 3.4. Our theoretical analysis in Section 3.6 will show that to store N blocks in Ring ORAM, the physical ORAM tree needs roughly $6N$ to $8N$ slots. Experiments show that server storage in practice for both Ring ORAM and Path ORAM can be $2N$ or even smaller.

Client storage is made up of a position map and a stash. The position map is a dictionary that maps each block in the ORAM to a random path in the ORAM tree (each path is uniquely identified by the path's leaf node). The stash buffers blocks that have not been evicted to the ORAM tree and additionally stores $Z(L + 1)$ blocks on the eviction path during an eviction operation. We will prove in Section 3.6 that stash overflow probability decreases exponentially as stash capacity increases, which means our required stash size is the same as Path ORAM. The position map stores $N * L$ bits, but can be squashed to constant storage using the standard recursion technique [46].

Main invariants. Ring ORAM has two main invariants:

1. (Same as Path ORAM): Every block is mapped to a path chosen uniformly at random in the ORAM tree. If a block a is mapped to path l, block a is contained either in the stash or in some bucket along the path from the root of the tree to leaf l.
2. **(Permuted buckets)** For every bucket in the tree, the physical positions of the $Z+S$ dummy and real blocks in each bucket are randomly permuted with respect to all past and future writes to that bucket.

Since a leaf uniquely determines a path in a binary tree, we will use leaves/paths interchangeably when the context is clear, and denote path l as $\mathcal{P}(l)$.

Access and Eviction Operations. The Ring ORAM access protocol is shown in Algorithm 1. Each access is broken into the following four steps:

1.) Position Map lookup (Lines 3-5): Look up the position map to learn which path l the block being accessed is currently mapped to. Remap that block to a new random path l'.

This first step is identical to other tree-based ORAMs [46, 71]. But the rest of the protocol differs substantially from previous tree-based schemes, and we highlight our key innovations in **red**.

Algorithm 1 Non-recursive Ring ORAM.

```
1:  function Access(a, op, data′)
2:      Global/persistent variables: round
3:      l′ ← UniformRandom(0, 2^L − 1)
4:      l ← PositionMap[a]
5:      PositionMap[a] ← l′
6:      data ← ReadPath(l, a)
7:      if data = ⊥ then
8:          ▷ If block a is not found on path l, it must
9:          be in Stash ◁
10:         data ← read and remove a from Stash
11:     end if
12:     if op = read then
13:         return data to client
14:     end if
15:     if op = write then
16:         data ← data′
17:     end if
18:     Stash ← Stash ∪ (a, l′, data)
19:     round ← round + 1 mod A
20:     if round ≟ 0 then
21:         EvictPath()
22:     end if
23:     EarlyReshuffle(l)
24: end function
```

2.) Read Path (Lines 6-18): The ReadPath(l, a) operation reads all buckets along $\mathcal{P}(l)$ to look for the block of interest (block a), and then reads that block into the stash. The block of interest is then updated in stash on a write, or is returned to the client on a read. We remind readers again that both reading and writing a data block are served by a ReadPath operation.

Unlike prior tree-based schemes, our ReadPath operation **only reads one block from each bucket—the block of interest if found or a previously-unread dummy block otherwise.** This is safe because of Invariant 2, above: each bucket is permuted randomly, so the slot being read looks random to an observer. This lowers the bandwidth overhead of ReadPath (i.e., online bandwidth) to $L+1$ blocks (the number of levels in the tree) or even a single block if the XOR trick is applied (Section 3.5.2).

3.) Evict Path (Line 19-22): The EvictPath operation reads Z blocks (all the remaining real blocks, and potentially some dummy blocks) from each bucket along a path into the stash, and then fills that path with blocks from the stash, trying to push blocks as far down towards the leaves as possible. The sole purpose of an eviction operation is to push blocks back to the ORAM tree to keep the stash occupancy low.

Unlike Path ORAM, eviction in Ring ORAM **selects paths in the reverse lexicographical order, and does not happen on every access [61]. Its rate is controlled by a public parameter A: every A ReadPath operations trigger a single EvictPath operation.** This means Ring ORAM needs much fewer eviction operations than Path ORAM. We will theoretically derive a tight relationship between A and Z in Section 3.6.

4.) Early Reshuffles (Line 23): Finally, we perform **a maintenance task called EarlyReshuffle on $\mathcal{P}(l)$, the path accessed by ReadPath.** This step is crucial in maintaining blocks randomly shuffled in each bucket, which enables ReadPath to securely read only one block from each bucket.

We will present details of ReadPath, EvictPath and EarlyReshuffle in the next three subsections. We defer low-level details for helper functions needed in these three subroutines to Section 3.10. We explain the security for each subroutine in Section 3.5.5. Finally, we discuss additional optimizations in Section 3.5.6 and recursion in Section 3.5.7.

3.5.2 Read Path Operation

Algorithm 2 ReadPath procedure.

```
1:  function ReadPath(l, a)
2:      data ← ⊥
3:      for i ← 0 to L do
4:          offset ← GetBlockOffset(P(l, i), a)
5:          data′ ← P(l, i, offset)
6:          Invalidate P(l, i, offset)
7:          if data′ ≠ ⊥ then
8:              data ← data′
9:          end if
10:         P(l, i).count ← P(l, i).count + 1
11:     end for
        return data
12: end function
```

The ReadPath operation is shown in Algorithm 2. For each bucket along the current path, ReadPath selects a *single* block to read from that bucket. For a given bucket, if the block of interest lives in that bucket, we read and invalidate the block of interest. Otherwise, we read and invalidate a randomly-chosen dummy block that is still valid at that point. The index of the block to read (either real or random) is returned by the GetBlockOffset function whose detailed description is given in Section 3.10.

Reading a single block per bucket is crucial for our bandwidth improvements. In addition to reducing online bandwidth by a factor of Z, it allows us to use larger Z and A to decrease overall bandwidth (Section 3.7). Without this, read bandwidth is proportional to Z, and the cost of larger Z on reads outweighs the benefits.

Bucket Metadata. Because the position map only tracks the *path* containing the block of interest, the client does not know where in each bucket to look for the block of interest. Thus, for each bucket we must store the permutation in the bucket metadata that maps each real block in the bucket to one of the $Z + S$ slots (Lines 4, GetBlockOffset) as well

as some additional metadata. Once we know the offset into the bucket, Line 5 reads the block in the slot, and invalidates it. We describe all metadata in Section 3.10, but make the important point that the metadata is small and independent of the block size.

One important piece of metadata to mention now is a counter which tracks how many times it has been read since its last eviction (Line 10). If a bucket is read too many (S) times, it may run out of dummy blocks (i.e., all the dummy blocks have been invalidated). On future accesses, if additional dummy blocks are requested from this bucket, we cannot re-read a previously invalidated dummy block: doing so reveals to the adversary that the block of interest is not in this bucket. Therefore, we need to reshuffle single buckets on-demand as soon as they are touched more than S times using EarlyReshuffle (Section 3.5.4).

XOR Technique. We further make the following key observation: during our ReadPath operation, each block returned to the client is a dummy block except for the block of interest. This means our scheme can also take advantage of the XOR technique introduced in [73] to reduce online bandwidth overhead to $O(1)$. To be more concrete, on each access ReadPath returns $L+1$ blocks in ciphertext, one from each bucket, $\mathsf{Enc}(b_0, r_0), \mathsf{Enc}(b_2, r_2), \cdots, \mathsf{Enc}(b_L, r_L)$. Enc is a randomized symmetric scheme such as AES counter mode with nonce r_i. With the XOR technique, ReadPath will return *a single ciphertext* — the ciphertext of all the blocks XORed together, namely $\mathsf{Enc}(b_0, r_0) \oplus \mathsf{Enc}(b_2, r_2) \oplus \cdots \oplus \mathsf{Enc}(b_L, r_L)$. The client can recover the encrypted block of interest by XORing the returned ciphertext with the encryptions of all the dummy blocks. To make computing each dummy block's encryption easy, the client can set the plaintext of all dummy blocks to a fixed value of its choosing (e.g., 0).

3.5.3 Evict Path Operation

Algorithm 3 EvictPath procedure.

```
1:  function EvictPath
2:      Global/persistent variables G initialized to 0
3:      l ← G mod 2^L
4:      G ← G + 1
5:      for i ← 0 to L do
6:          Stash ← Stash ∪ ReadBucket(P(l, i))
7:      end for
8:      for i ← L to 0 do
9:          WriteBucket(P(l, i), Stash)
10:         P(l, i).count ← 0
11:     end for
12: end function
```

The EvictPath routine is shown in Algorithm 3. As mentioned, evictions are scheduled statically: one eviction operation happens after every A reads. At a high level, an eviction operation reads all remaining real blocks on a path (in a secure fashion), and tries to push them down that path as far as possible. The leaf-to-root order in the writeback step (Lines 8) reflects that we wish to fill the deepest buckets as fully as possible. (For readers who are familiar with Path ORAM, EvictPath is like a Path ORAM access where no block is accessed and therefore no block is remapped to a new path.)

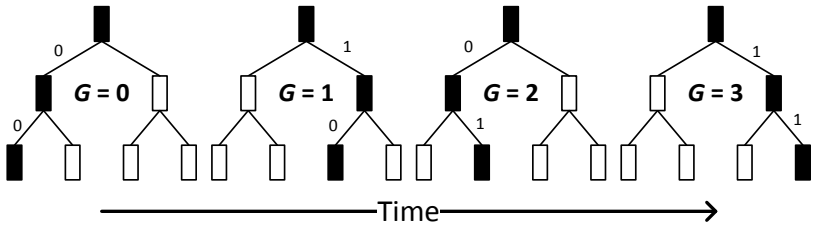


Figure 3-2: The reverse lexicographical eviction order used by EvictPath. **Black** buckets indicate those on each eviction path and G is the eviction count. The eviction paths corresponding to $G = 4$ and $G = 0$ are equal: the exact eviction sequence shown above cycles forever. We mark the eviction path edges as 0/1 (goto left child = 0, right child = 1) to illustrate that the eviction path equals G in reverse binary representation.

We emphasize two unique features of Ring ORAM eviction operations. First, evictions in Ring ORAM are performed to paths in a specific order called the *reverse-lexicographic order*, first proposed by Gentry et al. [61] and shown in Figure 3-2. The reverse-lexicographic order eviction aims to minimize the overlap between consecutive eviction paths, because (intuitively) evictions to the same bucket in consecutive accesses are less useful. This improves eviction quality and allows us to reduce the frequency of eviction. Evicting using this static order is also a key component in simplifying our theoretical analysis in Section 3.6.

Second, buckets in Ring ORAM need to be randomly shuffled (Invariant 2), and we mostly rely on EvictPath operations to keep them shuffled. An EvictPath operation reads Z blocks from each bucket on a path into the stash, and writes out $Z + S$ blocks (only up to Z are real blocks) to each bucket, randomly permuted. The details of reading/writing buckets (ReadBucket and WriteBucket) are deferred to Section 3.10.

3.5.4 Early Reshuffle Operation

Algorithm 4 EarlyReshuffle procedure.

```
1: function EarlyReshuffle(l)
2:     for i ← 0 to L do
3:         if P(l, i).count ≥ S then
4:             Stash ← Stash ∪ ReadBucket(P(l, i))
5:             WriteBucket(P(l, i), Stash)
6:             P(l, i).count ← 0
7:         end if
8:     end for
9: end function
```

Due to randomness, a bucket can be touched $> S$ times by ReadPath operations before it is reshuffled by the scheduled EvictPath. If this happens, we call EarlyReshuffle on that bucket to reshuffle it before the bucket is read again (see Section 3.5.2). More precisely, after each ORAM access EarlyReshuffle goes over all the buckets on the read path, and reshuffles all the buckets that have been accessed more than S times by performing ReadBucket and WriteBucket. ReadBucket and WriteBucket are the same as in EvictPath: that is, ReadBucket reads exactly Z slots in the bucket and WriteBucket re-permutes and writes back $Z + S$ real/dummy blocks. We note that though S does not affect security (Section 3.5.5), it

clearly has an impact on performance (how often we shuffle, the extra cost per reshuffle, etc.). We discuss how to optimally select S in Section 3.7.

3.5.5 Security Analysis

Claim 1. ReadPath *leaks no information.*

The path selected for reading will look random to any adversary due to Invariant 1 (leaves are chosen uniformly at random). From Invariant 2, we know that every bucket is randomly shuffled. Moreover, because we invalidate any block we read, we will never read the same slot. Thus, any sequence of reads (real or dummy) to a bucket between two shuffles is indistinguishable. Thus the adversary learns nothing during ReadPath. □

Claim 2. EvictPath *leaks no information.*

The path selected for eviction is chosen statically, and is public (reverse-lexicographic order). ReadBucket always reads exactly Z blocks from random slots. WriteBucket similarly writes $Z + S$ encrypted blocks in a data-independent fashion. □

Claim 3. EarlyShuffle *leaks no information.*

To which buckets EarlyShuffle operations occur is publicly known: the adversary knows how many times a bucket has been accessed since the last EvictPath to that bucket. ReadBucket and WriteBucket are secure as per observations in Claim 2. □

The three subroutines of the Ring ORAM algorithm are the only operations that cause externally observable behaviors. Claims 1, 2, and 3 show that the subroutines are secure. We have so far assumed that path remapping and bucket permutation are truly random, which gives unconditional security. If pseudorandom numbers are used instead, we have computational security through similar arguments.

3.5.6 Other Optimizations

Minimizing roundtrips. To keep the presentation simple, we wrote the ReadPath (EvictPath) algorithms to process buckets one by one. In fact, they can be performed for all buckets on the path in parallel which reduces the number of roundtrips to 2 (one for metadata and one for data blocks).

Tree-top caching. The idea of tree-top caching [66] is simple: we can reduce the bandwidth for ReadPath and EvictPath by storing the top t (a new parameter) levels of the Ring ORAM tree at the client as an extension of the stash[1]. For a given t, the stash grows by approximately $2^t Z$ blocks.

De-amortization. We can de-amortize the expensive EvictPath operation through a period of A accesses, simply by reading/writing a small number of blocks on the eviction path after each access. After de-amortization, worst-case overall bandwidth equals average overall bandwidth.

[1]We call this optimization tree-top caching following prior work. But the word cache is a misnomer: the top t levels of the tree are *permanently* stored by the client.

3.5.7 Recursive Construction

As mentioned in Section 3.1, we follow the standard recursion idea in tree-based ORAMs [46] to achieve small client storage. We remark that for reasonably block sizes (e.g., 4 KB), recursion contributes very little to overall bandwidth (e.g., $< 5\%$ for a 1 TB ORAM) because the position map ORAMs use much smaller blocks [4].

3.6 Stash Analysis

In this section we analyze the stash occupancy for a non-recursive Ring ORAM. Following the notations in Path ORAM [71], by $\mathsf{ORAM}_L^{Z,A}$ we denote a non-recursive Ring ORAM with $L+1$ levels, bucket size Z and one eviction per A accesses. The root is at level 0 and the leaves are at level L. We define the stash occupancy $\mathsf{st}(\mathcal{S}_Z)$ to be the number of real blocks in the stash after a sequence of ORAM sequences (this notation will be further explained later). We will prove that $\Pr[\mathsf{st}(\mathcal{S}_Z) > R]$ decreases exponentially in R for certain Z and A combinations. As it turns out, the deterministic eviction pattern in Ring ORAM dramatically simplifies the proof.

We note here that the reshuffling of a bucket does not affect the occupancy of the bucket, and is thus irrelevant to the proof we present here.

3.6.1 Proof outline

The proof consists of the two steps. The first step is the same as Path ORAM, and needs Lemma 1 and Lemma 2 in the Path ORAM paper [71], which we restate in Section 3.6.2. We introduce ∞-ORAM, which has an infinite bucket size and after a post-processing step has exactly the same distribution of blocks over all buckets and the stash (Lemma 1). Lemma 2 says the stash occupancy of ∞-ORAM after post-processing is greater than R if and only if there exists a subtree T in ∞-ORAM whose "occupancy" exceeds its "capacity" by more than R.

The second step (Section 3.6.3) is much simpler than the rest of Path ORAM's proof, thanks to Ring ORAM's static eviction pattern. We simply need to calculate the expected occupancy of subtrees in ∞-ORAM, and apply a Chernoff-like bound on their actual occupancy to complete the proof. We do not need the complicated eviction game, negative association, stochastic dominance, etc., as in the Path ORAM proof [4].

3.6.2 ∞-ORAM

We first introduce ∞-ORAM, denoted as $\mathsf{ORAM}_L^{\infty,A}$. Its buckets have infinite capacity. It receives the same input request sequence as $\mathsf{ORAM}_L^{Z,A}$. We then label buckets linearly such that the two children of bucket b_i are b_{2i} and b_{2i+1}, with the root bucket being b_1. We define the stash to be b_0. We refer to b_i of $\mathsf{ORAM}_L^{\infty,A}$ as b_i^∞, and b_i of $\mathsf{ORAM}_L^{Z,A}$ as b_i^Z. We further define ORAM *state*, which consists of the states of all the buckets in the ORAM, i.e., the blocks contained by each bucket. Let $\mathcal{S}_\infty$ be the state of $\mathsf{ORAM}_L^{\infty,A}$ and $\mathcal{S}_Z$ be the state of $\mathsf{ORAM}_L^{Z,A}$.

We now propose a new greedy post-processing algorithm G (different from the one in [71]), which by reassigning blocks in buckets makes each bucket b_i^∞ in ∞-ORAM contain the same set of blocks as b_i^Z. Formally, G takes as input $\mathcal{S}_\infty$ and $\mathcal{S}_Z$ after the same access sequence with the same randomness. For i from $2^{L+1}-1$ down to 1 (note that

the decreasing order ensures that a parent is always processed later than its children), G processes the blocks in bucket b_i^∞ in the following way:

1. For those blocks that are also in b_i^Z, keep them in b_i^∞.
2. For those blocks that are not in b_i^Z but in some ancestors of b_i^Z, move them from b_i^∞ to $b_{i/2}^\infty$ (the parent of b_i^∞, and note that the division includes flooring). If such blocks exist and the number of blocks remaining in b_i^∞ is less than Z, raise an error.
3. If there exists a block in b_i^∞ that is in neither b_i^Z nor any ancestor of b_i^Z, raise an error.

We say $G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = \mathcal{S}_Z$, if no error occurs during G and b_i^∞ after G contains the same set of blocks as b_i^Z for $i = 0, 1, \cdots 2^{L+1}$.

Lemma 1. *$G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = \mathcal{S}_Z$ after the same ORAM access sequence with the same randomness.*

Next, we investigate what state $\mathcal{S}_\infty$ will lead to the stash occupancy of more than R blocks in a post-processed ∞-ORAM. We say a subtree T is a rooted subtree, denoted as $T \in \mathsf{ORAM}_L^{\infty,A}$ if T contains the root of $\mathsf{ORAM}_L^{\infty,A}$. This means that if a node in $\mathsf{ORAM}_L^{\infty,A}$ is in T, then so are all its ancestors. We define $n(T)$ to be the total number of nodes in T. We define $c(T)$ (the capacity of T) to be the maximum number of blocks T can hold; for Ring ORAM $c(T) = n(T) \cdot Z$. Lastly, we define $X(T)$ (the occupancy of T) to be the actual number of real blocks that are stored in T. The following lemma characterizes the stash size of a post-processed ∞-ORAM:

Lemma 2. *$\mathsf{st}(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)) > R$ if and only if $\exists T \in \mathsf{ORAM}_L^{\infty,A}$ s.t. $X(T) > c(T) + R$ before post-processing.*

By Lemma 1 and Lemma 2, we have

$$\begin{aligned}
&\Pr[\mathsf{st}(\mathcal{S}_Z) > R] = \Pr[\mathsf{st}(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)) > R] \\
&\leq \sum_{T \in \mathsf{ORAM}_L^{\infty,A}} \Pr[X(T) > c(T) + R] \\
&< \sum_{n \geq 1} 4^n \max_{T: n(T) = n} \Pr[X(T) > c(T) + R] \qquad (3.1)
\end{aligned}$$

The above inequalities used a union bound and a bound on Catalan sequences.

3.6.3 Bounding the Stash Size

We first give a bound on the expected bucket load:

Lemma 3. *For any rooted subtree T in $\mathsf{ORAM}_L^{\infty,A}$, if the number of distinct blocks in the ORAM $N \leq A \cdot 2^{L-1}$, the expected load of T has the following upper bound:*

$$\forall T \in \mathsf{ORAM}_L^{\infty,A}, E[X(T)] \leq n(T) \cdot A/2.$$

Let $X(T) = \sum_i X_i(T)$, where each $X_i(T) \in \{0, 1\}$ and indicates whether the i-th block (can be either real or stale) is in T. Let $p_i = \Pr[X_i(T) = 1]$. $X_i(T)$ is completely determined by its time stamp i and the leaf label assigned to block i, so they are independent from

each other (refer to the proof of Lemma 4). Thus, we can apply a Chernoff-like bound to get an exponentially decreasing bound on the tail distribution. To do so, we first establish a bound on $E\left[e^{tX(T)}\right]$ where $t > 0$,

$$\begin{aligned} E\left[e^{tX(T)}\right] &= E\left[e^{t\sum_i X_i(T)}\right] = E\left[\Pi_i e^{tX_i(T)}\right] \\ &= \Pi_i E\left[e^{tX_i(T)}\right] \qquad \text{(by independence)} \\ &= \Pi_i \left(p_i(e^t - 1) + 1\right) \\ &\leq \Pi_i \left(e^{p_i(e^t-1)}\right) = e^{(e^t-1)\sum_i p_i} \\ &= e^{(e^t-1)E[X(T)]} \end{aligned} \tag{3.2}$$

For simplicity, we write $n = n(T)$ and $a = A/2$. By Lemma 4, $E[X(T)] \leq n \cdot a$. By the Markov Inequality, we have for all $t > 0$,

$$\begin{aligned} \Pr\left[X(T) > c(T) + R\right] &= \Pr\left[e^{tX(T)} > e^{t(nZ+R)}\right] \\ &\leq E\left[e^{tX(T)}\right] \cdot e^{-t(nZ+R)} \\ &\leq e^{(e^t-1)an} \cdot e^{-t(nZ+R)} \\ &= e^{-tR} \cdot e^{-n\left[tZ - a(e^t-1)\right]} \end{aligned}$$

Let $t = \ln(Z/a)$,

$$\Pr\left[X(T) > c(T) + R\right] \leq (a/Z)^R \cdot e^{-n[Z\ln(Z/a)+a-Z]} \tag{3.3}$$

Now we will choose Z and A such that $Z > a$ and $q = Z\ln(Z/a) + a - Z - \ln 4 > 0$. If these two conditions hold, from Equation (3.1) we have $t = \ln(Z/a) > 0$ and that the stash overflow probability decreases exponentially in the stash size R:

$$\Pr\left[\mathsf{st}\left(\mathcal{S}_Z\right) > R\right] \leq \sum_{n\geq 1} (a/Z)^R \cdot e^{-qn} < \frac{(a/Z)^R}{1 - e^{-q}}.$$

3.6.4 Stash Size in Practice

Now that we have established that $Z\ln(2Z/A) + A/2 - Z - \ln 4 > 0$ ensures an exponentially decreasing stash overflow probability, we would like to know how tight this requirement is and what the stash size should be in practice.

We simulate Ring ORAM with $L = 20$ for over 1 billion accesses in a random access pattern, and measure the maximum stash occupancy T (excluding the transient storage of a path). For several Z values, we look for the maximum A that results in an exponentially decreasing stash overflow probability. In Figure 3-3, we plot both the empirical curve based on simulation and the theoretical curve based on the proof. In all cases, the theoretical curve indicates an only slightly smaller A than we are able to achieve in simulation, indicating that our analysis is tight.

To determine required stash size in practice, Table 3.3 shows the extrapolated required stash size for a stash overflow probability of $2^{-\lambda}$ for several realistic λ. We show $Z = 16$, $A = 23$ for completeness: this is an aggressive setting that works for $Z = 16$ according

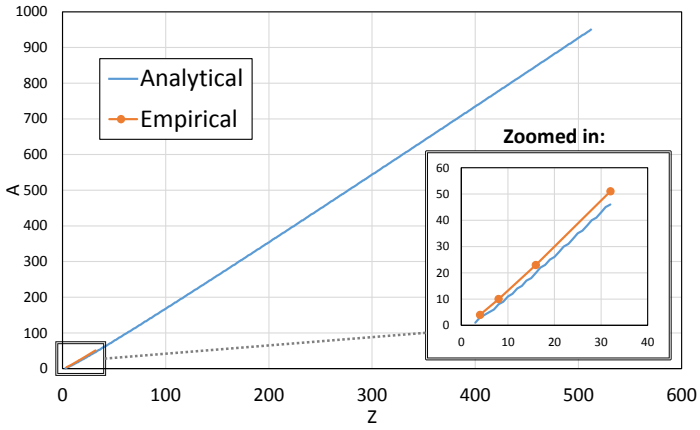


Figure 3-3: For each Z, determine analytically and empirically the maximum A that results in an exponentially decreasing stash failure probability.

Table 3.3: Maximum stash occupancy for realistic security parameters (stash overflow probability $2^{-\lambda}$) and several choices of A and Z. $A = 23$ is the maximum achievable A for $Z = 16$ according to simulation.

		Z,A Parameters				
		4,3	8,8	16,20	32,46	16,23
		Max Stash Size T				
λ	80	32	41	65	113	197
	128	51	62	93	155	302
	256	103	120	171	272	595

to simulation but does not satisfy the theoretical analysis; observe that this point requires roughly $3\times$ the stash occupancy for a given λ.

3.7 Bandwidth Analysis

In this section, we answer an important question: how do Z (the maximum number of real blocks per bucket), A (the eviction rate) and S (the number of extra dummies per bucket) impact Ring ORAM's performance (bandwidth)? By the end of the section, we will have a theoretically-backed analytic model that, given Z, selects optimal A and S to minimize bandwidth.

We first state an intuitive trade-off: for a given Z, increasing A causes stash occupancy to increase and bandwidth overhead to decrease. Let us first ignore early reshuffles and the XOR technique. Then, the overall bandwidth of Ring ORAM consists of ReadPath and EvictPath. ReadPath transfers $L+1$ blocks, one from each bucket. EvictPath reads Z blocks per bucket and writes $Z+S$ blocks per bucket, $(2Z+S)(L+1)$ blocks in total, but happens every A accesses. From the requirement of Lemma 4, we have $L = \log(2N/A)$, so the ideal amortized overall bandwidth of Ring ORAM is $(1 + (2Z + S)/A)\log(4N/A)$. Clearly, a larger A improves bandwidth for a given Z as it reduces both eviction frequency and tree depth L. So we simply choose the largest A that satisfies the requirement from the stash analysis in Section 3.6.3.

Now we consider the extra overhead from early reshuffles. We have the following trade-off

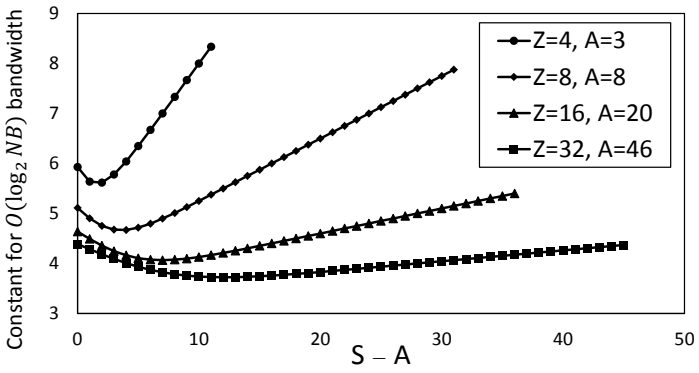


Figure 3-4: For different Z, and the corresponding optimal A, vary S and plot bandwidth overhead. We only consider $S \geq A$.

Table 3.4: Analytic model for choosing parameters, given Z.

Find largest $A \leq 2Z$ such that $Z \ln(2Z/A) + A/2 - Z - \ln 4 > 0$ holds.
Find $S \geq 0$ that minimizes $(2Z + S)(1 + \mathsf{Poiss\_cdf}(S, A))$.
Ring ORAM offline bandwidth is $\frac{(2Z+S)(1+\mathsf{Poiss\_cdf}(S,A))}{A} \cdot \log(4N/A)$.

in choosing S: as S increases, the early reshuffle rate decreases (since we have more dummies per bucket) but the cost to read+write buckets during an EvictPath and EarlyReshuffle increases. This effect is shown in Figure 3-4 through simulation: for S too small, early shuffle rate is high and bandwidth increases; for S too large, eviction bandwidth dominates.

To analytically choose a good S, we analyze the early reshuffle rate. First, notice a bucket at level l in the Ring ORAM tree will be processed by EvictPath *exactly* once for every $2^l A$ ReadPath operations, due to the reverse-lexicographic order of eviction paths (Section 3.5.3). Second, each ReadPath operation is to an independent and uniformly random path and thus will touch any bucket in level l with equal probability of 2^{-l}. Thus, the distribution on the expected number of times ReadPath operations touch a given bucket in level l, between two consecutive EvictPath calls, is given by a binomial distribution of $2^l A$ trials and success probability 2^{-l}. The probability that a bucket needs to be early reshuffled before an EvictPath is given by a binomial distribution cumulative density function $\mathsf{Binom\_cdf}(S, 2^l A, 2^{-l})$.[2] Based on this analysis, the expected number of times any bucket is involved in ReadPath operations between consecutive EvictPath operations is A. Thus, we will only consider $S \geq A$ as shown in Figure 3-4 ($S < A$ is clearly bad as it needs too much early reshuffling).

We remark that the binomial distribution quickly converges to a Poisson distribution. So the amortized overall bandwidth, taking early reshuffles into account, can be accurately approximated as $(L+1) + (L+1)(2Z+S)/A \cdot (1 + \mathsf{Poiss\_cdf}(S, A))$. We should then choose the S that minimizes the above formula. This method always finds the optimal S and perfectly matches the overall bandwidth in our simulation in Figure 3-4.

[2]The possibility that a bucket needs to be early reshuffled twice before an eviction is negligible.

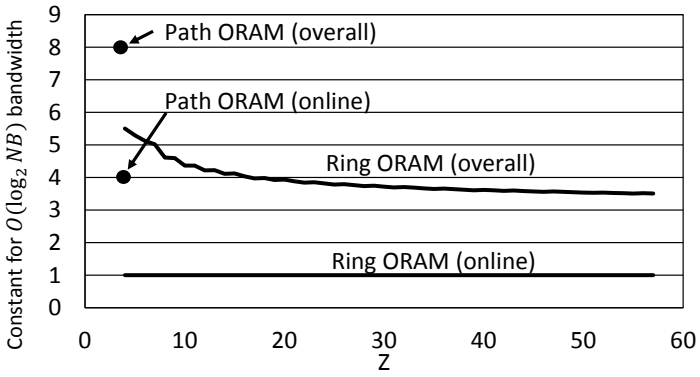


Figure 3-5: Overall bandwidth as a function of Z. Kinks are present in the graph because we always round A to the nearest integer. For Path ORAM, we only study $Z = 4$ since a larger Z strictly hurts bandwidth.

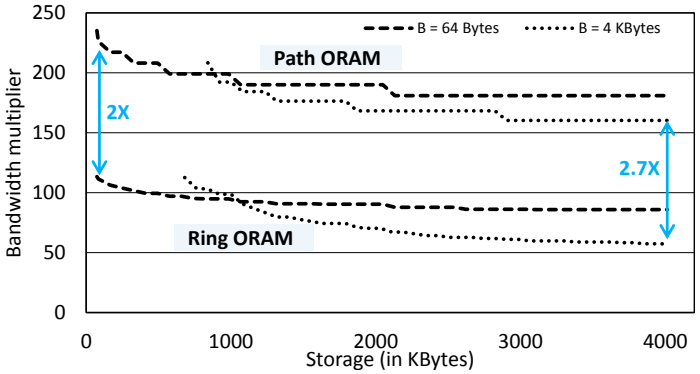


Figure 3-6: Bandwidth overhead vs. data block storage for 1 TByte ORAM capacities and ORAM failure probability 2^{-80}.

We recap how to choose A and S for a given Z in Table 3.4. For the rest of the chapter, we will choose A and S this way unless otherwise stated. Using this method to set A and S, we show online and overall bandwidth as a function of Z in Figure 3-5. In the figure, Ring ORAM does not use the XOR technique on reads. For $Z = 50$, we achieve $\sim 3.5 \log N$ bandwidth; for very large Z, bandwidth approaches $3 \log N$. Applying the XOR technique, online bandwidth overhead drops to close to 1 which reduces overall bandwidth to $\sim 2.5 \log N$ for $Z = 50$ and $2 \log N$ for very large Z.

3.8 Evaluation

3.8.1 Bandwidth vs. Client Storage

To give a holistic comparison between schemes, Figure 3-6 shows the best achievable bandwidth, for different client storage budgets, for Path ORAM and Ring ORAM. For each scheme in the figure, we apply all known optimizations and tune parameters to minimize overall bandwidth given a storage budget. For Path ORAM we choose $Z = 4$ (increasing Z strictly hurts bandwidth) and tree-top cache to fill remaining space. For Ring ORAM we adjust Z, A and S, tree-top cache and apply the XOR technique. ORAM capacity is fixed to 1 TByte.

Table 3.5: Breakdown between online and offline bandwidth given a client storage budget of 1000× the block size for several representative points (Section 3.8.1). Overheads are relative to an insecure system.

		Online, Overall Bandwidth overhead		
Block Size (Bytes)	Z, A (Ring ORAM only)	**Ring ORAM**	**Ring ORAM (XOR)**	Path ORAM
64	$10, 11$	48×, 144×	24×, 118×	120×, 240×
4096	$33, 48$	20×, 82×	$\sim$ 1×, 60×	80×, 160×

To simplify the presentation, "client storage" includes all ORAM data structures except for the position map – which has the same space/bandwidth cost for both Path ORAM and Ring ORAM. We remark that applying the recursion technique (Section 3.5.7) to get a small on-chip position map is cheap for reasonably large blocks. For example, recursing the on-chip position map down to 256 KBytes of space when the data block size is 4 KBytes increases overall bandwidth for Ring ORAM and Path ORAM by $< 3\%$.

The high order bit is that across different block sizes and client storage budgets, Ring ORAM consistently reduces overall bandwidth relative to Path ORAM by 2-2.7×. Hidden in Figure 3-6 is that when block size is small, online/overall bandwidth for Ring ORAM increases (see Table 3.5 for two representative points). The reason is that with small blocks, the cost to read bucket metadata cannot be ignored, forcing Ring ORAM to use smaller Z.

3.9 Ring ORAM with Large Client Storage

If given a large client storage budget, we can first choose very large A and Z for Ring ORAM, which means bandwidth approaches $2 \log N$ (Section 3.7).[3] Then, remaining client storage can be used to tree-top cache (Section 3.5.6). For example, tree-top caching $t = L/2$ levels requires $O(\sqrt{N})$ storage and bandwidth drops by a factor of 2 to $1 \cdot \log N$—which roughly matches the SSS construction [56].

Burst ORAM [73] extends the SSS construction to handle millions of accesses in a short period, followed by a relatively long idle time where there are few requests. The idea to adapt Ring ORAM to handle bursts is to delay multiple (potentially millions of) EvictPath operations until after the burst of requests. Unfortunately, this strategy means we will experience a much higher early reshuffle rate in levels towards the root. The solution is to coordinate tree-top caching with delayed evictions: For a given tree-top size t, we allow at most 2^t delayed EvictPath operations. This ensures that for levels $\geq t$, the early reshuffle rate matches our analysis in Section 3.7. We experimentally compared this methodology to the dataset used by Burst ORAM and verified that it gives comparable performance to that work.

3.10 Bucket Structure (Reference)

Table 3.6 lists all the fields in a Ring ORAM bucket and their size. We would like to make two remarks. First, only the data fields are permuted and that permutation is stored in ptrs. Other bucket fields do not need to be permuted because when they are needed, they will be read in their entirety. Second, count and valids are stored in plaintext. There is no

[3] We assume the XOR technique because large client storage implies a file server setting.

Table 3.6: Ring ORAM bucket format. All logs are taken to their ceiling.

Notation	Size (bits)	Meaning
count	$\log(S)$	# of times the bucket has been touched by ReadPath since it was last shuffled
valids	$(Z+S)*1$	Indicates whether each of the $Z+S$ blocks is valid
addrs	$Z*\log(N)$	Address for each of the Z (potentially) real blocks
leaves	$Z*L$	Leaf/path label for each of the Z (potentially) real blocks
ptrs	$Z*\log(Z+S)$	Offset in the bucket for each of the Z (potentially) real blocks
data	$(Z+S)*B$	Data field for each of the $Z+S$ blocks, **permuted** according to ptrs
IV	λ (security parameter)	Encryption seed for the bucket; count and valids are stored in the clear

need to encrypt them since the server can see which bucket is accessed (deducing count for each bucket), and which slot is accessed in each bucket (deducing valids for each bucket). In fact, if the server can do computation and is trusted to follow the protocol faithfully, the client can let the server update count and valids. All the other structures should be probabilistically encrypted.

Having defined the bucket structure, we can be more specific about some of the operations in earlier sections. For example, in Algorithm 2 Line 5 means reading $\mathcal{P}(l,i)$.data[offset], and Line 6 means setting $\mathcal{P}(l,i)$.valids[offset] to 0.

Now, we describe the helper functions in detail. GetBlockOffset reads in the valids, addrs, ptrs fields, and looks for the block of interest. If it finds the block of interest, meaning that the address of a still valid block matches the block of interest, it returns the permuted location of that block (stored in ptrs). If it does not find the block of interest, it returns the permuted location of a random valid dummy block.

ReadBucket reads all of the remaining real blocks in a bucket into the stash. For security reasons, ReadBucket always reads *exactly* Z blocks from that bucket. If the bucket contains less than Z valid real blocks, the remaining blocks read out are random valid dummy blocks. Importantly, since we allow at most S reads to each bucket before reshuffling it, it is guaranteed that there are at least Z valid (real + dummy) blocks left that have not been touched since the last reshuffle.

WriteBucket evicts as many blocks as possible (up to Z) from the stash to a certain bucket. If there are $z' \leq Z$ real blocks to be evicted to that bucket, $Z+S-z'$ dummy blocks are added. The $Z+S$ blocks are then randomly shuffled based on either a truly random permutation or a Pseudo Random Permutation (PRP). The permutation is stored in the bucket field ptrs. Then, the function resets count to 0 and all valid bits to 1, since this bucket has just been reshuffled and no blocks have been touched. Finally, the permuted data field along with its metadata are encrypted (except count and valids) and written out to the bucket.

Algorithm 5 Helper functions.

count, valids, addrs, leaves, ptrs, data are fields of the input bucket in each of the following three functions

```
1: function GetBlockOffset(bucket, a)
2:     read in valids, addrs, ptrs
3:     decrypt addrs, ptrs
4:     for j ← 0 to Z − 1 do
5:         if a = addrs[j] and valids[ptrs[j]] then
6:             return ptrs[j]                                  ▷ block of interest
7:         end if
8:     end forreturn a pointer to a random valid dummy
9: end function
```

```
1: function ReadBucket(bucket)
2:     read in valids, addrs, leaves, ptrs
3:     decrypt addrs, leaves, ptrs
4:     z ← 0                                  ▷ track # of remaining real blocks
5:     for j ← 0 to Z − 1 do
6:         if valids[ptrs[j]] then
7:             data′ ← read and decrypt data[ptrs[j]]
8:             z ← z + 1
9:             if addrs[j] ≠ ⊥ then
10:                block ← (addr[j], leaf[j], data′)
11:                Stash ← Stash ∪ block
12:            end if
13:        end if
14:    end for
15:    for j ← z to Z − 1 do
16:        read a random valid dummy
17:    end for
18: end function
```

```
1: function WriteBucket(bucket, Stash)
2:     find up to Z blocks from Stash that can reside
3:     in this bucket, to form addrs, leaves, data′
4:     ptrs ← PRP(0, Z + S)                          ▷ or truly random
5:     for j ← 0 to Z − 1 do
6:         data[ptrs[j]] ← data′[j]
7:     end for
8:     valids ← {1}^{Z+S}
9:     count ← 0
10:    encrypt addrs, leaves, ptrs, data
11:    write out count, valids, addrs, leaves, ptrs, data
12: end function
```

Chapter 4

Onion ORAM:

A Constant Bandwidth Blowup ORAM

This chapter presents Onion ORAM, the first ORAM with constant worst-case bandwidth blowup under standard cryptographic assumptions. Onion ORAM leverages poly-logarithmic server computation to circumvent the logarithmic lower bound on ORAM bandwidth blowup. Unlike prior work, Onion ORAM does not require fully homomorphic encryption, but requires only certain additively homomorphic encryption schemes. At the core of the construction is an ORAM scheme that has "shallow circuit depth" over the entire history of ORAM accesses (which we also refer to as "bounded feedback"). Onion ORAM utilizes novel techniques to achieve security against a malicious server, without resorting to expensive and non-standard techniques such as SNARKs.

Starting with the work of Goldreich and Ostrovsky [9, 10, 13], the ORAM literature has implicitly assumed that the server acts as a simple storage device that allows the client to read and write data to it, but does not perform any computation otherwise. However, in many scenarios investigated by subsequent works [58, 70, 73, 79] (e.g., the setting of remote oblivious file servers), the untrusted server has significant computational power, possibly even much greater than that of the client. Therefore, it is natural to extend the ORAM model to allow for server computation, and to distinguish between the amount of computation performed by the server and the amount of communication with the client.

Indeed, many recent ORAM schemes have implicitly or explicitly leveraged some amount of server computation to either reduce bandwidth cost [56, 64, 84, 81, 79, 61, 3, 72] (also, see Chapter 3), or reduce the number of online roundtrips [57]. We remark that some prior works [79, 72] call themselves oblivious storage (or oblivious outsourced storage) to distinguish from the standard ORAM model where there is no server computation. We will simply apply the term ORAM to both models, and refer to ORAM *with/without server computation* to distinguish between the two.

At first, many works implicitly used server computation in ORAM constructions [56, 81, 79, 61, 3, 57], without making a clear definitional distinction from standard ORAM. Apon et al. were the first to observe that such a distinction is warranted [72], not only for the extra rigor, but also because the definition renders the important Goldreich-Ostrovsky ORAM lower bound [13] inapplicable to the server computation setting — as we discuss below.

4.1 Attempts to "Break" the Goldreich-Ostrovsky Bound

Traditionally, ORAM constructions are evaluated by their *bandwidth*, *client storage* and *server storage*. Bandwidth is the amount of communication (in bits) between client/server to serve a client request, including the communication in the background to maintain the ORAM (i.e., ORAM evictions). We also define bandwidth blowup to be bandwidth measured in the number of blocks (i.e., blowup compared to a normal RAM). Client storage is the amount of trusted local memory required at the client side to manage the ORAM protocol and server storage is the amount of storage needed at the server to store all data blocks.

In their seminal work [13], Goldreich and Ostrovsky showed that an ORAM of N blocks must incur a $O(\log N)$ lower bound in bandwidth blowup, under $O(1)$ blocks of client storage. If we allow the server to perform computation, however, the Goldreich-Ostrovsky lower bound no longer applies with respect to client-server bandwidth [72]. The reason is that the Goldreich-Ostrovsky bound is in terms of the *number of operations* that must be performed. With server computation, though the number of operations is still subject to the bound, most operations can be performed on the server-side without client intervention, making it possible to break the bound in terms of bandwidth between client and server. Since historically bandwidth has been the most important metric and the bottleneck for ORAM, breaking the bound in terms of bandwidth constitutes a significant advance.

However, it turns out that this is not easy. Indeed, two prior works [79, 72] have made endeavors towards this direction using homomorphic encryption. Path-PIR [79] leverages additively homomorphic encryption (AHE) to improve ORAM online bandwidth, but its overall bandwidth blowup is still poly-logarithmic. On the other hand, Apon et al. [72] showed that using a fully homomorphic encryption (FHE) scheme with *constant ciphertext expansion*, one can construct an ORAM scheme with constant bandwidth blowup. The main idea is that, instead of having the client move data around on the server "manually" by reading and writing to the server, the client can instruct the server to perform ORAM request and eviction operations under an FHE scheme without revealing any data and its movement. While this is a very promising direction, it suffers from the following drawbacks:

- First, ORAM keeps access patterns private by continuously shuffling memory as data is accessed. This means the ORAM circuit depth that has to be evaluated under FHE depends on the number of ORAM accesses made and can grow unbounded (which we say to mean any polynomial amount in N). Therefore, Apon et al. [72] needs FHE bootstrapping, which not only requires circular security but also incurs a large performance penalty in practice.[1]

- Second, with the server performing homomorphic operations on encrypted data, achieving malicious security is difficult. Consequently, most existing works either only guarantee semi-honest security [79, 81], or leveraged powerful tools such as SNARKs to ensure malicious security [72]. However, SNARKs not only require non-standard assumptions [41], but also incur prohibitive cost in practice.

[1] While bootstrapping performance has been made asymptotically efficient by recent works [51], the cost in practice is still substantial, on the order of tens of seconds to minutes (amortized), whereas other homomorphic operations are on the order of milliseconds to seconds [88].

4.2 Contributions

With the above observation, the goal of this work is to construct constant bandwidth blowup ORAM schemes from *standard assumptions* that have *practical efficiency* and *verifiability in the malicious setting*. Specifically, we give proofs by construction for the following theorems. Let B be the block size in bits and N the number of blocks in the ORAM.

Theorem 1 (Semi-honest security construction)**.** *Under the Decisional Composite Residuosity assumption (DCR) or Learning With Errors (LWE) assumption, there exists an ORAM scheme with semi-honest security, $O(B)$ bandwidth, $O(BN)$ server storage and $O(B)$ client storage. To achieve negligible in N probability of ORAM failure and success from best known attacks, our schemes require poly-logarithmic in N block size and server computation.*

We use negligible in N security following prior ORAM work but will also give asymptotics needed for exact exponential security in Section 4.7.3 and Table 4.2.

Looking at the big picture, our DCR-based scheme is the first demonstration of a constant bandwidth blowup ORAM using any additively homomorphic encryption scheme (AHE), as opposed to FHE. Our LWE-based scheme is the first time ORAM has been combined with SWHE/FHE in a way that does not require Gentry's bootstrapping procedure.

Our next goal is to extend our semi-honest constructions to the malicious setting. In Section 4.6, we will introduce the concept of "abstract server computation ORAM" which both of our constructions satisfy. Then, we can achieve malicious security due to the following theorem:

Theorem 2 (Malicious security construction)**.** *With the additional assumption of collision-resistant hash functions, any "abstract server computation ORAM" scheme with semi-honest security can be compiled into a "verified server computation ORAM" scheme which has malicious security.*

We stress that these are the *only* required assumptions. We do not need the circular security common in FHE schemes and do not rely on SNARKs for malicious security.

Main ideas. The key technical contributions enabling the above results are:

- (Section 4.4) An ORAM that, when combined with server computation, has *shallow circuit depth*, i.e., $O(\log N)$ over the entire history of all ORAM accesses. This is a necessity for our constructions based on AHE or SWHE, and removes the need for FHE (Gentry's bootstrapping operations). We view this technique as an important step towards practical constant bandwidth blowup ORAM schemes.

- (Section 4.6) A novel technique that combines a cut and choose-like idea with an error-correcting code to amplify soundness.

Table 4.1 summarizes our contributions and compares our schemes with some of the state-of-the-art ORAM constructions.

Table 4.1: **Our contribution.** N is the number of blocks. The optimal block size is the data block size needed to achieve the stated bandwidth, and is measured in bits. All schemes have $O(B)$ client storage and $O(BN)$ server storage (both asymptotically optimal) and negligible failure probability in N. Computation measures the number of two-input plaintext gates evaluated per ORAM access. "M" stands for malicious security, and "SH" stands for semi-honest. We set parameters for AHE/SWHE (the Damgård-Jurik and Ring-LWE cryptosystems [21, 40], respectively) to get super-poly in N defense to best known attacks [28, 44]. At the end of the chapter, we show another version of the table (Table 4.2) where there is no assumed relation between parameters.

Scheme	Optimal Block size B	Bandwidth	Server Computation	Client Computation	Security
Circuit ORAM [5]	$\Omega(\log^2 N)$	$\omega(B \log N)$	N/A	N/A	M
Path-PIR [79]	$\omega(\log^5 N)$	$O(B \log N)$	$\widetilde{\omega}(B \log^5 N)$	$\widetilde{O}(B \log^4 N)$	SH
AHE Onion ORAM	$\widetilde{\Omega}(\log^5 N)$	$O(B)$	$\widetilde{\omega}(B \log^4 N)$	$\widetilde{O}(B \log^4 N)$	SH
	$\widetilde{\omega}(\log^6 N)$	$O(B)$	$\widetilde{\omega}(B \log^4 N)$	$\widetilde{O}(B \log^4 N)$	M
SWHE Onion ORAM	$\widetilde{\omega}(\log^2 N)$	$O(B)$	$\widetilde{\omega}(B \log^2 N)$	$\widetilde{\omega}(B)$	SH
	$\widetilde{\omega}(\log^4 N)$	$O(B)$	$\widetilde{\omega}(B \log^2 N)$	$\widetilde{\omega}(B + \log^2 N)$	M

Practical efficiency. To show how our results translate to practice, Section 4.7.4 compares our semi-honest AHE-based construction against Path PIR [79] and Circuit ORAM [5]—the best prior schemes with and without server computation that match our scheme in client/server storage. The top order bit is that as block size increases, our construction's bandwidth approaches $2B$. When all three schemes use an 8 MB block size (a proxy for modern image file size), Onion ORAM improves over Circuit ORAM and Path-PIR's bandwidth (in bits) by **35**× and **22**×, respectively. For larger block sizes, our improvement increases. We note that in many cases, block size is an application constraint: for applications asking for a large block size (e.g., image sharing), all ORAM schemes will use that block size.

4.3 Overview of Techniques

In our schemes, the client "guides" the server to perform ORAM accesses and evictions homomorphically by sending the server some "helper values". With these helper values, the server's main job will be to run a sub-routine called the "*homomorphic select*" operation (select operation for short), which can be implemented using either AHE or SWHE – resulting in the two different constructions in Section 4.5 and Section 4.9. We can achieve constant bandwidth blowup because helper value size is independent of data block size: when the block size sufficiently large, sending helper values does not affect the asymptotic bandwidth blowup. We now explain these ideas along with pitfalls and solutions in more detail. For the rest of the section, we focus on the AHE-based scheme but note that the story with SWHE is very similar.

Building block: homomorphic select operation. The select operation, which resembles techniques from private information retrieval (PIR) [28], takes as input m plaintext data blocks $\mathsf{pt}_1, \ldots, \mathsf{pt}_m$ and encrypted helper values which represent a user-chosen index i^*. The output is an encryption of block pt_{i^*}. Obviously, the helper values should not reveal i^*.

Our ORAM protocol will need select operations to be performed over the *outputs* of prior select operations. For this, we require a sequence of AHE schemes $\mathcal{E}_\ell$ with plaintext space $\mathbb{L}_\ell$ and ciphertext space $\mathbb{L}_{\ell+1}$ where $\mathbb{L}_{\ell+1}$ is again in the plaintext space of $\mathcal{E}_{\ell+1}$. Each scheme $\mathcal{E}_\ell$ is additively homomorphic meaning $\mathcal{E}_\ell(x) \oplus \mathcal{E}_\ell(y) = \mathcal{E}_\ell(x+y)$. We denote an ℓ-layer onion encryption of a message x by $\mathcal{E}^\ell(x) := \mathcal{E}_\ell(\mathcal{E}_{\ell-1}(\ldots \mathcal{E}_1(x)))$.

Suppose the inputs to a select operation are encrypted with ℓ layers of onion encryption, i.e., $\mathsf{ct}_i = \mathcal{E}^\ell(\mathsf{pt}_i)$. To select block i^*, the client sends an encrypted select vector (select vector for short), $\mathcal{E}_{\ell+1}(b_1), \ldots, \mathcal{E}_{\ell+1}(b_m)$ where $b_{i^*} = 1$ and $b_i = 0$ for all other $i \neq i^*$. Using this select vector, the server can homomorphically compute $\mathsf{ct}^* = \bigoplus_i \mathcal{E}_{\ell+1}(b_i) \cdot \mathsf{ct}_i = \mathcal{E}_{\ell+1}(\sum_i b_i \cdot \mathsf{ct}_i) = \mathcal{E}_{\ell+1}(\mathsf{ct}_{i^*}) = \mathcal{E}^{\ell+1}(\mathsf{pt}_{i^*})$. The result is the selected data block pt_{i^*}, with $\ell+1$ layers of onion encryption. Notice that the result has one more layer than the input.

All ORAM operations can be implemented using homomorphic select operations. In our schemes, for each ORAM operation, the client read/writes per-block metadata and creates a select vector(s) based on that metadata. The client then sends the encrypted select vector(s) to the server, who does the heavy work of performing actual computation over block contents.

Specifically, we will build on top of tree-based ORAMs [46, 71], a standard type of ORAM without server computation. Metadata for each block includes its logical address and the path it is mapped to. To request a data block, the client first reads the logic addresses of all blocks along the read path. After this step, the client knows which block to select and can run the homomorphic select protocol with the server. ORAM eviction operations require that the client sends encrypted select vectors to indicate how blocks should percolate down the ORAM tree. As explained above, each select operation adds an encryption layer to the selected block.

Achieving constant bandwidth blowup. To get constant bandwidth blowup, we must ensure that select vector bandwidth is smaller than the data block size. For this, we need several techniques. First, we will split each plaintext data block into C chunks $\mathsf{pt}_i = (\mathsf{pt}_i[1], \ldots, \mathsf{pt}_i[C])$, where each chunk is encrypted separately, i.e., $\mathsf{ct}_i = (\mathsf{ct}_i[1], \ldots, \mathsf{ct}_i[C])$ where $\mathsf{ct}_i[j]$ is an encryption of $\mathsf{pt}_i[j]$. Crucially, each select vector can be reused for all the C chunks. By increasing C, we can increase the data block size to decrease the relative bandwidth of select vectors.

Second, we require that each encryption layer adds a small *additive* ciphertext expansion (even a constant multiplicative expansion would be too large). Fortunately, we do have well established additively homomorphic encryption schemes that meet this requirement, such as the Damgård-Jurik cryptosystem [21]. Third, the "depth" of the homomorphic select operations has to be bounded and shallow. This requirement is the most technically challenging to satisfy, and we will now discuss it in more detail.

Bounding the select operation depth. We address this issue by constructing a new tree-based ORAM, which we call a "*bounded feedback ORAM*". By "feedback", we refer to the situation where during an eviction some block a gets stuck in its current bucket b. When this happens, an eviction into b needs select operations that take both incoming blocks and block a as input, resulting in an extra layer on bucket b (on top of the layers bucket b already has). The result is that buckets will accumulate layers (with AHE) or ciphertext noise (with SWHE) on each eviction, which grows unbounded over time.

Our bounded feedback ORAM breaks the feedback loop by guaranteeing that bucket b will be empty at public times, which allows upstream blocks to move into b without feedback from blocks already in b. It turns out that breaking this feedback is not trivial: in all existing tree-based ORAM schemes [46, 71, 5], blocks can get stuck in buckets during evictions which means there is no guarantee on when buckets are empty.[2] We remark that cutting feedback is equivalent to our claim of shallow circuit depth in Section 4.2: Without cutting feedback, the depth of the ORAM circuit keeps growing with the number of ORAM accesses.

Techniques for malicious security. We are also interested in achieving malicious security, i.e., enforcing honest behaviors of the server, while avoiding SNARKs. Our idea is to rely on probabilistic checking, and to leverage an error-correcting code to amplify the probability of detection. As mentioned before, each block is divided into C chunks. We will have the client randomly sample security parameter $\lambda \ll C$ chunks per block (the same random choice for all blocks), referred to as *verification chunks*, and use standard memory checking to ensure their authenticity and freshness. On each step, the server will perform homomorphic select operations on all C chunks in a block, and the client will perform the same homomorphic select operations on the λ verification chunks. In this way, whenever the server returns the client some encrypted block, the client can check whether the λ corresponding chunks match the verification chunks.

Unfortunately, the above scheme does not guarantee negligible failure of detection. For example, the server can simply tamper with a random chunk and hope that it's not one of the verification chunks. Clearly, the server succeeds with non-negligible probability. The fix is to leverage an error-correcting code to encode the original C chunks of each block into $C' = 2C$ chunks, and ensure that as long as $\frac{3}{4}C'$ chunks are correct, the block can be correctly decoded. Therefore, the server knows *a priori* that it will have to tamper with at least $\frac{1}{4}C'$ chunks to cause any damage at all, in which case it will get caught except with negligible probability.

4.4 Bounded Feedback ORAM Protocol

We now present the bounded feedback ORAM, a traditional ORAM scheme without server computation, to illustrate its important features.

4.4.1 The Basics

Like Ring ORAM, the bounded feedback ORAM organizes server storage as a binary tree of nodes [46]. The binary tree has $L + 1$ levels, where the root is at level 0 and the leaves are at level L. Each node in the binary tree is called a bucket and can contain up to Z data blocks. The leaves are numbered $0, 1, \ldots, 2^L - 1$ in the natural manner. Pseudo-code for our algorithm is given in Figure 4-1 and described below.

Note that many parts of our algorithm refer to *paths* down the tree where a path is a contiguous sequence of buckets from the root to a leaf. For a leaf bucket l, we refer to the path to l as path l or $\mathcal{P}(l)$. $\mathcal{P}(l, k)$ denotes the bucket at level $k \in [0..L]$ on $\mathcal{P}(l)$. Specifically, $\mathcal{P}(l, 0)$ denotes the root, and $\mathcal{P}(l, L)$ denotes the leaf bucket on $\mathcal{P}(l)$.

[2] We remark that some hierarchical ORAM schemes (e.g., [13]) also have bounded feedback, but achieve worse results in different respects relative our construction (e.g., worse server storage, deeper select circuits), when combined with server computation.

Main invariant. Like all tree-based ORAMs, each block is associated with a random path and we say that each block can only live in a bucket along that path at any time. In a local position map, the client stores the path associated to each block.

Recursion. To avoid incurring a large amount of client storage, the position map should be recursively stored in other smaller ORAMs [46]. When the data block size is $\Omega(\log^2 N)$ for an N element ORAM—which will be the case for all of our final parameterizations—the asymptotic costs of recursion (in terms of server storage or bandwidth blowup) are insignificant relative to the main ORAM [4]. Thus, for the remainder of the chapter, we no longer consider the bandwidth cost of recursion.

Metadata. To enable all ORAM operations, each block of data in the ORAM tree is stored alongside its address and leaf label (the path the block is mapped to). This metadata is encrypted using a semantically secure encryption scheme.

ORAM Request. Requesting a block with address a (ReadPath in Figure 4-1) is similar to most tree-based ORAMs: look up the position map to obtain the path block a is currently mapped to, read all blocks on that path to find block a, invalidate block a, remap it to a new random path and add it to the root bucket. This involves decrypting the address metadata of every block on the path (Line 16) and setting one address to $\perp$ (Line 18). All addresses must be then re-encrypted to hide which block was invalidated.

ORAM Eviction. The goal of eviction is to percolate blocks towards the leaves to avoid bucket overflows and it is this procedure where we differ from existing tree-based ORAMs [46, 61, 71, 5, 91]. We now describe our eviction procedure in detail.

4.4.2 New Triplet Eviction Procedure

We combine techniques from [46], [61] and Ring ORAM (Chapter 3) to design a novel eviction procedure (Evict in Figure 4-1) that enables us to break select operation feedback.

Triplet eviction on a path. Similar to other Tree ORAMs, eviction is performed along a path. To perform an eviction: For every bucket $\mathcal{P}(l_e, k)$ (k from 0 to L, i.e., from root to leaf), we move blocks from $\mathcal{P}(l_e, k)$ to its two children. Specifically, each block in $\mathcal{P}(l_e, k)$ moves to either the left or right child bucket depending on which move keeps the block on the path to its leaf (this can be determined by comparing the block's leaf label to l_e). We call this process a bucket-triplet eviction.

In each of these bucket-triplet evictions, we call $\mathcal{P}(l_e, k)$ the *source bucket*, the child bucket also on $\mathcal{P}(l_e)$ the *destination bucket*, and the other child the *sibling bucket*. A crucial change that we make to the eviction procedure of the original binary-tree ORAM [46] is that we move *all* the blocks in the source bucket to its two children.

Eviction frequency and order. For every A (a parameter proposed in Chapter 3, which we will set later) ORAM requests, we select the next path to evict based on the reverse lexicographical order of paths (Chapter 3). As we will see, the reverse lexicographical order eviction is crucial for our construction because it *evenly* and *deterministically* spreads out the eviction on all paths in the tree. Specifically, a bucket at level k will get evicted *exactly* every $A \cdot 2^k$ ORAM requests.

```
1: function Access(a, op, data′)
2:     l′ ← UniformRandom(0, 2^L − 1)
3:     l ← PositionMap[a]
4:     PositionMap[a] ← l′

5:     data ← ReadPath(l, a)
6:     if op = read then
7:         return data to client
8:     end if
9:     if op = write then
10:        data ← data′
11:    end if
12:    P(l, 0, cnt) ← (a, l′, data)

13:    Evict()
14: end function
```

```
15: function ReadPath(l, a)
16:     Read all blocks on path P(l)
17:     Select and return the block with address a
18:     Invalidate the block with address a
19: end function

20: function Evict( )
21:     Persistent variables cnt and G, initialized to 0
22:     cnt ← cnt + 1 mod A
23:     if cnt ≟ 0 then
24:         l_e ← bitreverse(G)
25:         EvictAlongPath(l_e)
26:         G ← G + 1 mod 2^L
27:     end if
28: end function

29: function EvictAlongPath(l_e)
30:     for k ← 0 to L − 1 do
31:         Read all blocks in P(l_e, k) and its two children
32:         Move all blocks in P(l_e, k) to its two children
33:             ▷ P(l_e, k) is empty at this point (Observation 1)
34:     end for
35: end function
```

Figure 4-1: Bounded Feedback ORAM (no server computation). Note that our construction differs from the original tree ORAM [46] only in the Evict procedure. We split Evict into EvictAlongPath to simplify the presentation later.

Setting parameters for bounded feedback. As mentioned, we require that during a bucket-triplet eviction, *all* blocks in the source bucket move to the two child buckets. The last step to achieve bounded feedback is to show that child buckets will have enough room to receive the incoming blocks, i.e., no child bucket should ever overflow except with negligible probability. (If any bucket overflows, we have experienced ORAM failure.) We guarantee this property by setting the bucket size Z and the eviction frequency A properly. According to the following lemma, if we simply set $Z = A = \Theta(\lambda)$, the probability that a bucket overflows is $2^{-\Theta(\lambda)}$, exponentially small.

Lemma 4 (No bucket overflows)**.** *If $Z \geq A$ and $N \leq A \cdot 2^{L-1}$, the probability that a bucket overflows after an eviction operation is bounded by $e^{-\frac{(2Z-A)^2}{6A}}$.*

The proof of Lemma 4 relies on a careful analysis of the stochastic process stipulated by the reverse lexicographic ordering of eviction, and boils down to a Chernoff bound. We defer the full proof to Section 4.8.1. Now, Lemma 4 with $Z = A = \Theta(\lambda)$ immediately implies the following key observation.

Observation 1 (Empty source bucket)**.** *After a bucket-triplet eviction, the source bucket is empty.*

Furthermore, straightforwardly from the definition of reverse lexicographical order, we have,

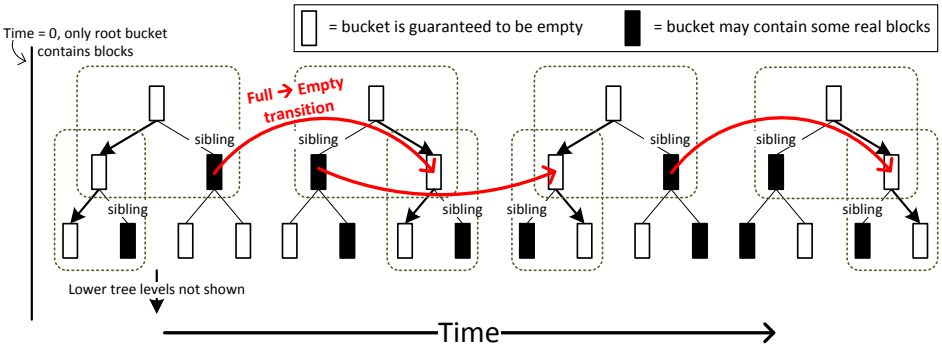


Figure 4-2: ORAM tree state immediately after each of a sequence of four evictions. After an eviction, the buckets on the eviction path (excluding the leaves) are guaranteed to be empty. Further, at the start of each eviction, each sibling bucket for that eviction is guaranteed to be empty. **Notations:** Assume the ORAM tree has more levels (not shown for simplicity). The eviction path is marked with black arrows. The dotted boxes indicate bucket triplets during each eviction. Thick red arrows indicate *causality* in the second tree level: if the arrow source is full at the end of eviction i, the bounded feedback ORAM guarantees it will be empty at the end of eviction $i+1$. These arrows, at each level, will look symmetric over time.

Observation 2. *In reverse-lexicographic order eviction, each bucket rotates between the following roles in the following order: source, sibling, and destination.*

These observations together guarantee that buckets are empty at public and predetermined times, as illustrated in Figure 4-2.

Towards bounded feedback. The above two observations are the keys to achieving bounded feedback. An empty source bucket b will be a sibling bucket the next time it is involved in a triplet eviction. So select operations that move blocks into b do not get feedback from b itself. Thus, the number of encryption layers (with AHE) or ciphertext noise (SWHE) becomes a function of previous levels in the tree only, which we can tightly bound later in Lemma 5 in Section 4.5.3.

Constant server storage blowup. We note that under our parameter setting $N \leq A \cdot 2^{L-1}$ and $Z = A$, our bounded feedback ORAM's server storage is $O(2^{L+1} \cdot Z \cdot B) = O(BN)$, a constant blowup.

4.5 Onion ORAM (Additively Homomorphic Encryption)

In this section, we describe how to leverage an AHE scheme with additive ciphertext expansion to transform our bounded feedback ORAM into our semi-honest secure Onion ORAM scheme. First, we detail the homomorphic select operation that we introduced in Section 4.3.

4.5.1 Additively Homomorphic Select Sub-protocol

Suppose the client wishes to select the i^*-th block from m blocks denoted $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$, each with $\ell_1, \ldots, \ell_m$ layers of encryption respectively. The sub-protocol works as follows:

1. Let $\ell := \max(\ell_1, \ldots, \ell_m)$. The client creates and sends to the server the following encrypted select vector $\langle \mathcal{E}_{\ell+1}(b_1), \mathcal{E}_{\ell+1}(b_2), \ldots \mathcal{E}_{\ell+1}(b_m) \rangle$, where $b_{i^*} = 1$ and $b_i = 0$ for $i \neq i^*$.

2. The server "lifts" each block to ℓ-layer ciphertexts, simply by continually re-encrypting a block until it has ℓ layers $\mathsf{ct}'_i[j] = \mathcal{E}_\ell(\mathcal{E}_{\ell-1}(\ldots \mathcal{E}_{\ell_i}(\mathsf{ct}_i[j])))$.

3. The server evaluates the homomorphic select operation on the lifted blocks: $\mathsf{ct}_{out}[j] := \bigoplus_i (\mathcal{E}_{\ell+1}(b_i) \otimes \mathsf{ct}'_i[j]) = \mathcal{E}_{\ell+1}(\mathsf{ct}'_{i^*})$. The outcome is the selected block ct_{i^*} with $\ell + 1$ layers of encryption.

As mentioned in Section 4.3, we divide each block into C chunks. Each chunk is encrypted separately. All C chunks share the same select vector—therefore, encrypting each element in the select vector only incurs the chunk size (instead of the block size).

We stress again that every time a homomorphic select operation is performed, the output block gains an extra layer of encryption, on top of $\ell = \max(\ell_1, \ldots, \ell_m)$ onion layers. This poses the challenge of bounding onion encryption layers, which we address in Section 4.5.3.

4.5.2 Detailed Protocol

We now describe the detailed protocol. Recall that each block is tagged with the following metadata: the block's logical address and the leaf it is mapped to, and that the size of the metadata is independent of the block size.

Initialization. The client runs a key generation routine for all layers of encryption, and gives all public keys to the server.

Read path. $\mathsf{ReadPath}(l, a)$ from Section 4.4.1 can be done with the following steps:

1. Client downloads and decrypts the addresses of all blocks on path l, locates the block of interest a, and creates a corresponding select vector $\vec{b} \in \{0,1\}^{Z(L+1)}$.

2. Client and server run the homomorphic select sub-protocol with client's input being encryptions of each element in $\vec{b}$ and server's input being all encrypted blocks on path l. The outcome of the sub-protocol—block a—is sent to the client.

3. Client re-encrypts and writes back the addresses of all blocks on path l, with block a now invalidated. This removes block a from the path without revealing its location. Then, the client re-encrypts block a (possibly modified) under 1 layer, and appends it to the root bucket.

Eviction. To perform $\mathsf{EvictAlongPath}(l_e)$, do the following for each level k from 0 to $L-1$:

1. Client downloads all the metadata (addresses and leaf labels) of the bucket triplet. Based on the metadata, the client determines each block's location after the bucket-triplet eviction.

2. For each slot to be written in the two child buckets:

   - Client creates a corresponding select vector $\vec{b} \in \{0,1\}^{2Z}$.
   - Client and server run the homomorphic select sub-protocol with the client's input being encryptions of each element in $\vec{b}$, and the server's input being the child bucket (being written to) and its parent bucket. Note that if the child bucket is empty due to Observation 1 (which is public information to the server), it conceptually has zero encryption layers.
   - Server overwrites the slot with the outcome of the homomorphic select sub-protocol.

4.5.3 Bounding Layers

Given the above protocol, we bound layers with the following lemma:

Lemma 5. *Any block at level $k \in [0..L]$ has at most $2k+1$ encryption layers.*

The proof of Lemma 5 is deferred to Section 4.8.2. The key intuition for the proof is that due to the reverse-lexicographic eviction order, each bucket will be written to exactly twice (i.e., be a destination or sibling bucket) before being emptied (as a source bucket). Also in Section 4.8.2, we introduce a further optimization called the "copy-to-sibling" optimization, which yields a tighter bound: blocks at level $k \in [0..L]$ will have only $k+1$ layers.

Eviction post-processing—peel off layers in leaf. The proof only applies to non-leaf buckets: blocks can stay inside a leaf bucket for an unbounded amount of time. Therefore, we need the following post-processing step for leaf nodes. After $\mathsf{EvictAlongPath}(l_e)$, the client downloads all blocks from the leaf node, peels off the encryption layers, and writes them back to the leaves as layer-$\Theta(L)$ re-encrypted ciphertexts (meeting the same layer bound as other levels). Since the client performs an eviction every A ORAM requests, and each leaf bucket has size $Z = A$, this incurs only $O(1)$ amortized bandwidth blowup.

4.5.4 Remarks on Cryptosystem Requirements

Let L' be the layer bound (derived in Section 4.5.3). To get constant bandwidth blowup, we require the output of an arbitrary select operation performed during an ORAM request (note that $\ell = L'$ in this case) to be a constant times larger than the block size B. Since $L' = \omega(1)$, this implies we need additive blowup per encryption layer, independent of L' (any multiplicative blowup is too large). One cryptosystem that satisfies the above requirement, for appropriate parameters, is the Damgård-Jurik cryptosystem (Section 4.7.2). We use this scheme to derive final parameters for the AHE construction in Section 4.7.3.

4.6 Security Against A Fully Malicious Server

So far, we have seen an ORAM scheme that achieves security against an *honest-but-curious* server who follows the protocol correctly. We now show how to extend this to get a scheme that is secure against a fully malicious server who can deviate arbitrarily from the protocol.

4.6.1 Abstract Server Computation ORAM

We start by describing several abstract properties of the Onion ORAM scheme from the previous section. We will call any server computation ORAM scheme satisfying these properties an *abstract server computation ORAM*.

Data blocks and metadata. The server storage consists of two types of data: *data blocks* and *metadata*. The server performs computation on data blocks, but never on metadata. The client reads and writes the metadata directly, so the metadata can be encrypted under any semantically secure encryption scheme.

Operations on data blocks. Following the notations in Section 4.3, each plaintext data block is divided into C chunks, and each chunk is separately encrypted $\mathsf{ct}_i = (\mathsf{ct}_i[1], \ldots, \mathsf{ct}_i[C])$. The client operates on the data blocks either by: (1) directly reading/writing an encrypted data block, or (2) instructing the server to apply a function f to form a new data block ct_i, where $\mathsf{ct}_i[j]$ only depends on the j-th chunk of other data blocks, i.e., $\mathsf{ct}_i[j] = f(\mathsf{ct}_1[j], \ldots, \mathsf{ct}_m[j])$ for all $j \in [1..C]$.

It is easy to check that the two Onion ORAM schemes are instances of the above abstraction. The metadata consists of the encrypted addresses and leaf labels of each data block, as well as additional space needed to implement ORAM recursion. The data blocks are encrypted under either a layered AHE scheme or a SWHE scheme. Function f is a "homomorphic select operation", and is applied to each chunk.

4.6.2 Semi-Honest to Malicious Compiler

We now describe a generic compiler that takes any "abstract server computation ORAM" that satisfies honest-but-curious security and compiles it into a "verified server computation ORAM" which is secure in the fully malicious setting.

Verifying metadata. We can use standard "memory checking" [11] schemes based on Merkle trees [7] to ensure that the client always gets the correct metadata, or aborts if the malicious server ever sends an incorrect value. A generic use of a Merkle tree would add an $O(\log N)$ multiplicative overhead to the process of accessing metadata [64], which is good enough for us. This $O(\log N)$ overhead can also be avoided by aligning the Merkle tree with the ORAM tree [68], or using generic authenticated data structures [80]. In any case, verifying metadata is basically free in Onion ORAM.

Challenge of verifying data blocks. Unfortunately, we cannot rely on standard memory checking to protect the encrypted data blocks when the client doesn't read/write them directly but rather instructs the server to compute on them. The problem is that a malicious server that learns some information about the client's access pattern based on *whether the client aborts or not*.

Consider Onion ORAM for example. The malicious server wants to learn if, during the homomorphic select operation of a ORAM request, the location being selected is i. The server can perform the operation correctly except that it would replace the ciphertext at position i with some incorrect value. In this case, if the location being selected was indeed i then the client will abort since the data it receives will be incorrect, but otherwise the client will accept. This violates ORAM's privacy requirement.

A more general way to see the problem is to notice that the client's abort decision above depends on the decrypted value, which depends on the secret key of the homomorphic encryption scheme. Therefore, we can no longer rely on the semantic security of the encryption scheme if the abort decision is revealed to the server. To fix this problem, we need to ensure that the client's abort decision only depends on ciphertext and not on the plaintext data.

Verifying data blocks. For our solution, the client selects a random subset $\mathcal{V}$ consisting of λ chunk positions. This set $\mathcal{V}$ is kept secret from the server. The subset of chunks in positions $\{j : j \in \mathcal{V}\}$ of every encrypted data block are treated as additional metadata, which we call the "verification chunks". Verification chunks are encrypted and memory checked in the same way as the other metadata. Whenever the client instructs the server to update an encrypted data block, the client performs the same operation himself on the verification chunks. Then, when the client reads an encrypted data block from the server, he can check the chunks in $\mathcal{V}$ against the ciphertexts of verification chunks. This check ensures that the server cannot modify too many chunks without getting caught. To ensure that this check is sufficient, we apply an error-correcting code which guarantees that the server has to modify a large fraction of chunks to affect the plaintext. In more detail:

- Every plaintext data block $\mathsf{pt} = (\mathsf{pt}[1], \ldots, \mathsf{pt}[C])$ is first encoded via an error-correcting code into a codeword block $\mathsf{pt\_ecc} = \mathsf{ECC}(\mathsf{pt}) = (\mathsf{pt\_ecc}[1], \ldots, \mathsf{pt\_ecc}[C'])$. The error-correcting code ECC has a rate $C/C' = \alpha < 1$ and can efficiently recover the plaintext block if at most a δ-fraction of the codeword chunks are erroneous. For concreteness, we can use a Reed-Solomon code, and set $\alpha = \frac{1}{2}, \delta = (1-\alpha)/2 = \frac{1}{4}$. The client then uses the "abstract server computation ORAM" over the codeword blocks $\mathsf{pt\_ecc}$ (instead of pt).

- During initialization, the client selects a secret random set $\mathcal{V} = \{v_1, \ldots, v_\lambda\} \subseteq [C']$. Each ciphertext data block ct_i has verification chunks $\mathsf{verCh}_i = (\mathsf{verCh}_i[1], \ldots, \mathsf{verCh}_i[\lambda])$. We ensure the invariant that, during an honest execution, $\mathsf{verCh}_i[j] = \mathsf{ct}_i[s_j]$ for $j \in [1..\lambda]$.

- The client uses a memory checking scheme to ensure the authenticity and freshness of the metadata including the verification chunks. If the client detects a violation in metadata at any point, the client aborts (we call this abort_0).

- Whenever the client directly updates or instructs the server to apply the aforementioned function f on an encrypted data block ct_i, it also updates or applies the same function f on the corresponding verification chunks $\mathsf{verCh}_i[j]$ for $j \in [1..\lambda]$, which possibly involves reading other verification chunks that are input to f.

- When the client reads an encrypted data block ct_i, it also reads verCh_i and checks that $\mathsf{verCh}_i[j] = \mathsf{ct}_i[s_j]$ for each $j \in [1..\lambda]$ and aborts if this is not the case (we call this abort_1). Otherwise the client decrypts ct_i to get $\mathsf{pt\_ecc}_i$ and performs error-correction to recover pt_i. If the error-correction fails, the client aborts (we call this abort_2).

If the client ever aborts during any operation with $\mathsf{abort}_0, \mathsf{abort}_1$ or abort_2, it refuses to perform any future operations. This completes the compiler which gives us Theorem 2.

Security Intuition. Notice that in the above scheme, the decision whether abort_1 occurs does not depend on any secret state of the abstract server computation ORAM scheme, and therefore can be revealed to the server without sacrificing privacy. We will argue that, if abort_1 does not occur, then the client retrieves the correct data (so abort_2 will not occur) with overwhelming probability. Intuitively, the only way that a malicious server can cause the client to either retrieve the incorrect data or trigger abort_2 without triggering abort_1 is to modify at least a δ (by default, $\delta = 1/4$) fraction of the chunks in an encrypted data block, but avoid modifying any of the λ chunks corresponding to the secret set $\mathcal{V}$. This happens with probability at most $(1-\delta)^\lambda$ over the random choice of $\mathcal{V}$, which is negligible. The complete proof is given in Section 4.8.3.

4.7 Optimizations and Analysis

In this section we present two optimizations, an asymptotic analysis and a concrete (with constants) analysis for our AHE-based protocol.

4.7.1 Optimizations

Hierarchical Select Operation and Sorting Networks. For simplicity, we have discussed select operations as inner products between the data vector and the coefficient vector. As an optimization, we may use the Lipmaa construction [28] to implement select hierarchically as a tree of d-to-1 select operations for a constant d (say $d = 2$). In that case, for a given 1 out of Z selection, $\vec{b}^{\mathsf{hier}} \in \{0,1\}^{\log Z}$. Eviction along a path requires $O(\log N)$ bucket-triplet operations, each of which is a Z-to-Z permutation. To implement an arbitrary Z-to-Z permutation, we can use the Beneš sorting network, which consists of a total of $O(Z \log Z)$ 2-to-1 select operations per triplet.

At the same time, both the hierarchical select and the Beneš network add $\Theta(\log Z)$ layers to the output as opposed to a single layer. Clearly, this makes the layer bound from Lemma 5 increase to $\Theta(\log Z \log N)$. However, we can set related parameters larger to compensate.

Permuted Buckets. Observe that on a request operation, the client and the server need to run a homomorphic select protocol among $O(\lambda \log N)$ blocks. We can reduce this number to $O(\log N)$ blocks during the online phase of the request, and $O(\lambda)$ blocks per request overall, using the permuted bucket technique from Ring ORAM (similar ideas were used in hierarchical ORAMs [13]).

Instead of reading all slots along the tree path during each read, we can randomly permute blocks in each bucket and only read/remove a block at a random looking slot (out of $Z = \Theta(\lambda)$ slots) per bucket. Each random-looking location will either contain the block of interest or a dummy block. We must ensure that no bucket runs out of dummies before the next eviction refills that bucket's dummies. Given our reverse-lexicographic eviction order, a simple Chernoff bound shows that adding $\Theta(A) = \Theta(\lambda)$ dummies, which increases bucket size by a constant factor, is sufficient to ensure that dummies do not run out except with probability $2^{-\Theta(\lambda)}$.

Additional care is needed to keep the root permuted. Each requested block now has to be written to a random (and unread) slot in the root bucket, as opposed to simply being appended as in Step 3 of ReadPath in Section 4.5.2. This requires a "PIR write" operation

from [79], which requires the client to send $O(Z)$ encrypted coefficients and the server to evaluate an operation similar to $O(Z)$ 2-to-1 homomorphic select operations (see [79] for details). We make two remarks. First, this operation occurs offline — after the client receives the block. Second, the download phase in the protocol from [79] is not needed since unread slots can be set to encryptions of 0 by the server after each eviction.

4.7.2 Damgård-Jurik Cryptosystem

We will implement our AHE-based protocol over the Damgård-Jurik cryptosystem [21], a generalization of Paillier's cryptosystem [17]. Both schemes are based on the hardness of the decisional composite residuosity assumption. In this system, the public key $\mathsf{pk} = n = pq$ is an RSA modulus (p and q are two large, random primes) and the secret key $\mathsf{sk} = \mathsf{lcm}(p-1, q-1)$. In the terminology from our onion encryptions, $\mathsf{sk}_i, \mathsf{pk}_i = \mathcal{G}_i()$ for $i \geq 0$.

We denote the integers mod n as $\mathbb{Z}_n$. The plaintext space for the i-th layer of the Damgård-Jurik cryptosystem encryption, $\mathbb{L}_i$, is $\mathbb{Z}_{n^{s_0+i}}$ for some user specified choice of s_0. The ciphertext space for this layer is $\mathbb{Z}_{n^{s_0+i+1}}$. Thus, we clearly have the property that ciphertexts are valid plaintexts in the next layer. Let γ be the number of bits in n, i.e., $\gamma = |n|$. After i layers of Damgård-Jurik onion encryption, a message of length γs_0 bits is encrypted to a ciphertext of length $\gamma(s_0 + i + 1)$ bits. An interesting property that immediately follows is that if $s_0 = \Theta(i)$, then $|\mathbb{L}_i|/|\mathbb{L}_0|$ is a constant. In other words, by setting s_0 appropriately the ciphertext blowup after i layers of encryption is a constant.

We further have that $\oplus$ (the primitive for homomorphic addition) is integer multiplication and $\otimes$ (for scalar multiplication) is modular exponentiation. If these operations are performed on ciphertexts in $\mathbb{L}_i$, operations are mod $\mathbb{Z}_{n^{s_0+i}}$.

4.7.3 Asymptotic Analysis

We first perform the asymptotic analysis for exact exponential security, using the Damgård-Jurik Cryptosystem, and summarize the results in Table 4.2. The results for negligible in N security in Table 4.1 are derived by setting $\lambda = \omega(\log N)$ and $\gamma = \Theta(\log^3 N)$ according to best known attacks [28].

Semi-Honest Case

Chunk size. Using a Beneš network, each ciphertext chunk accumulates $O(\log \lambda \log N)$ layers of encryption at the maximum. Suppose the plaintext chunk size is $B_c := \gamma s_0$, then at the maximum onion layer, the ciphertext size would be $\gamma(s_0 + O(\log \lambda \log N))$. Therefore, to ensure constant ciphertext expansion at all layers, it suffices to set $s_0 := \Omega(\log \lambda \log N)$ and chunk size $B_c := \Omega(\gamma \log \lambda \log N)$. This means ciphertext chunks and homomorphic select vectors are also $\Omega(\gamma \log \lambda \log N)$ bits.

Size of select vectors. Each read requires $O(\log \lambda)$ encrypted coefficients of $O(B_c)$ bits each. Eviction along a path requires $O(\log N)$ Beneš networks (one for each bucket-triplet), a total of $O(\lambda \log \lambda \log N)$ encrypted coefficients. Also recall that one eviction happens per $A = \Theta(\lambda)$ accesses. Therefore, the select vector size per ORAM access (amortized) is dominated by evictions, and is $\Theta(B_c \log \lambda \log N)$ bits.

Setting the block size. We want our block size to be asymptotically larger than the select vectors at each step of our protocol (other metadata are much smaller). Clearly, if we set the block size to be $B := \Omega(B_c \log \lambda \log N)$, the cost of homomorphic select vectors could be asymptotically absorbed, thereby achieving constant bandwidth blowup. Since the chunk size $B_c = \Omega(\gamma \log \lambda \log N)$, we have $B = \Omega(\gamma \log^2 \lambda \log^2 N)$ bits.

Server computation The bottleneck of server computation is to homomorphically multiply a block with a encrypted select coefficient. In Damgård-Jurik, this is a modular exponentiation operation, which has $\widetilde{O}(\gamma^2)$ computational complexity for γ-bit ciphertexts. This means the *per-bit* computational overhead is $\widetilde{O}(\gamma)$. The server needs to perform this operation on $O(\lambda)$ blocks of size B, and therefore has a computational overhead of $\widetilde{O}(\gamma)O(B\lambda)$.

Client computation Client needs to decrypt $O(\log \lambda \log N)$ layers to get the plaintext block, and therefore has a computational overhead of $\widetilde{O}(\gamma)O(B \log \lambda \log N)$.

Malicious Case

Setting the block size. The main difference from the semi-honest case is that on a read, the client must additionally download $\Theta(\lambda)$ verification chunks from each of the $\Theta(\lambda)$ blocks (assuming permuted buckets). Select vector size stays the same, and the error-correcting code increases block size by only a constant factor. Thus, the block size we need to achieve constant bandwidth over the entire protocol is $B = \Omega(B_c \lambda^2) = \Omega(\gamma \lambda^2 \log \lambda \log N)$.

Client computation. Another difference is that the client now needs to emulate the server's homomorphic select operation on the verification chunks. But a simple analysis will show that the bottleneck of client computation is still onion decryption, and therefore remains the same asymptotically.

4.7.4 Concrete Analysis (Semi-honest case only)

Figure 4-3 shows bandwidth as a function of block size for our optimized semi-honest construction, taking into account all constant factors (including the extra bandwidth cost to recursively look up the position map). Other scheme variants in this chapter have the same general trend. We compare to Path PIR and Circuit ORAM, the most bandwidth-efficient constructions with/without server computation that match our server/client storage asymptotics.

Takeaway. The high order bit is that as block size increases, Onion ORAM's bandwidth approaches $2B$. Note that $2B$ is the inherent lower bound in bandwidth since every ORAM access must at least read the block of interest from the server and send it back after possibly modifying it. Given an 8 MB block size, which is approximately the size of an image file, we improve in bandwidth over Circuit ORAM by **35**× and improve over Path PIR by **22**×. For very large block sizes, our improvement continues to increase but Circuit ORAM and Path PIR improve less dramatically because their asymptotic bandwidth blowup has a $\log N$ factor. Note that for sufficiently small block sizes, both Path PIR and Circuit ORAM beat our bandwidth because our select vector bandwidth dominates. Yet, this crossover point is around 128 KB, which is reasonable in many settings.

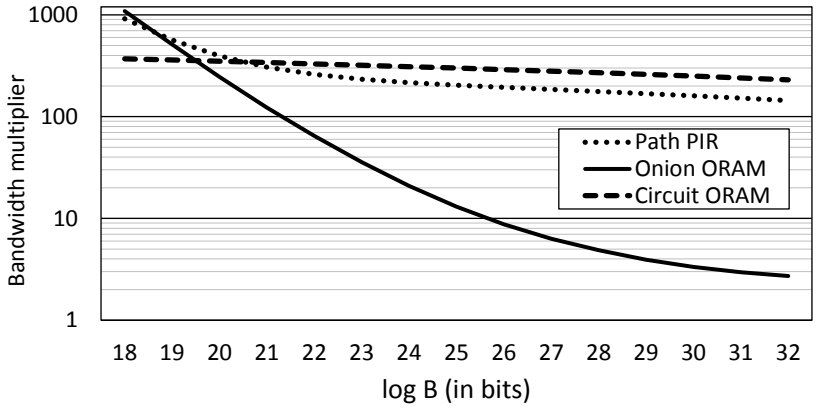


Figure 4-3: Plots the bandwidth multiplier (i.e., the hidden constant for $O(B)$) for semi-honest Onion ORAM and two prior proposals. We fix the ORAM capacity to $NB = 2^{50}$ and give each scheme the same block size across different block sizes (hence as B increases, N decreases).

Constant factor optimization: Less frequent leaf post-processing. In the above evaluation, we apply an additional constant factor optimization. Since $Z = A = \Theta(\lambda)$, we must send and receive one additional data block (amortized) per ORAM request to post-process leaf buckets during evictions (Section 4.5.3). To save bandwidth, we can perform this post-processing on a particular leaf bucket every p evictions to that leaf (p is a free variable). The consequence is that the number of layers that accumulate on leaf buckets increases by p which makes each ORAM read path more expensive by the corresponding amount. In practice, $p \geq 8$ yields the best bandwidth.

Parameterization details. For both schemes, we set acceptable ORAM failure probability to 2^{-80} which results in $Z = A \approx 300$ for Onion ORAM, $Z = 120$ for Path PIR [46] and a stash size (stored on the server) of 50 blocks for Circuit ORAM [5]. For Onion ORAM and Path PIR we set $\gamma = 2048$ bits. For Circuit ORAM, we use the reverse lexicographic eviction order as described in that work, which gives 2 evictions per access and $Z = 2$. For Path PIR, we set the eviction frequency $v = 2$ [46].

4.7.5 Other Optimizations

De-Amortization. We remark that it is easy to de-amortize the above algorithm so that the worst-case bandwidth equals amortized bandwidth and overall bandwidth doesn't increase. First, it is trivial to de-amortize the leaf bucket post-processing (Section 4.5.3) over the A read path operations because $A = Z$ and post-processing doesn't change the underlying plaintext contents of that bucket. Second, the standard de-amortization trick of Williams et al. [58] can be applied directly to our EvictAlongPath operation. We remark that it is easy to de-amortize evictions over the next A read operations because moving blocks from buckets (possibly on the eviction path) to the root bucket does not impact our eviction algorithm.

Online Roundtrips. The standard recursion technique [4] uses a small block size for position map ORAMs (to save bandwidth) and requires $O(\log N)$ roundtrips. In Onion

ORAM, the block in the main ORAM is large $B = \Omega(\lambda \log N)$. We can use Onion ORAM with the same large block size for position map ORAMs. This achieves a constant number of recursive levels if N is polynomial in λ, and therefore maintains the constant bandwidth blowup.

4.8 Proofs

4.8.1 Bounded Feedback ORAM: Bounding Overflows

We now give formal proofs to show that buckets do not overflow in bounded feedback ORAM except with negligible probability.

Proof: (of Lemma 4). First of all, notice that when $Z \geq A$, the root bucket will never overflow. So we will only consider non-root buckets. Let b be a non-root bucket, and $Y(b)$ be the number of blocks in it after an eviction operation. We will first assume all buckets have infinite capacity and show that $E[Y(b)] \leq A/2$, i.e., the expected number of blocks in a non-root bucket after an eviction operation is no more than $A/2$ at any time. Then, we bound the overflow probability given a finite capacity.

If b is a leaf bucket, each of the N blocks in the system has a probability of 2^{-L} to be mapped to b independently. Thus $E[Y(b)] \leq N \cdot 2^{-L} \leq A/2$.

If b is a non-leaf (and non-root) bucket, we define two variables m_1 and m_2: the last EvictAlongPath operation where b is on the eviction path is the m_1-th EvictAlongPath operation, and the EvictAlongPath operation where b is a sibling bucket is the m_2-th EvictAlongPath operation. If $m_1 > m_2$, then $Y(b) = 0$, because b becomes empty when it is the source bucket in the m_1-th EvictAlongPath operation. (Recall that buckets have infinite capacity so this outcome is guaranteed.) If $m_1 < m_2$, there will be some blocks in b and we now analyze what blocks will end up in b. We time-stamp the blocks as follows. When a block is accessed and remapped, it gets time stamp m^*, which is the number of EvictAlongPath operations that have happened. Blocks with $m^* \leq m_1$ will not be in b as they will go to either the left child or the right child of b. Blocks with $m^* > m_2$ will not be in b as the last eviction operation that touches b (m_2-th) has already passed. Therefore, only blocks with time stamp $m_1 < m^* \leq m_2$ can be in b. There are at most $d = A|m_1 - m_2|$ such blocks. Such a block goes to b if and only if it is mapped to a path containing b. Thus, each block goes to b independently with a probability of 2^{-i}, where i is the level of b. The deterministic order of EvictAlongPath makes it easy to see[3] that $|m_1 - m_2| = 2^{i-1}$. Therefore, $E[Y(b)] \leq d \cdot 2^{-i} = A/2$ for any non-leaf bucket as well.

Now that we have independence and the bound on expectation, a simple Chernoff bound completes the proof. ■

4.8.2 Onion ORAM: Bounding Layers of Encryption

To bound the layers of onion encryption, we consider the following abstraction. Suppose all buckets in the tree have a layer associated with it.

- The root bucket contains layer-1 ciphertexts.
- For a bucket known to be empty, we define bucket.layer := 0.

[3] One way to see this is that a bucket b at level i will be on the evicted path every 2^i EvictAlongPath operations, and its sibling will be on the evicted path halfway in that period.

- Each bucket-triplet operation moves data from parent to child buckets. After the operation, $\mathsf{child.layer} := \max\{\mathsf{parent.layer}, \mathsf{child.layer}\} + 1$.

Recall that we use the following terminology. The bucket being evicted from is called the *source*, its child bucket on the eviction path is called the *destination*, and its other child forking off the path is called the *sibling*.

Proof: (of Lemma 5). We prove by induction.

Base case. The lemma holds obviously for the root bucket.

Inductive step. Suppose that this holds for all levels $\ell < k$. We now show that this holds for level k. Let bucket denote a bucket at level k. We focus on this particular bucket, and examine bucket.layer after each bucket-triplet operation that involves bucket. It suffices to show that after each bucket-triplet operation involving bucket, it must be that $\mathsf{bucket.layer} \leq 2k + 1$. If a bucket-triplet operation involves bucket as a source, we call it a *source operation* (from the perspective of bucket). Similarly, if a bucket-triplet operation involves bucket as a destination or sibling, we call it a *destination operation* or a *sibling operation* respectively.

Based on Observation 1,

$$\mathsf{bucket.layer} = 0 \qquad \text{(after each source operation)}$$

Since a sibling operation must be preceded by a source operation (if there is any preceding operation), bucket must be empty at the beginning of each sibling operation. By induction hypothesis, after each sibling operation, it must be that

$$\mathsf{bucket.layer} \leq 2(k-1) + 1 + 1 = 2k \qquad \text{(after each sibling operation)}$$

Since a destination operation must be preceded by a sibling operation (if there is any preceding operation), from the above we know that at the beginning of a destination operation bucket.layer must be bounded by $2k$. Now, by induction hypothesis, it holds that

$$\mathsf{bucket.layer} \leq 2k + 1 \qquad \text{(after each destination operation)}$$

Finally, our post-processing on leaves where the client peels off the onion layers extends this lemma to all levels including leaves. ■

Copy-to-sibling optimization and a tighter layer bound An immediate implication of Observation 1 plus Observation 2 is that whenever a source evicts into a sibling, the sibling bucket is empty to start with because it was a source bucket in the last operation it was involved in. This motivates the following optimization: the server can simply copy blocks from the source bucket into the sibling. The client would read the metadata corresponding to blocks in the source bucket, invalidate blocks that do not belong to the sibling, before writing the (re-encrypted) metadata to the sibling.

This copy-to-sibling optimization avoids accumulating an extra onion layer upon writes into a sibling bucket. With this optimization and using a similar inductive proof, it is not hard to show a bucket at level k in the tree has at most $k + 1$ layers.

4.8.3 Malicious Security Proof

We now prove the security of our malicious construction matches Definition 2. See Sections 2.1 and 2.2 for simulator notations.

The Simulator. To simulate the setup protocol with some data of size N, the simulator chooses a dummy database D' of size N consisting of all 0s. It then follows the honest setup procedure on behalf of the client with database D'. To simulate each access operation, the simulator follows the honest protocol for reading a dummy index, say, $ind' = 0$, on behalf of the client.

During each operation, if the client protocol that's being executed by the simulator aborts then the simulator sends abort to $\mathcal{F}$ and stops responding to future commands on behalf of the client, else it gives ok to $\mathcal{F}$.

Sequence of Hybrids. We now follow a sequence of hybrid games to show that the real world and the simulation are indistinguishable:

$$|\Pr\left[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1\right] - \Pr\left[\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1\right]| \leq \mathsf{negl}(\lambda)$$

Game 0. Let this be the real game $\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}$ with an adversarial server $\mathcal{A}$ and an environment $\mathcal{Z}$.

Game 1. In this game, the client also keeps a local copy of the correct metadata and data-blocks (in plaintext) that should be stored on the server. Whenever the client reads any (encrypted) metadata from the server during any operation, if the memory checking does not abort, then instead of decrypting the read metadata, the client simply uses the locally stored plaintext copy.

The only difference between Game 0 and Game 1 occurs if in Game 0 the memory checking does not abort, but the client retrieves the incorrect encrypted metadata, which happens with negligible probability by the security of memory checking. Therefore Game 0 and Game 1 are indistinguishable.

Game 2. In this game the client doesn't store the correct values of verCh_i with the encrypted metadata on the server, but instead replaces these with dummy values. The client still stores the correct values of verCh_i in the plaintext metadata stored locally, which it uses to do all of the actual computations.

Game 1 and Game 2 are indistinguishable by the CPA security of the symmetric-key encryption scheme used to encrypt metadata. We only need CPA security since, in Games 1 and 2, the client never decrypts any of the metadata ciphertexts.

Game 3. In this game, whenever the client reads an encrypted data block ct_i from the server, if abort_1 does not occur, instead of decrypting and decoding the encrypted data-block, the client simply uses local copy of the plaintext data-block.

The only difference between Game 2 and Game 3 occurs if at some point in time the client reads an encrypted data block ct_i from the server such that at least a δ fraction of the ciphertext chunks $\{\mathsf{ct}_i[j]\}$ in the block have been modified (so that decoding either fails with abort_2 or returns an incorrect value) but none of the chunks in locations $i \in \mathcal{V}$ have been modified (so that abort_1 does not occur).

We claim that Game 2 and Game 3 are statistically indistinguishable, with statistical distance at most $q(1-\delta)^\lambda$, where q is the total number of operations performed by the client. To see this, note that in both games the set $\mathcal{V}$ is initially completely random and unknown to the adversarial server. In each operation i that the client reads an encrypted data-block, the server can choose some set $\mathcal{V}'_i \subseteq [C']$ of positions in which

the ciphertext chunks are modified, and if $\mathcal{V}'_i \cap \mathcal{V} = \emptyset$ the server learns this information about the set $\mathcal{V}$ and the game continues, else the client aborts and the game stops. The server never gets any other information about $\mathcal{V}$ throughout the game. The games 2 and 3 only diverge if at some point the adversarial server guesses a set $\mathcal{V}'_i$ of size $|\mathcal{V}'_i| \geq \delta C'$ such that $\mathcal{V} \cap \mathcal{V}'_i = \emptyset$. We call this the "bad event". Notice that the sets $\mathcal{V}'_i$ can be thought of as being chosen non-adaptively at the beginning of the game prior to the adversary learning any knowledge about $\mathcal{V}$ (this is because we know in advance that the server will learn $\mathcal{V}'_i \cap \mathcal{V} = \emptyset$ for all i prior to the game ending). Therefore, the probability that the bad event happens in the j'th operation is

$$\Pr_{\mathcal{V}}[\mathcal{V}'_j \cap \mathcal{V} = \emptyset] \leq \binom{(1-\delta)C'}{\lambda} / \binom{C'}{\lambda} \leq (1-\delta)^{\lambda}$$

where $\mathcal{V} \subseteq [C']$ is a random subset of size $|\mathcal{V}| = \lambda$. By the union bound, the probability that the bad event happens during some operation $j \in \{1, \ldots, q\}$ is at most $q(1-\delta)^{\lambda}$.

Game' 3. In this game, the client runs the setup procedure using the dummy database D' (as in the simulation) instead of the one given by the environment. Furthermore, for each access operation, the client just runs a dummy operation consisting of a read with the index $ind' = 0$ instead of the operation chosen by the environment. (We also introduce an ideal functionality $\mathcal{F}$ in this world which is given the correct database D at setup and the correct access operations as chosen by the environment. Whenever the client doesn't abort, it forwards the outputs of $\mathcal{F}$ to the environment.)

Games 3 and Game' 3 are indistinguishable due to the semi-honest Onion ORAM scheme. In particular, in both games whenever the client doesn't abort, the client reads the correct metadata and data blocks as when interacting with an honest server, and therefore follows the same protocols as when interacting with an honest server. Furthermore, the decision whether or not the client aborts in these games (with abort_0 or abort_1; there is no more abort_2) only depends on the secret set $\mathcal{V}$ and the internal state of the memory checking scheme, but is independent of any of the secret state or decryption keys of the underlying semi-honest Onion ORAM scheme. Therefore, the view of the adversarial server in these games can be simulated given the view of the honest server.

Game' 2,1,0. We define Game' i for $i = 0, 1, 2$ the same way as Game i except that the client uses the dummy database D' and the dummy operations (reads with index $idx' = 0$) instead of those specified by the environment.

The arguments that Game' $i+1$ and Game' i are indistinguishable as the same as those for Game $i+1$ and Game i. Finally, we notice that Game 0 is the ideal game $\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ with the simulator $\mathcal{S}$.

Putting everything together, we see that the real and ideal games $\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}$ and $\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ are indistinguishable as we wanted to show.

4.9 Onion ORAM (Somewhat Homomorphic Encryption)

As discussed in Section 4.3, our bounded feedback ORAM can be viewed as an ORAM protocol with *very shallow circuit depth*. Shallow circuit depth is interesting beyond onion

AHE: it also gives us a way to construct efficient constant bandwidth blowup ORAM using SWHE.

As an extension, we will describe how to efficiently map our bounded feedback ORAM to a BGV SWHE scheme [47].[4] Our construction based on BGV SWHE has the following attractive features over the construction that uses AHE:

1. **No layers.** SWHE select (multiplication) operations do not add encryption layers. Thus, the client only needs to decrypt once to retrieve the underlying plaintext.

2. **Coefficient packing to achieve small blocks.** One downside of AHE Onion ORAM is that block size is at least as large as the number of encrypted coefficients needed to manage reads and evictions. Using plaintext packing techniques developed for SWHE/FHE schemes [47, 50], we can pack many select bits into a single ciphertext and achieve a much smaller block size.

4.9.1 BGV-Style Somewhat Homomorphic Cryptosystems

BGV-style cryptosystems [47] are a framework for constructing efficient FHE or SWHE schemes. For concreteness, we assume the underlying cryptosystem is Ring Learning with Errors (RLWE) as in [47], although in practice, we can switch to the LTV variant of NTRU [55] which achieves better parameters by constant factors. Let $A(x)$ mod $(\Phi_m(x), p)$, denoted R_p for short, denote the ring of polynomials $A(x)$ modulo the m-th cyclotomic polynomial (of degree $n = \phi(m)$ where ϕ is the totient function) and coefficients modulo p. Plaintexts are polynomials in R_t for some t of the user's choice. If the circuit we are evaluating has depth d, a ciphertext encoding the plaintext polynomial $x \in R_t$ at the ℓ-th level $\ell \in [0..d]$ is an element in $R^2_{q_\ell}$ where $q_\ell/q_{\ell+1} = q_L/t = \mathsf{poly}(n)$ (RLWE ciphertexts consist of 2 polynomials).

The scheme is additively and multiplicatively homomorphic; that is, given $x, y \in R_t$ and $\mathcal{E}^{\mathsf{RLWE}}_\ell(x), \mathcal{E}^{\mathsf{RLWE}}_\ell(y) \in R^2_{q_\ell}$, we have efficient operations $\boxplus$ and $\boxtimes$ such that $\mathcal{E}^{\mathsf{RLWE}}_\ell(x) \boxplus \mathcal{E}^{\mathsf{RLWE}}_\ell(y) = \mathcal{E}^{\mathsf{RLWE}}_\ell(x+y) \in R^2_{q_\ell}$ and $\mathcal{E}^{\mathsf{RLWE}}_\ell(x) \boxtimes \mathcal{E}^{\mathsf{RLWE}}_\ell(y) = \mathcal{E}^{\mathsf{RLWE}}_{\ell+1}(x \cdot y) \in R^2_{q_{\ell+1}}$. Implicit in the $\boxtimes$ operator, there is a $\mathsf{Refresh}$ operation performed after each homomorphic multiplication operation. There is also a Scale procedure that promotes a level-i ciphertext into a level-j $(j > i)$ ciphertext encrypting the same plaintext.

Connection to encryption layers. The chain of moduli used by BGV is analogous to encryption layers in our AHE construction. That is, a BGV ciphertext living in $R^2_{q_\ell}$ is analogous to a ciphertext that has accumulated $\ell + 1$ AHE encryption layers in Section 4.5, and our layer bound in Section 4.5.3 applies directly through this isomorphism. To get a similar abstraction, we write $R^2_{q_\ell}$ as $\mathbb{L}_\ell$ just as in the AHE construction. However, unlike the AHE construction, here $\mathbb{L}_0$ has the biggest ciphertext size (to leave room for noise that will be added by future homomorphic operations). This will change several design decisions.

The other differences in the protocols are minor, which we explain below.

4.9.2 Somewhat Homomorphic Select Sub-protocol

We now describe a single homomorphic select, using the same notation as in Section 4.5.1:

[4] We will refer to BGV without bootstrapping as a SWHE scheme as opposed to as a "leveled FHE scheme".

1. Let $\ell := \max(\ell_1, \ldots, \ell_m)$. The client first creates and sends to the server the following homomorphic select vector $\langle \mathcal{E}_\ell^{\mathsf{RLWE}}(b_1), \mathcal{E}_\ell^{\mathsf{RLWE}}(b_2), \ldots \mathcal{E}_\ell^{\mathsf{RLWE}}(b_m) \rangle$.

2. The server "lifts" each ciphertext chunk (C chunks per block) to $\mathbb{L}_\ell$, by performing Scale on each chunk.

3. Then, the server evaluates the following homomorphic select operation on all chunks $j \in [1..C]$ (of the lifted blocks):

$$\mathsf{ct}_{out}[j] := \boxplus_i \left(\mathcal{E}_\ell^{\mathsf{RLWE}}(b_i) \boxtimes \mathsf{ct}'_i[j] \right) = \mathcal{E}_\ell^{\mathsf{RLWE}}(\mathsf{ct}'_{i^*})$$

Delay $\mathsf{Refresh}$. There is a subtle optimization that is worth mentioning. The $\mathsf{Refresh}$ procedure is typically performed after each homomorphic multiplication and is typically an expensive part of the BGV construction (costing poly-logarithmic factors of computation more than a regular homomorphic multiplication). However, we can perform $\mathsf{Refresh}$ only once per select operation, after the results of the select are added together. In this way, we can amortize its cost when the select operation's fan-in is $\Omega(\lambda)$.

4.9.3 Onion ORAM Protocol over BGV

We can plug in the BGV select sub-protocol from the previous section directly into the complete Onion ORAM protocol from Section 4.5.2. The only difference between AHE and SWHE is how we initialize the system, which we now detail:

Initialization and ORAM tree setup. To start, the user and server use the layer bound from Section 4.5.3 to compute $L' = 2L + 1$, the BGV circuit depth, from the ORAM tree height L. The user and server then agree on t and q_i for $i \in [0..L']$ which satisfy $t = \Omega(\mathsf{poly}(n))$ and $q_i/q_{i+1} = q'_L/t = \Omega(\mathsf{poly}(n))$. Finally, the user shares public and evaluation keys for each modulus with the server.

We remark that in Step 3 of $\mathsf{ReadPath}$ (Section 4.5.2), the client *must* re-encrypt blocks using a symmetric encryption scheme. Sending a "layer 1" ciphertext, which lives in $\mathbb{L}_0$, is no longer an option, because in BGV, $\mathbb{L}_0$ is $R_{q_0}^2$ where $q_0 = \mathsf{poly}(n)^{L'}$, which implies that sending back a $\mathbb{L}_0$ ciphertext immediately makes the ORAM bandwidth blowup increase to $\log q_0 / \log t = L' = \Theta(\log N)$!

4.9.4 Optimizations

We now present two optimizations specific to the SWHE construction.

CRT Packing to Reduce Coefficient Size

As with the AHE scheme from Section 4.5, the above scheme described thus far has to send a lot of encrypted coefficients to the server, which are large because each encrypted coefficient will live in $\mathbb{L}_0$. We can reduce block size substantially by using FHE CRT packing techniques, described in [47, 50]. At a high level, the client can pack multiple coefficients into a single $\mathbb{L}_0$ ciphertext. The server will then unpack each coefficient and post-process it into a form that can be used in the protocol (e.g., by using the full replication procedure from [77]) from Section 4.9.3.

All data blocks must also be encoded using the CRT representation. This will be done by the server since the client just sends symmetric-encrypted blocks. With suitable parameters, we can pack $\Theta(\lambda)$ coefficients per ciphertext, reducing the encrypted coefficient size by a factor of $\Theta(\lambda)$.

Packing parameters. Encoding blocks in CRT form and packing coefficients places several additional constraints on how we choose parameters. To maximize the number of coefficients that can be packed into a single ciphertext, we desire that $t \equiv 1 \mod m$ (the scheme is over the m-th cyclotomic polynomial) which is simple to achieve since $t = \Omega(\mathsf{poly}(n))$. To implement the full replication procedure from [77], we require the largest cyclic subgroup of Z_m^* to be as large as possible with respect to m. For example, using $\Phi_m(x) = x^{2^h} + 1$ (the most common setting in FHE literature), 1/4-th of available slots will be usable for rotations which would mean that for that choice of $\Phi_m(x)$, the ORAM scheme's bandwidth blowup has a hidden constant 4. If the cyclotomic polynomial is of prime degree, the hidden constant is 1.

Impact on circuit depth. The full replication procedure can be accomplished in depth $O(\log n)$, which contributes *additively* to the depth of the entire SWHE scheme. As we will see later, $n = \Theta(\lambda)$ therefore the overall depth does not increase asymptotically.

Permuted Buckets to Remove Online Select Operations

By combining our SWHE-based scheme with the permuted bucket optimization from Section 4.7.1, we no longer need to send any encrypted coefficients or perform any homomorphic select operations online. This may greatly improve the latency of the scheme.

With permuted buckets, the blocks read along the tree path are guaranteed to be dummy blocks (encryptions of zero) except for the block of interest. Let L' denote the total ORAM circuit depth. Once the server knows the physical offset of each block b_i for each bucket i, it can first compute $\mathcal{E}_{L'}^{\mathsf{RLWE}}(b_i)$ using Scale[5] and then compute the encrypted block of interest as $\boxplus_i \mathcal{E}_{L'}^{\mathsf{RLWE}}(b_i)$. The client now only sends physical offsets to determine each b_i, which has negligible cost. The PIR write operation from Section 4.7.1 still requires sending $O(Z)$ encrypted coefficients, but importantly it happens *offline*, i.e., after the client receives the block of interest.

4.9.5 Asymptotic Analysis

The SWHE construction we will analyze below uses permuted buckets in Section 4.7.1, but not the Beneš sorting network optimization, as sorting networks will result in non-constant server storage blowup (see below). We do note that the Beneš network does reduce server computation and can be applied if server storage is cheap.

Semi-Honest Case

Chunk size. With BGV SWHE, all ciphertexts returned to the user will be polynomials whose coefficients are in $q_{L'}$ where $q_{L'}/t = \Theta(\mathsf{poly}(n))$. Therefore, to achieve constant bandwidth we require $t = \Omega(\mathsf{poly}(n))$. We note that $n = \Theta(\lambda)$ [40] and that the circuit

[5] In the AHE scheme, dummy blocks are encryptions of 0 but at different layers. There is no operation to make them encryptions of 0 at the same layer.

depth of the entire protocol is $O(\log N) = O(\log \lambda)$. Thus, each coefficient (or set of packed coefficients, if CRT packing is used) is $B'_c = 2n \log n = \Omega(\lambda \log \lambda)$ bits.

Size of select vectors With CRT packing, the user needs $O(\log N)$ ciphertexts to pack the $O(\lambda \log N)$ coefficients (without the Beneš network).

Setting the block size. To absorb the above select vectors, block size is $B' := \Omega(B'_c \log N) = \Omega(\lambda \log^2 \lambda)$.

Server computation In BGV SWHE, multiplying a block with a encrypted coefficient is a polynomial multiplication operation, which has $\widetilde{O}(n \log n \log q)$ computational complexity for $n \log q$-bit ciphertexts. This means the *per-bit* computational overhead is $\widetilde{O}(\log n) = \widetilde{O}(\log \lambda)$, which is much cheaper than that in AHE Onion ORAM. When the block size is small, unpacking will actually become the bottleneck, and its cost is still $\mathsf{polylog}(\lambda)$. Therefore, server computation is $\mathsf{polylog}(\lambda)O(B\lambda \log N) = \mathsf{polylog}(\lambda)O(B\lambda)$.

Client computation Client only needs to decrypt once (no more layers) and therefore has a computational overhead of $\mathsf{polylog}(\lambda)O(B)$.

Malicious Case

Setting the block size. The client needs to download and perform computation on $\Theta(\lambda)$ chunks for each of the $O(\lambda \log N)$ blocks per ORAM access (amortized). Thus, block size becomes $B' := \Omega(B'_c \lambda^2 \log N) = \Omega(\lambda^3 \log^2 \lambda)$.

Client computation Besides decryption, the client has to emulate server computation on the λ verification chunks for $O(\lambda \log N)$ blocks, costing $O(\lambda^2 \log N)\mathsf{polylog}(\lambda) = \widetilde{O}(\lambda^2)$.

Server Storage

The alert reader will have noticed that our scheme over BGV will potentially have a non-constant server storage blowup since $\log q_0 / \log t = \omega(1)$. We now show that server storage blowup is still constant. The intuition is that few buckets (towards the root) has large blowup, and most buckets (towards the leaves) has small or constant blowup. Specifically, server storage blowup is given by:

$$\frac{\sum_{i=0}^{L} 2 \cdot n \cdot \log q_i \cdot 2^i}{\sum_{i=0}^{L} n \cdot \log t \cdot 2^i} = 2 \cdot \sum_{i=0}^{L} \frac{2(L-i+1) \cdot 2^i}{2^{L+1}} = 4 \sum_{i=0}^{L} \frac{(L-i+1)}{2^{L-i+1}} = 4 \sum_{i=1}^{L+1} \frac{i}{2^i} \approx 8$$

which is a constant blowup as desired. In the first equation, the factor of 2 outside the sum is because RLWE ciphertexts are in $R^2_{q_i}$ for each i. The factor of 2 in $2(L-i+1)$ is due to the bound on q_i at level i of the ORAM tree being approximately q_{2i}.

If we were to use a Beneš network, there will be an extra $\log N$ storage blowup at each level of the tree, and the overall server storage blowup is no longer a constant.

4.10 Asymptotic Results for Exponential Security

For reference, we present precise results for exponential security, making no assumptions on parameters, in Table 4.2. The results for negligible in N security in Table 4.1 are derived by setting $\lambda = \omega(\log N)$, $\gamma = \Theta(\log^3 N)$ and $n = \Theta(\lambda)$ according to best known attacks [28, 44].

Table 4.2: **Detailed asymptotics.** N is the number of blocks. The optimal block size is the data block size needed to achieve the stated bandwidth, and is measured in bits. All schemes achieve $O(B)$ client storage and $O(BN)$ server storage (asymptotically optimal) and have $2^{-\lambda}$ failure probability ($\lambda = 80$ is a reasonable value). Computation measures the number of two-input plaintext gates evaluated per ORAM access. "M" stands for malicious security, and "SH" stands for semi-honest. For Path-PIR and AHE Onion ORAM, γ denotes the length of the modulus of the Damgård-Jurik cryptosystem [21]. For SWHE Onion ORAM, n is the degree of the polynomial in the Ring-LWE cryptosystem [40].

Scheme	Optimal Block size B	Bandwidth	Server Computation	Client Computation	Security
Circuit ORAM [5]	$\Omega(\log^2 N)$	$O(B\lambda)$	N/A	N/A	M
Path-PIR [79]	$\Omega(\gamma\lambda \log N)$	$O(B \log N)$	$\widetilde{O}(\gamma)O(B\lambda \log N)$	$\widetilde{O}(\gamma)O(B \log N)$	SH
AHE Onion ORAM	$\Omega(\gamma \log^2 \lambda \log^2 N)$	$O(B)$	$\widetilde{O}(\gamma)O(B\lambda)$	$\widetilde{O}(\gamma)O(B \log \lambda \log N)$	SH
	$\Omega(\gamma\lambda^2 \log \lambda \log N)$	$O(B)$	$\widetilde{O}(\gamma)O(B\lambda)$	$\widetilde{O}(\gamma)O(B \log \lambda \log N)$	M
SWHE Onion ORAM	$\Omega(n \log n \log N)$	$O(B)$	$\mathsf{polylog}(\lambda)O(B\lambda \log N)$	$\widetilde{O}(\log \lambda)O(B)$	SH
	$\Omega(n\lambda^2 \log n \log N)$	$O(B)$	$\mathsf{polylog}(\lambda)O(B\lambda \log N)$	$\widetilde{O}(\log \lambda)O(B) + \widetilde{O}(\lambda^2)$	M

Chapter 5

Tiny ORAM:
A Hardware ORAM Memory Controller

This chapter presents Tiny ORAM, the first hardware ORAM taped-out and validated in silicon. Tiny ORAM is the first hardware ORAM with small client storage, integrity verification, or encryption units. With these attributes, Tiny ORAM can be used by a single-chip secure processor to obfuscate its execution to an adversary watching the chip's I/O pins. As a proof of concept, we evaluate our design as the on-chip memory controller of a 25 core processor. Tiny ORAM design takes up 1/72-th the area (.51 mm^2 of silicon in 32 nm technology) of the chip, which is less than the area of a single core, and consumes 299 mW of power at a 857 MHz clock frequency. With a 128 bits/cycle channel to main memory (roughly equivalent to 2 DRAM channels), Tiny ORAM can complete a 1 GByte non-recursive ORAM lookup for a 512 bit block in $\sim$ 1275 processor cycles. (An insecure DRAM access for a 512 bit block takes an average 58 processor cycles.)

Up to this point, we have focused primarily on the outsourced storage setting. In the secure processor setting, the only implementation-level treatment of ORAM is a system called Phantom, by Maas et al. [66]. In this chapter, we will address the challenges left open by the Phantom design. By the end, we will present a complete silicon tape-out of our design – the first for any type of ORAM – and integrate it with general purpose processor cores to create the first single-chip secure processor able to hide its access pattern to main memory.

What content is covered. This chapter covers a subset of schemes proposed in [69, 86, 87]. Focus is given to techniques that were actually implemented in hardware, or weren't implemented but are particularly relevant to the final hardware design. We will mention when the techniques make a theoretic contribution, but defer the details to the relevant publication.

5.1 Design Challenges for Hardware ORAM

In building Tiny ORAM, we discovered two major challenges areas where new ideas were needed ensure design efficiency and compactness.

5.1.1 Challenge #1: Position Map Management

The first challenge for hardware ORAM controllers is that they need to store and manage the *Position Map* (PosMap for short). Recall from prior chapters: the PosMap is a key-value store that maps data blocks to random locations in external memory. Hence, the PosMap's size is proportional to the number of data blocks (e.g., cache lines) in main memory and can be hundreds of MegaBytes in size. This is too large to fit in a processor's on-chip memory.

PosMap size has been an issue for all prior hardware ORAM proposals. For instance, the Phantom design stores the whole PosMap on-chip. As a result, to scale beyond 1 GByte ORAM capacities, Phantom requires the use of multiple FPGAs *just to store the PosMap*, and thus is not suitable for integration with a single-chip secure processor. On the other hand, Ren et al. [69] (prior collaborative work done by the author) evaluate the *Recursive ORAM* technique (Chapter 2.6.1, [46]) in the secure hardware setting. Recall, the idea is to store the PosMap in additional ORAMs to reduce the on-chip storage requirement. The cost of Recursive ORAM is performance. One must access all the ORAMs in the recursion on each ORAM access. Even after architectural optimizations [69], a Recursive ORAM can spend a majority of its time looking up PosMap ORAMs (see Section 5.4 for a detailed analysis).

We believe that to be practical and scalable to large ORAM capacities, Recursive ORAM is necessary in secure hardware. To that end, one focus in this chapter is to explore novel ways to dramatically reduce the performance overhead of Recursive ORAM.

We then take a new direction and show how our optimized PosMap construction can also, for very little additional cost, be used to perform extremely efficient *integrity verification* for ORAM. Obviously, integrity verification is an important consideration for any secure storage system where data can be tampered with. Yet, prior ORAM integrity schemes based on Merkle trees [68] require large hash unit bandwidth to rate match memory. We show how clever use of our optimized PosMap simplifies this problem dramatically.

5.1.2 Challenge #2: Throughput with Large Memory Bandwidths

The second challenge in designing hardware ORAM is how to maximize data throughput for a given memory (e.g., DRAM) bandwidth. For a given memory bandwidth, the factor limiting data throughput *should* be the memory. Yet, as shown by the Phantom design, this may not be the case because of other factors, described below.

The first bottleneck in Phantom occurs when the application block size is small. Phantom was parameterized for 4 KByte blocks and will incur large pipeline stalls with small blocks (Section 5.6.2). Further, the minimum block size Phantom can support without penalty grows with processor memory bandwidth. While benefiting applications with good data locality, a large block size (like 4 KBytes in Phantom) severely hurts performance for applications with erratic data locality (see Section 5.7.5 for a detailed analysis).[1] We will develop schemes that flexibly support any block size (e.g., we evaluate 64-Byte blocks) without incurring performance loss.

The second bottleneck is that to keep up with an FPGA's large memory bandwidth, an ORAM controller requires many encryption units, imposing large area overheads. This is because in prior art ORAM algorithms, all blocks transferred must be decrypted/re-encrypted, so encryption bandwidth must scale with memory bandwidth. To illustrate the issue, Phantom projects that encryption units alone would take $\sim 50\%$ of the logic of a

[1] Most modern processors have a 64-Byte cache block size for this reason.

state-of-the-art FPGA device. (Although we note that the Phantom design did not actually implement encryption or integrity verification.) We will develop new ORAM schemes that reduce the required encryption bandwidth, and to carefully engineer the system to save hardware area.

5.2 Contributions

To more efficiently manage the PosMap (Challenge #1), we contribute the following mechanisms (Section 5.5):

1. We propose the **PosMap Lookaside Buffer**, or *PLB* for short, a mechanism that significantly reduces the memory bandwidth overhead of Recursive ORAMs depending on underlying *program address locality*.

2. We propose a way to **compress the PosMap**, which reduces the cost of recursion and improves the PLB's effectiveness.

3. We then show how to further use PosMap compression to create an ORAM integrity scheme, called **PosMap MAC** or *PMMAC* for short, which is extremely efficient in practice and is asymptotically optimal.

With the PLB and PosMap compression, we reduce PosMap-related memory bandwidth overhead by 95%, reduce overall ORAM bandwidth overhead by 37% and improve SPEC performance by 1.27×. As a standalone scheme, PMMAC reduces the amount of hashing needed for integrity checking by $\geq 68\times$ relative to prior schemes. Using PosMap compression and PMMAC as a combined scheme, we demonstrate an integrity checking mechanism for ORAM that increases performance overhead by only 7%.

To improve design throughput for high memory bandwidths (Challenge #2), we contribute the following mechanisms (Section 5.6):

1. We propose a **subtree** layout scheme to improve memory bandwidth of tree ORAMs implemented over DRAM.

2. We propose a **bit-based stash management** scheme to enable small block sizes. When implemented in hardware, our scheme removes the block size bottleneck in the Phantom design.

3. We propose a new ORAM scheme called **RAW ORAM**, derived from Ring ORAM in Chapter 3, to reduce the required encryption engine bandwidth.

The subtree layout scheme ensures that over 90% of available DRAM bandwidth is actually used by Tiny ORAM. The stash management scheme prevents a performance bottleneck in Phantom when the block size is small, and allows Tiny ORAM to support any reasonable block size (e.g., from 64-4096 Bytes). In particular: with a 64 Byte block size, Tiny ORAM improves access latency by $\geq 40\times$ in the best case compared to Phantom. On the other hand, RAW ORAM reduces the number of encryption units by $\sim 3\times$ while maintaining comparable bandwidth to the original design.

5.3 Design Prototypes and Availability

In addition to evaluating our proposals using software simulation, we prototype a complete ORAM design with the above mechanisms in FPGA and ASIC (Section 5.8 to end of chapter). Our design is open source and available at http://kwonalbert.github.io/oram.

FPGA results. On the FPGA side, we evaluate the design for performance and area on a Virtex-7 VC707 FPGA board. With the VC707 board's 12.8 GByte/s DRAM bandwidth,[2] Tiny ORAM can complete an access for a 64 Byte block in 1.4μs (also measured on live hardware). This design (with encryption units) requires 5% of the XC7VX485T FPGA's logic and 13% of its on-chip memory.

ASIC results. On the ASIC side, we have taped-out our hardware ORAM in real silicon where it serves as the on-chip memory controller for a 25 core Ascend processor (the author's prior collaborative work [49]). During chip bring-up (January 2016 - February 2017), the chip passed functionality tests, running at 857 MHz and consuming 299 mW of power. **These measurements were taken on live hardware post tape-out**. The entire ORAM controller (parameterized for 2 DRAM channels) requires .51 mm^2, whereas the chip's overall area is 36 mm^2. This prototype proves the feasibility of a single-chip secure processor capable of preventing software IP or data theft through the access pattern to main memory (as described in Section 1).

5.4 Path ORAM Overview (Detailed)

Partly due to its simplicity, our hardware ORAM will be mainly based on the Path ORAM algorithm [71]. Thus, we will now describe that scheme in detail. Path ORAM in hardware is made up of two components: a (trusted) client and (untrusted) external memory or server.

As with other schemes in this thesis, Path ORAM's [46] server storage is logically structured as a binary tree, as shown in Figure 5-1. The ORAM tree's levels range from 0 (the root) to L (the leaves). Each node in the tree is called a *bucket* and has a fixed number of slots (denoted Z) which can store B-bit data blocks. Bucket slots may be empty at any point, and are filled with *dummy blocks*. Non-empty slots contain *real blocks*. All blocks in the tree including dummy blocks are encrypted with a probabilistic encryption scheme (more details given below). Thus, any two blocks (dummy or real) are indistinguishable after encryption. As before, a path is a contiguous sequence of buckets from the root to some leaf l and is referred to as $\mathcal{P}(l)$. $\mathcal{P}(l)$ is uniquely labeled by l.

The client is made up of the *position map*, the *stash* and control logic. The position map, PosMap for short, is a lookup table that associates each data block with a random path in the ORAM tree (see below). If N is the maximum number of real data blocks in the ORAM, the PosMap capacity is $N * L$ bits: one mapping per block. The stash is a memory that temporarily stores up to a small number of data blocks.

Tree ORAM invariant and operation. At any time, each data block in Path ORAM is mapped to a random path via the PosMap. Path ORAM maintains the following invariant: *If a block is mapped to $\mathcal{P}(l)$, then it must be either in some bucket on $\mathcal{P}(l)$ or in the stash.*

[2] Given the FPGA's slower clock frequency, this is roughly equivalent to 8 DRAM channels on an ASIC.

Blocks are stored in the stash or ORAM tree along with labels indicating their path and block address.

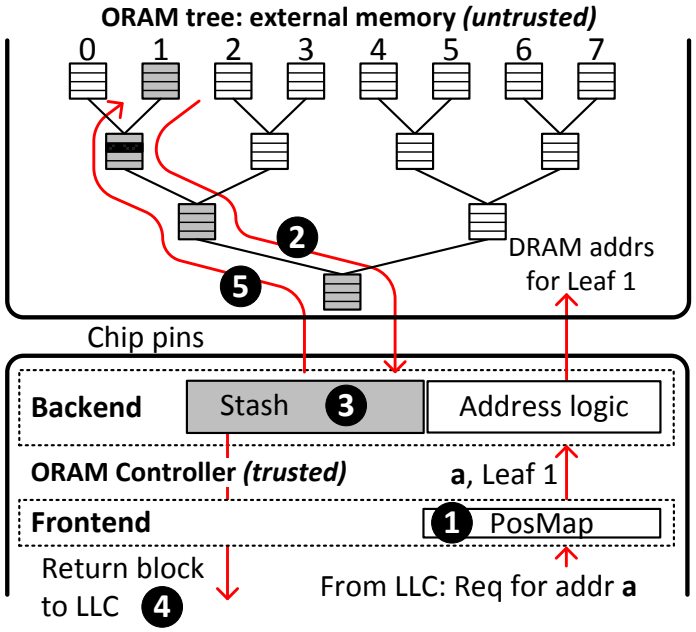


Figure 5-1: A Path ORAM of $L = 3$ levels and $Z = 4$ slots per bucket. Suppose block a, shaded black, is mapped to $\mathcal{P}(1)$. At any time, block a can be located in any of the shaded structures (i.e., on path 1 or in the stash).

To make a request for a block with address a (block a for short), the client calls the function $\mathsf{accessORAM}(a, op, d')$, where op is either read or write and d' is the new data when $op = \mathsf{write}$ (the steps are also shown in Figure 5-1):

1. Look up PosMap with a, yielding the corresponding path label l. Randomly generate a new leaf l' and update the PosMap for a with l'.
2. Read and decrypt all the blocks along path l. Add all the real blocks to the stash (dummies are discarded). Due to the Path ORAM invariant, block a must be in the stash at this point.
3. Update block a in the stash to have leaf l'.
4. If $op = \mathsf{read}$, return block a to the client. If $op = \mathsf{write}$, replace the contents of block a with data d'.
5. Evict and encrypt as many blocks as possible from the stash to $\mathcal{P}(l)$ in the ORAM tree (to keep the stash occupancy low) while keeping the invariant. Fill any remaining space on the path with encrypted dummy blocks.

We refer to Step 1 (the PosMap lookup) as the $\mathsf{Frontend}(a)$, or $\mathsf{Frontend}$, and Steps 2-5 as the $\mathsf{Backend}(a, l, l', op, d')$, or $\mathsf{Backend}$.

Bucket encryption. By default, we encrypt each bucket using a scheme from the author's prior work [69]. For purposes of encryption, we append a counter/initialization vector to

each bucket (referred to as IV). Let $\mathsf{AES}_K(in)$ denote AES encryption with key K and input in. (We will use AES for the rest of the chapter although any private key scheme with semantic security is sufficient.) To encrypt a bucket:

1. $\mathsf{IV} \leftarrow \mathsf{IV} + 1$.[3]

2. Break up the plaintext bits that make up the bucket into 128-bit chunks. To encrypt $chunk_i$, apply the following OTP: $\mathsf{AES}_K(BucketID||\mathsf{IV}||i) \oplus chunk_i$, where $BucketID$ is a unique identifier for each bucket in the ORAM tree.

The encrypted bucket is the concatenation of each chunk along with the IV value in the clear.

Bucket contents. With the encryption scheme above, each bucket is $BucketSize = Z(L + O(L) + B) + |\mathsf{IV}|$ bits in size. Here, L bits denote the path each block is assigned to, $O(L)$ denotes the (encrypted) logical address $addr$ (a for short) for each block, and $|\mathsf{IV}|$ refers to the IV bit length. For practical deployments, we assume $|\mathsf{IV}| = 64$ to avoid overflow. In practice $O(L) = L + 2$ bits for $Z = 4$, which means 50% of the DRAM can be used to store real blocks. Conceptually, we can reserve a unique logical address $\perp$ to mark a bucket slot as containing a dummy block.[4]

ORAM initialization. One may initialize ORAM simply by zeroing-out main memory. In that case, AES units performing bucket decryption should treat the bucket as fully empty when the IV equals 0. Our actual implementation uses this method. The downside of this scheme is that it requires $O(N)$ work upfront, for every program execution. A 'lazy initialization' scheme which gradually initializes each bit of memory as it is accessed the first time is described in [69].

Security. The intuition for Path ORAM's security is that every PosMap lookup (Step 1) will yield a fresh random leaf to access the ORAM tree for that access. This makes the sequence of ORAM tree paths accessed independent of the actual program address trace. Probabilistic encryption hides *which* block is accessed on the path. Further, stash overflow probability is negligible if $Z \geq 4$ (proven for $Z \geq 5$ [71] and shown experimentally for $Z = 4$ [66]).

Security in the presence of caches. Basic Path ORAM (augmented with a Merkle Tree for integrity verification [69]) satisfies Definition 2. We don't consider it problematic that hardware ORAM is only invoked on LLC misses. For example, the environment $\mathcal{Z}$ may be the program. In that case, the LLC "filters out" a $\mathcal{Z}$-dependent subset of accesses to ORAM, which breaks Definition 2. To fix the issue, we take $\mathcal{Z}$ to be *the program plus the LLC*. In both cases, a function of the program's behavior is revealed through the ORAM access count, which leaks a logarithmic number of bits in time.

[3] Note that since the counter always increments by one, it is important for the ORAM controller to use a different user session key K during each run, to avoid a replay attack.

[4] Our actual marks dummies with Z extra 'valid' bits per bucket.

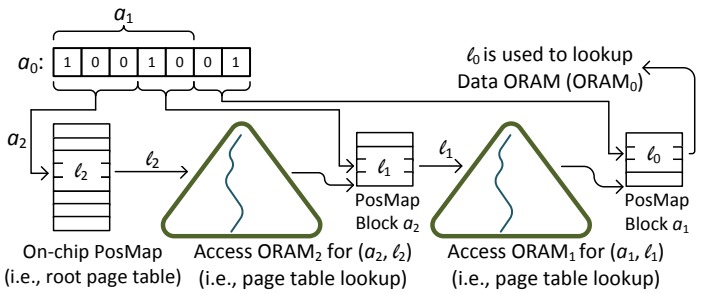


Figure 5-2: Recursive ORAM with PosMap block sizes $X = 4$, making an access to the data block with program address $a_0 = 1001001_2$. Recursion shrinks the PosMap capacity from $N = 128$ to 8 entries.

5.4.1 Recursive ORAM

As mentioned in previous sections, the number of entries in the PosMap scales linearly with the number of data blocks in the ORAM. In the secure processor setting, this results in a significant amount of on-chip storage (hundreds of KiloBytes to hundreds of MegaBytes). To address this issue, Shi et al. [46] (Section 2.6.1) proposed a scheme called *Recursive ORAM*, which has been studied in simulation for trusted hardware proposals in the author's prior work [69]. The basic idea is to store the PosMap in a separate ORAM, and store the new ORAM's (smaller) PosMap on-chip. *We make an important observation that the mechanics of Recursive ORAM are remarkably similar to multi-level page tables in traditional virtual memory systems.* We use this observation to help explain ideas and derive optimizations.

We explain Recursive ORAM through the example in Figure 5-2, which uses two levels of recursion. The system now contains 3 separate ORAM trees: the *Data ORAM*, denoted as ORam_0, and two *PosMap ORAMs*, denoted ORam_1 and ORam_2. Blocks in the PosMap ORAMs are akin to page tables. We say that PosMap blocks in ORam_i store X leaf labels which refer to X blocks in ORam_{i-1}. This is akin to having X pointers to the next level page table and X is a parameter.[5]

Suppose the LLC requests block a_0, stored in ORam_0. The leaf label l_0 for block a_0 is stored in PosMap block $a_1 = a_0/X$ of ORam_1 (all division is floored). Like a page table, block a_1 stores leaves for neighboring data blocks (i.e., $\{a_0, a_0 + 1, \ldots, a_0 + X - 1\}$ in the case where a_0 is a multiple of X). The leaf l_1 for block a_1 is stored in the block $a_2 = a_0/X^2$ stored in ORam_2. Finally, leaf l_2 for PosMap block a_2 is stored in the on-chip PosMap. The on-chip PosMap is now akin to the root page table, e.g., register CR3 on X86 systems.

To make a Data ORAM access, we must first lookup the on-chip PosMap, ORam_2 and ORam_1 in that order. Thus, a Recursive ORAM access is akin to a full page table walk. Additional PosMap ORAMs ($\mathsf{ORam}_3, \mathsf{ORam}_4, \ldots, \mathsf{ORam}_{H-1}$) may be added as needed to shrink the on-chip PosMap further. H denotes the total number of ORAMs including the Data ORAM in the recursion and $H = \log(N/p)/\log X + 1$ if p is the number of entries in the on-chip PosMap.

[5] Generally, each PosMap level can have a different X. We assume the same X for all PosMaps for simplicity.

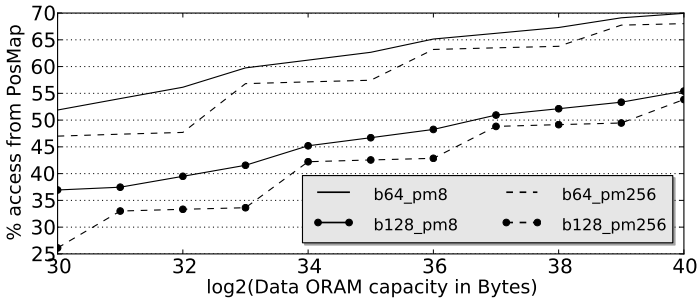


Figure 5-3: The percentage of Bytes read from PosMap ORAMs in a full Recursive ORAM access for $X = 8$ (following [69]) and $Z = 4$. All bucket sizes are padded to 512 bits to estimate the effect in DDR3 DRAM. The notation b64_pm8 means the ORAM block size is 64 Bytes and the on-chip PosMap is at most 8 KB.

5.4.2 Overhead of Recursion

It should now be clear that Recursive ORAM increases total ORAM access latency. Counter-intuitively, with small block sizes, PosMap ORAMs can contribute to more than half of the total ORAM latency as shown in Figure 5-3. For a 4 GB Data ORAM capacity, 39% and 56% of bandwidth is spent on looking up PosMap ORAMs (depending on block size), and increasing the on-chip PosMap only slightly dampens the effect. Abrupt kinks in the graph indicate when another PosMap ORAM is added (i.e., when H increases).

We now explain this overhead from an asymptotic perspective. We know a single Path ORAM (without recursion) with a block size of B bits transfers $O(B \log N)$ bits per access. In Recursive ORAM, the best strategy to minimize bandwidth is to set X to be constant, resulting in a PosMap ORAM block size of $B_p = \Theta(\log N)$. Then, the number of PosMap ORAMs needed is $\Theta(\log N)$, and the resulting bandwidth overhead becomes

$$O\left(\log N + \frac{H B_p \log N}{B}\right) = O\left(\log N + \frac{\log^3 N}{B}\right)$$

The first term is for Data ORAM and the second term accounts for all PosMap ORAMs combined. In realistic processor settings, $\log N \approx 25$ and data block size $B \approx \log^2 N$ (512 or 1024 in Figure 5-3). Thus, it is natural that PosMap ORAMs account for roughly half of the bandwidth overhead.

In the next section, we show how insights from traditional virtual memory systems, coupled with security mechanisms, can dramatically reduce this PosMap ORAM overhead (Section 5.5).

5.5 Frontend

We now present several mechanisms to optimize the PosMap. The techniques in this section only impact the Frontend (Section 5.4) and can be applied to any Position-based ORAM Backend (such as [46, 56, 61]).

5.5.1 PosMap Lookaside Buffer

Given our understanding of Recursive ORAM as a multi-level page table for ORAM (Section 5.4.1), a natural optimization is to cache PosMap blocks (i.e., page tables) so that LLC accesses exhibiting program address locality require less PosMap ORAM accesses on average. This idea is the essence of the *PosMap Lookaside Buffer*, or PLB, whose name obviously originates from the Translation Lookaside Buffer (TLB) in conventional systems. Unfortunately, unless care is taken, this idea totally breaks the security of ORAM. This section develops the scheme and fixes the security holes.

High-level Idea and Ingredients

PLB Caches. The key point from Section 5.4.1 is that blocks in PosMap ORAMs contain a set of leaf labels for *consecutive blocks* in the next ORAM. Given this fact, we can eliminate some PosMap ORAM lookups by adding a hardware cache to the ORAM Frontend called the PLB. Suppose the LLC requests block a_0 at some point. Recall from Section 5.4.1 that the PosMap block needed from $\mathsf{ORam_i}$ for a_0 has address $a_i = a_0/X^i$. If this PosMap block is in the PLB when block a_0 is requested, the ORAM controller has the leaf needed to lookup $\mathsf{ORam_{i-1}}$, and can skip $\mathsf{ORam_i}$ *and all the smaller PosMap ORAMs*. Otherwise, block a_i is retrieved from $\mathsf{ORam_i}$ and added to the PLB. When block a_i is added to the PLB, another block may have to be evicted in which case it is appended to the stash of the corresponding ORAM.

A minor but important detail is that a_i may be a valid address for blocks in multiple PosMap ORAMs; to disambiguate blocks in the PLB, block a_i is stored with the tag $i||a_i$ where $||$ denotes bit concatenation.

PLB (In)security. Unfortunately, since each PosMap ORAM is stored in a different physical ORAM tree and PLB hits/misses correlate directly to a program's access pattern, the PosMap ORAM access *sequence* leaks the program's access pattern. To show how this breaks security, consider two example programs in a system with one PosMap ORAM $\mathsf{ORam_1}$ (whose blocks store $X = 4$ leaves) and a Data ORAM $\mathsf{ORam_0}$. Program A unit strides through memory (e.g., touches $a, a+1, a+2, \ldots$). Program B scans memory with a stride of X (e.g., touches $a, a+X, a+2X, \ldots$). For simplicity, both programs make the same number of memory accesses. Without the PLB, both programs generate the same access sequence, namely: $1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \ldots$ where 0 denotes an access to $\mathsf{ORam_0}$, and 1 denotes an access to $\mathsf{ORam_1}$. However, with the PLB, the adversary sees the following access sequences (**0** denotes an access to $\mathsf{ORam_0}$ on a PLB hit):

$$\text{Program A} : 1, 0, \mathbf{0}, \mathbf{0}, \mathbf{0}, 1, 0, \mathbf{0}, \mathbf{0}, \mathbf{0}, 1, 0, \ldots$$
$$\text{Program B} : 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \ldots$$

Program B constantly misses in the PLB and needs to access $\mathsf{ORam_1}$ on every access. Clearly, the adversary can tell program A apart from program B in the PLB-enabled system.

Security Fix: Unified ORAM Tree. To hide PosMap access sequence, we will change Recursive ORAM such that *all PosMap ORAMs and the Data ORAM store blocks in the same physical tree* which we denote $\mathsf{ORam_U}$. Organizationally, the PLB and on-chip PosMap become the new Path ORAM Frontend, which interacts with a single ORAM Backend (Sec-

tion 5.4). Security-wise, both programs from the previous section access *only* $\mathsf{ORam_U}$ with the PLB and the adversary cannot tell them apart (see the end of this sub-section for more discussion on security).

TLB vs. PLB. We remark that while a traditional TLB caches *single address translations*, the PLB caches entire PosMap blocks (akin to whole page tables). The address locality exploited by both structures, however, is the same.

Detailed Construction

Blocks Stored in $\mathsf{ORam_U}$. Data blocks and the PosMap blocks originally from the PosMap ORAMs (i.e., $\mathsf{ORam_1}$,...,ORam_{H-1}) are now stored in a single ORAM tree ($\mathsf{ORam_U}$) and all accesses are made to this one ORAM tree. Both data and PosMap blocks now have the same size. Since the number of blocks that used to be stored in some $\mathsf{ORam_i}$ ($i > 0$) is X times smaller than the number of blocks stored in $\mathsf{ORam_{i-1}}$, storing PosMap blocks alongside data blocks adds at most one level to $\mathsf{ORam_U}$.

Each set of PosMap blocks must occupy a disjoint address space so that they can be disambiguated. For this purpose we apply the following addressing scheme: Given data block a_0, the address for the PosMap block originally stored in $\mathsf{ORam_i}$ for block a_0 is given by $i||a_i$, where $a_i = a_0/X^i$. This address $i||a_i$ is used to fetch the PosMap block from the $\mathsf{ORam_U}$ and to lookup the PosMap block in the PLB. To simplify the notation, we don't show the concatenated address $i||a_i$ in future sections and just call this block a_i.

ORAM readrmv and append Operations. We use two new flavors of ORAM access to support PLB refills/evictions (i.e., *op* in Section 5.4): read-remove and append. The idea of these two type of accesses appeared in [69] but we describe them in more detail below. Read-remove (readrmv) is the same as read except that it physically deletes the block in the stash after it is forwarded to the ORAM Frontend. Append (append) adds a block to the stash without performing an ORAM tree access. $\mathsf{ORam_U}$ must not contain duplicate blocks: only blocks that are currently not in the ORAM (possibly read-removed previously) can be appended. Further, when a block is appended, the current leaf it is mapped to in $\mathsf{ORam_U}$ must be known so that the block can be written back to the ORAM tree during later ORAM accesses.

PLB Architecture. The PLB is a conventional hardware cache that stores PosMap blocks. Each PosMap block is tagged with its block address a_i. On a hit, one of the X leaves in the block is read out and remapped. PosMap blocks are read-removed and appended from/to $\mathsf{ORam_U}$. Thus, each block is stored in the PLB alongside its *current leaf*. The PLB itself has normal cache parameters (size, associativity), and we explore how this space impacts performance in Section 5.7.3.

ORAM Access Algorithm

The steps to read/write a data block with address a_0 are given below (shown pictorially in Figure 5-4):

1. **(PLB lookup)** For $i = 0, \ldots, H-2$, look up the PLB for the leaf of block a_i (contained in block a_{i+1}). If one access hits, save i and go to Step 2; else, continue. If no access

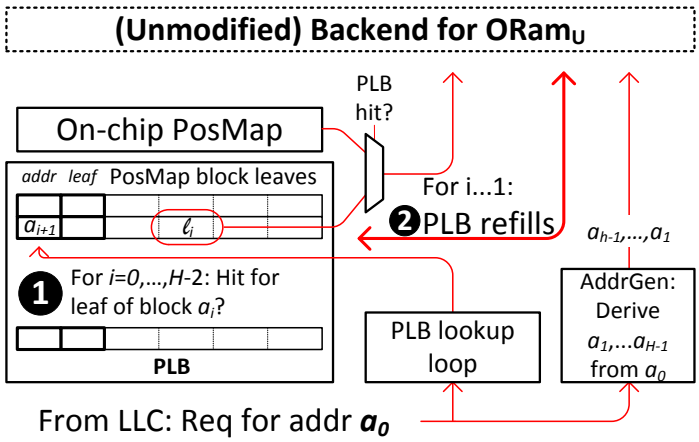


Figure 5-4: PLB-enabled ORAM Frontend with $X = 4$. Accessing the actual data block a_0 (Step 3 in Section 5.5.1) is not shown.

hits for $i = 0, \ldots, H-2$, look up the on-chip PosMap for the leaf of block a_{H-1} and save $i = H - 1$.

2. **(PosMap block accesses)** While $i \geq 1$, perform a readrmv operation to ORam$_U$ for block a_i and add that block to the PLB. If this evicts another PosMap block from the PLB, append that block to the stash. Decrement i. (This loop will not be entered if $i = 0$.)

3. **(Data block access)** Perform an ordinary read or write access to ORam$_U$ for block a_0.

Importantly, aside from adding support for readrmv and append, the above algorithm requires no change to the ORAM Backend.

5.5.2 PosMap Compression

We now show how to compress the PosMap using pseudorandom functions (PRFs, introduced below). The high level goal is to store *more leaves per PosMap block*, thereby reducing the number of Recursive PosMaps.

This scheme by itself does not dramatically improve performance (nor did we implement it in hardware). We include it for completeness, and because it improves the PLB scheme from Section 5.5.1, and helps motivate the integrity scheme presented in Section 5.5.3.

Background: PRFs

A pseudorandom Function, or PRF, family $y = \mathsf{PRF}_K(x)$ is a collection of efficiently-computable functions, where K is a random secret key. A PRF guarantees that anyone who does not know K (even given x) cannot distinguish y from a truly random bit-string in polynomial time with non-negligible probability [8]. For the rest of the chapter, we implement $\mathsf{PRF}_K()$ using AES-128.

Construction

Main Idea. Following previous notation, suppose each PosMap block contains X leaf labels for the next ORAM. For example, some PosMap block contains leaf labels for the blocks with addresses $\{a,\ a+1,\ \cdots,\ a+X-1\}$. With the compressed PosMap scheme, the PosMap block's contents are replaced with an α-bit group counter (GC) and X β-bit individual counters (IC):

$$GC \ || \ IC_0 \ || \ IC_1 \ || \ IC_2 \ || \ \cdots \ || \ IC_{X-1}$$

With this format, we can then compute the current leaf label for block $a+j$ through $\mathsf{PRF}_K(a+j||GC||IC_j) \bmod 2^L$. Note that with this technique, the on-chip PosMap is unchanged and still stores an uncompressed leaf per entry.

Block Remap. For $\mathsf{PRF}_K()$ to generate a uniform random sequence of leaves, we must ensure that each $GC||IC_j$ *strictly increases* (i.e., the $\mathsf{PRF}_K()$ must never see the same input twice). This is achieved by the following modified remapping operation:

When remapping block $a+j$, the ORAM controller first increments its individual counter IC_j. If the individual counter rolls over (becomes zero again), the ORAM controller will increment the group counter GC. This changes the leaf label for all the blocks in the group, so we have to read each block through the Backend, reset its individual counter and remap it to the updated path given by $\mathsf{PRF}_K(a+j||GC+1||0) \bmod 2^L$. In the worst-case where the program always requests the same block in a group, we need to reset X individual counters in the group every 2^β accesses.

We remark that this reset operation is very expensive for baseline Recursive ORAM (Section 5.4.1). In that case, the ORAM controller must make X full Recursive ORAM accesses to reset the individual counters in *a certain* PosMap ORAM block. Otherwise, it reveals that individual counters have overflown in that certain ORAM, which is related to the access pattern. On the other hand, using a single Unified ORAM tree as we do to support the PLB (Section 5.5.1) reduces this to X accesses to $\mathsf{ORam_U}$.

System Impact and the PLB. The compressed PosMap format can be used with or without a PLB and, like the PLB, does not require changes to the Backend. That is, PosMap blocks are stored in their compressed format inside the PLB and ORAM tree/Backend. Uncompressed leaves are generated using the PRF on-demand by the Frontend. Each block stored in the Backend or ORAM tree is still stored alongside its uncompressed leaf label (a one time cost per block), to facilitate ORAM evictions.

Benefit of Compressed Format

Our scheme *compresses* the PosMap block by setting α, β and X such that $\alpha/X + \beta < L$, implying that the (amortized) bits needed to store each leaf has decreased. A larger X means a fewer number of PosMap ORAMs are needed as discussed in Section 5.4.1. Further, this scheme improves the PLB's hit rate (Section 5.5.1) since more blocks are associated with a given PosMap block.

For concreteness, suppose the ORAM block size in bits is $B = 512$. The compressed PosMap scheme enables $X' = 32$ by setting $\alpha = 64$ and $\beta = 14$, regardless of ORAM tree

depth L.[6] In this configuration, the worst case block remap overhead is $X'/2^{\beta} = .2\%$. By comparison, the original PosMap representation (up to Section 5.5.1) only achieves $X = 16$ for ORAM tree depths of $L = 17$ to $L = 32$.

Theoretic improvement. The compressed PosMap also improves recursive Path ORAM (and Ring ORAM) bandwidth asymptotically by a $\log \log N$ factor for certain block sizes. Details can be found in [86], Section 5.4.

5.5.3 PosMap MAC

We now describe a novel and simple integrity verification scheme for ORAM called *PosMap MAC*, or *PMMAC*, that is facilitated by our PosMap compression technique from the previous section. PMMAC achieves asymptotic improvements in hash bandwidth over prior schemes and is easy to implement in hardware.

Background: MACs

Suppose two parties Alice and Bob share a secret K and Alice wishes to send messages to Bob over an insecure channel where data packets d_i $(i = 0, \dots)$ can be tampered by some adversary Eve. To guarantee message *authenticity*, Alice can send Bob tuples (h_i, d_i) where $h_i = \mathsf{MAC}_K(d_i)$ and $\mathsf{MAC}_K()$ is a Message Authentication Code (e.g., a keyed hash function [12]). For the rest of the chapter, we implement $\mathsf{MAC}_K()$ using SHA3-224.

The MAC scheme guarantees that Eve can only produce a message forgery $(h\star, d\star)$ with negligible probability, where $h\star = \mathsf{MAC}_K(d\star)$ and $d\star$ was not transmitted previously by Alice. In other words, without knowing K, Eve cannot come up with a forgery for a message whose MAC it has never seen.

Importantly, Eve can still perform a replay attack, violating *freshness*, by replacing some (h_i, d_i) with a previous legitimate message (h_j, d_j). A common fix for this problem is to embed a *non-repeating counter* in each MAC [32]. Suppose Alice and Bob have shared access to an oracle that, when queried, returns the number of messages sent by Alice but not yet checked by Bob. Then, for message i, Alice transmits (h'_i, d_i) where $h'_i = \mathsf{MAC}_K(i||d_i)$. Eve can no longer replay an old packet (h'_j, d_j) because $\mathsf{MAC}_K(i||d_j) \neq \mathsf{MAC}_K(j||d_j)$ with overwhelming probability. The challenge in implementing these schemes is that Alice and Bob must have access to a shared, *tamper-proof* counter.

Main Idea and Non-Recursive PMMAC

Clearly, any memory system including ORAM that requires integrity verification can implement the replay-resistant MAC scheme from the previous section by storing per-block counters in a tamper-proof memory. Unfortunately, the size of this memory is even larger than the original ORAM PosMap making the scheme untenable. *We make a key observation that if PosMap entries are represented as non-repeating counters, as is the case with the compressed PosMap (Section 5.5.2), we can implement the replay-resistant MAC scheme without additional counter storage.*

We first describe PMMAC without recursion and with simple/flat counters per-block to illustrate ideas. Suppose block a which has data d has access count c. Then, the on-chip PosMap entry for block a is c and we generate the leaf l for block a through

[6] We restrict X' to be a power of two to simplify the PosMap block address translation from Section 5.4.1.

$l = \mathsf{PRF}_K(a||c) \bmod 2^L$ (i.e., same idea as Section 5.5.2). Block a is written to the Backend as the tuple (h, d) where

$$h = \mathsf{MAC}_K(c \;||\; a \;||\; d)$$

When block a is read, the Backend returns $(h\star, d\star)$ and PMMAC performs the following check to verify authenticity/freshness:

$$\text{assert } h\star == \mathsf{MAC}_K(c \;||\; a \;||\; d\star)$$

where $\star$ denotes values that may have been tampered with. After the assertion is checked, c is incremented for the returned block.

Security follows if it is infeasible to tamper with block counters and no counter value for a given block is ever repeated. The first condition is clearly satisfied because the counters are stored on-chip. We can satisfy the second condition by making each counter is wide enough to not overflow (e.g., 64 bits wide).

As with our previous mechanisms, PMMAC requires no change to the ORAM Backend because the MAC is treated as extra bits appended to the original data block.[7] As with PosMap compression, the leaf currently associated with each block in the stash/ORAM tree is stored in its original (uncompressed) format.

Adding Recursion and PosMap Compression

To support recursion, PosMap blocks (including on-chip PosMap entries) may contain either a flat (64 bits) or compressed counter (Section 5.5.2) per next-level PosMap or Data ORAM block. As in the non-Recursive ORAM case, all leaves are generated via a PRF. The intuition for security is that the tamper-proof counters in the on-chip PosMap form the root of trust and then recursively, the PosMap blocks become the root of trust for the next level PosMap or Data ORAM blocks. Note that in the compressed scheme, the α and β components of each counter are already sized so that each block's count never repeats/overflows. We give a formal analysis for security with Recursive ORAM in Section 5.5.4.2.

For realistic parameters, the scheme that uses flat counters in PosMap blocks incurs additional levels of recursion. For example, using $B = 512$ and 64 bit counters we have $X = B/64 = 8$. *Importantly, with the compressed PosMap scheme we can derive each block counter from GC and IC_j (Section 5.5.2) without adding levels of recursion or extra counter storage.*

Key Advantage: Hash Bandwidth and Parallelism

Combined with PosMap compression, the overheads for PMMAC are the bits added to each block to store MACs and the cost to perform cryptographic hashes on blocks. The extra bits per block are relatively low-overhead — the ORAM block size is usually 64-128 Bytes and each MAC may be 80-128 bits depending on the security parameter.

To perform a non-Recursive ORAM access (i.e., read/write a single path), Path ORAM reads/writes $O(\log N)$ blocks from external memory. Merkle tree constructions [68, 72] need to integrity verify all the blocks on the path to check/update the root hash. Crucially, our PMMAC construction only needs to integrity verify (check and update) 1 block — namely the block of interest — per access, achieving an asympotic reduction in hash bandwidth.

[7] That is, the MAC is encrypted along with the block when it is written to the ORAM tree.

To give some concrete numbers, assume $Z = 4$ block slots per ORAM tree bucket following [71, 66]. Then, there are $Z * (L + 1)$ blocks per path in ORAM tree, and our construction reduces hash bandwidth by $68\times$ for $L = 16$ and by $132\times$ for $L = 32$. We did not include the cost of reading sibling hashes for the Merkle tree for simplicity.

Integrity verifying only a single block also prevents a serialization bottleneck present in Merkle tree schemes. Consider the scheme from [68], a scheme optimized for Path ORAM. Each hash in the Merkle tree node must be recomputed based on the contents of the corresponding ORAM tree bucket and *its child hashes*, and is therefore fundamentally sequential. If this process cannot keep up with memory bandwidth, it will be the system's performance bottleneck.

Adding Encryption: Subtle Attacks and Defenses

Up to this point we have discussed PMMAC in the context of providing integrity only. ORAM must also apply a probabilistic encryption scheme (we assume AES counter mode as done in [69]) to all data stored in the ORAM tree. In this section we first show how the encryption scheme of [69] breaks under active adversaries because the adversary is able to *replay the one-time pads used for encryption.* ([69] presented an integrity verification scheme based on Merkle trees to prevent such attacks.) We show how PMMAC doesn't prevent this attack by default and then provide a fix that applies to PMMAC.

We first show the scheme used by [69] for reference: Each bucket in the ORAM tree contains, in addition to Z encrypted blocks, a seed used for encryption (the $BucketSeed$) that is stored in plaintext. ($BucketSeed$ is synonymous to the "counter" in AES counter mode.) If the Backend reads some bucket (Step 2 in Section 5.4) whose seed is $BucketSeed$, the bucket will be re-encrypted and written back to the ORAM tree using the one-time pad (OTP) $\mathsf{AES}_K(BucketID||BucketSeed + 1||i)$, where i is the current chunk of the bucket being encrypted.

The above encryption scheme breaks privacy under PMMAC because PMMAC doesn't integrity verify $BucketSeed$. For a bucket currently encrypted with the pad $P = \mathsf{AES}_K(BucketID||BucketSeed||i)$, suppose the adversary replaces the plaintext bucket seed to $BucketSeed - 1$. This modification will cause the contents of that bucket to decrypt to garbage, *but won't trigger an integrity violation under PMMAC unless bucket $BucketID$ contains the block of interest for the current access.* If an integrity violation is not triggered, due to the replay of $BucketSeed$, that bucket will next be encrypted using the *same one-time pad* P again.

Replaying one-time pads obviously causes security problems. If a bucket re-encrypted with the same pad P contains plaintext data D at some point and D' at another point, the adversary learns $D \oplus D'$. If D is known to the adversary, the adversary immediately learns D' (i.e., the plaintext contents of the bucket).

The fix for this problem is relatively simple: To encrypt chunk i of a bucket about to be written to DRAM, we will use the pad $\mathsf{AES}_K(GlobalSeed||i)$, where $GlobalSeed$ is now a single monotonically increasing counter stored in the ORAM controller in a dedicated register (this is similar to the global counter scheme in [34]). When a bucket is encrypted, the current $GlobalSeed$ is written out alongside the bucket as before and $GlobalSeed$ (in the ORAM controller) is incremented. Now it's easy to see that each bucket will always be encrypted with a fresh OTP which defeats the above attack.

5.5.4 Security Analysis

We now give a security analysis for the PLB and PMMAC schemes.

5.5.4.1 PosMap Lookaside Buffer

We now give a proof sketch that our PLB+Unified ORAM tree construction achieves satisfies Definition 2. To do this, we use the fact that the PLB interacts with a normal Path ORAM Backend. We make the following observations, which we will use to argue security:

Observation 3. *If all leaf labels l_i used in* {read, write, readrmv} *calls to* Backend *are random and independent of other l_j for $i \neq j$, the* Backend *achieves the security of the original Path ORAM (Section 5.4).*

Observation 4. *If an* append *is always preceded by a* readrmv*, stash overflow probability does not increase (since the net stash occupancy is unchanged after both operations).*

Theorem 3. *The PLB+Unified ORAM tree scheme reduces to the security of the ORAM* Backend*.*

Proof: The PLB+Unified ORAM Frontend calls Backend in two cases: First, if there is a PLB hit the Backend request is for a PosMap or Data block. In this case, the leaf l sent to Backend was in a PosMap block stored in the PLB. Second, if all PLB lookups miss, the leaf l comes from the on-chip PosMap. In both cases, leaf l was remapped the instant the block was last accessed. We conclude that all {read, write, readrmv} commands to Backend are to random/independent leaves and Observation 3 applies. Further, an append command can only be caused by a PLB refill which is the result of a readrmv operation. Thus, Observation 4 applies. ■

Of course, the PLB may further influence the ORAM trace *length* (the number of calls to Access for a given $\mathcal{Z}$ in Section 2.2) by filtering out some calls to Backend for PosMap blocks. Now the trace length is determined by, and thus reveals, the sum of LLC misses and PLB misses. We reiterate that processor cache and the PLB are both on-chip and outside the ORAM Backend, so adding a PLB is the same (security-wise) to adding more processor cache: in both cases, only the total number of ORAM accesses leaks. By comparison, using a PLB without a Unified ORAM tree leaks the *set* of PosMap ORAMs needed on every Recursive ORAM access, which makes leakage grow linearly with the trace length.

5.5.4.2 PosMap MAC (Integrity)

We show that breaking our integrity verification scheme is as hard as breaking the underlying MAC. Thus, the scheme attains the integrity definition from Section 2.3. First, we have the following observation:

Observation 5. *If the first $k-1$ address and counter pairs (a_i, c_i)'s the* Frontend *receives have not been tampered with, then the* Frontend *seeds a MAC using a unique (a_k, c_k), i.e., $(a_i, c_i) \neq (a_k, c_k)$ for $1 \leq i < k$. This further implies $(a_i, c_i) \neq (a_j, c_j)$ for all $1 \leq i < j \leq k$.*

This property can be seen directly from the algorithm description, with or without the PLB and/or PosMap compression. For every a, we have a dedicated counter, sourced from the on-chip PosMap or the PLB, that increments on each access. If we use PosMap compression, each block counter will either increment (on a normal access) or jump to the

next multiple of the group counter in the event of a group remap operation. Thus, each address and counter pair will be different from previous ones. We now use Observation 5 to prove the security of our integrity scheme.

Theorem 4. *Breaking the PMMAC scheme is as hard as breaking the underlying MAC scheme.*

Proof: We proceed via induction on the number of accesses. In the first ORAM access, the Frontend uses (a_1, c_1), to call Backend for (h_1, d_1) where $h_1 = \mathsf{MAC}_K(c_1||a_1||d_1)$. (a_1, c_1) is unique since there are no previous $(a_i, c_i)'s$. Note that a_1 and c_1 cannot be tampered with since they come from the Frontend. Thus, producing a forgery (h'_1, d'_1) where $d'_1 \neq d_1$ and $h'_1 = \mathsf{MAC}_K(c_1||a_1||d'_1)$ is as hard as breaking the underlying MAC. Suppose no integrity violation has happened and Theorem 4 holds up to access $n-1$. Then, the Frontend sees fresh and authentic (a_i, c_i)'s for $1 \leq i \leq n-1$. By Observation 5, (a_n, c_n) will be unique and $(a_i, c_i) \neq (a_j, c_j)$ for all $1 \leq i < j \leq n$. This means the adversary cannot perform a replay attack (Section 5.5.3) because all (a_i, c_i)'s are distinct from each other and are tamper-proof. It is also hard to generate a valid MAC with unauthentic data without the secret key. Being able to produce a forgery (h'_i, d'_i) where $d'_i \neq d_i$ and $h'_i = \mathsf{MAC}_K(c_i||a_i||d'_i)$ means the adversary can break the underlying MAC. ■

5.5.4.3 PosMap MAC (Privacy)

The system's privacy guarantees require certain assumptions under PMMAC because PMMAC is an *authenticate-then-encrypt* scheme [22]. Since the integrity verifier only checks the block of interest returned to the Frontend, other (tampered) data on the ORAM tree path will be written to the stash and later be written back to the ORAM tree. For example, if the adversary tampers with the block-of-interest's address bits, the Backend won't recognize the block and won't be able to send any data to the integrity verifier (clearly an error). The adversary may also coerce a stash overflow by replacing dummy blocks with real blocks or duplicate blocks along a path.

To address these cases, we have to make certain assumptions about how the Backend will possibly behave in the presence of tampered data. We require a correct implementation of the ORAM Backend to have the following property:

Property 1. *If the* Backend *makes an ORAM access, it only reveals to the adversary (a) the leaf sent by the* Frontend *for that access and (b) a fixed amount of encrypted data to be written back to the ORAM tree.*

If Property 1 is satisfied, it is straightforward to see that any memory request address trace generated by the Backend is indistinguishable from other traces of the same length. That is, the Frontend receives tamper-proof responses (by Theorem 4) and therefore produces independent and random leaves. Further, the global seed scheme in Section 5.5.3 trivially guarantees that the data written back to memory gets a fresh pad.

If Property 1 is satisfied, the system can still leak the ORAM request trace *length*; i.e., *when* an integrity violation is detected, or when the Backend enters an illegal state. Conceptually, an integrity violation generates an exception that can be handled by the processor. When that exception is generated and how it is handled can leak some privacy. For example, depending on how the adversary tampered with memory, the violation may be detected immediately or after some period of time depending on whether the tampered bits were of interest to the Frontend. Quantifying this leakage is outside our scope, but we

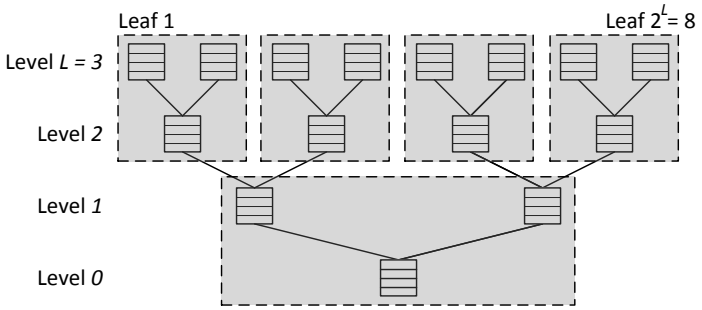


Figure 5-5: Illustration of subtree locality.

remark that this level of security matches our combined privacy+integrity definition from Section 2.3.

5.6 Backend

We now present several mechanisms to improve the ORAM Backend's throughput when memory bandwidth is high. The techniques in this section only impact the Backend (Section 5.4) and can be applied regardless of optimizations from the previous section.

5.6.1 Building Tree ORAMs on DRAM

To start, we propose a simple technique to improve memory throughput for Tree ORAMs, like Path ORAM, implemented over DRAM. The issue is that to be secure, ORAM accesses inherently have low spatial locality in memory. Yet, achievable throughput in DRAM depends on spatial locality: bad spatial locality means more DRAM row buffer misses which means time delay between consecutive accesses. (We will assume an open page policy on DRAM for the rest of the thesis.) Indeed, when naïvely storing the Path ORAM tree into an array, two consecutive buckets along the same path hardly have any locality, and it can be expected that row buffer hit rate would be low. The following technique can improve Path ORAM's performance on DRAM.

We pack each subtree with k levels together, and treat them as the nodes of a new tree, a 2^k-ary tree with $\lceil \frac{L+1}{k} \rceil$ levels. Figure 5-5 is an example with $k = 2$. We adopt the address mapping scheme in which adjacent addresses first differ in channels, then columns, then banks, and lastly rows. We set the node size of the new tree to be the row buffer size times the number of channels, which together with the original bucket size determines k.

Performance impact. With commercial DRAM DIMMs, $k = 6$ or $k = 7$ is possible which allows the ORAM to maintain $90 - 95\%$ of peak possible DRAM bandwidth for every parameterization we later evaluate. Without the technique, achievable bandwidth may be $< 50\%$ depending on the data block size, recursion scheme used, number of DRAM channels, and other parameters. We note that Phantom was able to achieve 94% of peak DRAM bandwidth [66] without the subtree packing technique as there was sufficient spatial locality given their large 4 KByte block size.

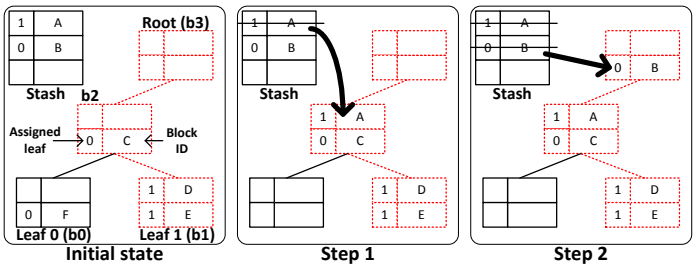


Figure 5-6: Stash eviction example for $Z = 2$ slots per bucket. Buckets are labeled $b0, b1, \ldots$ etc. We evict along the path to leaf 1, which includes buckets $b1$, $b2$ and $b3$. Each block is represented as a tuple (path, block ID), where 'path' indicates which path the block is mapped to.

5.6.2 Stash Management

As mentioned in Section 5.1.1, deciding where to evict each block in the stash is a challenge for Path ORAM hardware designs. Conceptually, this operation tries to push each block in the stash as deep (towards the leaves) into the ORAM tree as possible while keeping to the invariant that blocks can only live on the path to their assigned leaf.

Figure 5-6 works out a concrete example for the eviction logic. In Step 1 of Figure 5-6, block A is mapped to leaf 1 and therefore may be placed in buckets $b1$, $b2$, and $b3$. It gets placed in bucket $b2$ because bucket $b1$ is full and $b2$ is deeper than $b3$. In Step 2, block B could be placed in $b2$ and $b3$ and gets placed in $b3$ because $b2$ is full (since block A moved there previously).

To decide where to evict blocks, Phantom constructs an FPGA-optimized *heap sort* on the stash [66]. Unfortunately, this approach creates a performance bottleneck because the initial step of sorting the stash takes multiple cycles per block. For example, in the Phantom design, adding a block to the heap takes 11 cycles (see Appendix A of [66]). If the ORAM block size and memory bandwidth is such that writing a block to memory takes less than 11 cycles, system performance is bottlenecked by the heap-sort-based eviction logic and not by memory bandwidth.[8]

In this section, we propose a new and simple stash eviction algorithm based on bit-level hardware tricks that takes a single cycle to evict a block and can be implemented efficiently in FPGA logic. This *eliminates* the above performance overhead for any practical block size and memory bandwidth.

PushToLeaf With Bit Tricks

Our proposal, the $\mathsf{PushToLeaf}()$ routine, is shown in Algorithm 6. $\mathsf{PushToLeaf}(\mathsf{Stash}, l)$ is run once during each ORAM access and populates an array of pointers occ. Stash can be thought of as a single-ported RAM that stores data blocks and their metadata. Once populated, $\mathsf{occ}[i]$ points to the block in Stash that will be written back to the i-th position along $\mathcal{P}(l)$. Thus, to complete the ORAM eviction, a hardware state machine sends each block given by $\mathsf{Stash}[\mathsf{occ}[i]]$ for $i = 0, \ldots, (L+1) * Z - 1$ to be encrypted and written to external memory.

[8] Given the 1024 bits/cycle memory bandwidth assumed by the Phantom system, the only way for Phantom to avoid this bottleneck is to set the ORAM block size to be ≥ 1408 Bytes.

Notations. Suppose l is the current leaf being accessed. We represent leaves as L-bit words which are read right-to-left: the i-th bit indicates whether path l traverses the i-th bucket's left child (0) or right child (1). On Line 3, we initialize each entry of occ to $\perp$, to indicate that the eviction path is initially empty. Occupied is an $L+1$ entry memory that records the number of real blocks that have been pushed back to each bucket so far.

Algorithm 6 Bit operation-based stash scan. 2C stands for two's complement arithmetic.

```
1:  Inputs: The current leaf l being accessed
2:  function PushToLeaf(Stash, l)
3:      occ ← {⊥ for i = 0, ..., (L + 1) * Z − 1}
4:      Occupied ← {0 for i = 0, ..., L}
5:      for i ← 0 to T + L * Z − 1 do
6:          (a, l_i, D) ← Stash[i]                    ▷ Leaf assigned to i-th block
7:          level ← PushBack(l, l_i, Occupied)
8:          if a ≠ ⊥ and level > −1 then
9:              offset ← level * Z + Occupied[level]
10:             occ[offset] ← i
11:             Occupied[level] ← Occupied[level] + 1
12:         end if
13:     end for
14: end function
15: function PushBack(l, l′, Occupied)
16:     t_1 ← (l ⊕ l′)||0                             ▷ Bitwise XOR
17:     t_2 ← t_1 & −t_1                              ▷ Bitwise AND, 2C negation
18:     t_3 ← t_2 − 1                                 ▷ 2C subtraction
19:     full ← {(Occupied[i] ≟ Z) for i = 0 to L}
20:     t_4 ← t_3 & ∼full                             ▷ Bitwise AND/negation
21:     t_5 ← reverse(t_4)                            ▷ Bitwise reverse
22:     t_6 ← t_5 & −t_5
23:     t_7 ← reverse(t_6)
24:     if t_7 ≟ 0 then
25:         return −1                                 ▷ Block is stuck in stash
26:     end if
27:     return log_2(t_7)                             ▷ Note: t_7 must be one-hot
28: end function
```

Details for PushBack(). The core operation in our proposal is the PushBack() subroutine, which takes as input the path l we are evicting to, the path l' a block in the stash is mapped to, and outputs which level on path l that block should get written back to. In Line 16, t_1 represents in which levels the paths $P(l)$ and $P(l')$ diverge. In Line 17, t_2 is a one-hot bus where the set bit indicates the *first* level where $P(l)$ and $P(l')$ diverge. Line 18 converts t_2 to a vector of the form $000\ldots111$, where set bits indicate which levels the block *can* be pushed back to. Line 20 further excludes buckets that already contain Z blocks (due to previous calls to PushBack()). Finally, Lines 21-23 turn all current bits off except for the *left-most set bit*, which now indicates the level furthest towards the leaves that the block can be pushed back to.

Security. We remark that while our stash eviction procedure is highly-optimized for hardware implementation, it is algorithmically equivalent to the original stash eviction procedure

described in Path ORAM [71]. Thus, security follows from the original Path ORAM analysis.

Hardware Implementation and Pipelining

Algorithm 6 runs $T + (L+1)Z$ iterations of PushBack() per ORAM access, where T is the stash size not counting the path length. In hardware, we pipeline Algorithm 6 in three respects to hide its latency:

First, the PushBack() circuit itself is pipelined to have 1 block / cycle throughput. PushBack() itself synthesizes to simple combinational logic where the most expensive operation is two's complement arithmetic of $(L+1)$-bit words (which is still relatively cheap due to optimized FPGA carry chains). reverse() costs no additional logic in hardware. The other bit operations (including $\log_2(x)$ when x is one-hot) synthesize to LUTs. To meet our FPGA's clock frequency, we had to add 2 pipeline stages after Lines 17 and 18. An important subtlety is that we don't add pipeline stages between when Occupied is read and updated. Thus, a new iteration of PushBack() can be started every cycle.

Second, as soon as the leaf for the ORAM access is determined (i.e., concurrent with Step 2 in Section 5.4), blocks already in the stash are sent to the PushBack() circuit "in the background". Following the previous paragraph, $T+2$ is the number of cycles it takes to perform the background scan in the worst case.

Third, after cycle $T+2$, we send each block read on the path to the PushBack() circuit *as soon as it arrives from external memory.* Since a new block can be processed by PushBack() each cycle, eviction logic will not be the system bottleneck.

5.6.3 Reducing Encryption Bandwidth

Another serious problem for ORAM design is the area needed for encryption units. Recall from Section 5.4 that all data touched by ORAM must get decrypted and re-encrypted to preserve privacy. Encryption bandwidth hence scales with memory bandwidth and quickly becomes the area bottleneck. To address this problem we now propose a new ORAM design, which we call *RAW ORAM*, optimized to minimize encryption bandwidth at the algorithmic and engineering level.

RAW ORAM Algorithm

RAW ORAM is based on Ring ORAM and splits ORAM Backend operations into two flavors: ReadPath and EvictPath accesses. ReadPath operations perform the minimal amount of work needed to service a client processor's read/write requests (i.e., last level cache misses/writebacks) and EvictPath accesses perform evictions (to empty the stash) in the background. To reduce the number of encryption units needed by ORAM, we optimize ReadPath accesses *to only decrypt the minimal amount of data needed to retrieve the block of interest, as opposed to the entire path.* EvictPath accesses require more encryption/decryption, but occur less frequently. We now describe the protocol in detail:

Parameter A. Like Ring ORAM, RAW ORAM uses the parameter A, set at system boot time. For a given A, RAW ORAM obeys a strict schedule that the ORAM controller performs one EvictPath access after every A reads.

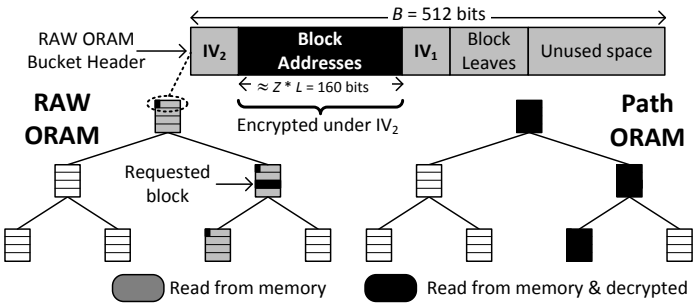


Figure 5-7: Data read vs. data decrypted on a RAW ORAM ReadPath (left) and Path ORAM access (right) with $Z = 3$. IV_1 and IV_2 are initialization vectors used for encryption.

Read Path. ReadPath operations read an ORAM tree path (as does Path ORAM) but only perform the minimal number of on-chip operations (e.g., decryption) needed to decrypt/move the requested block into the stash and logically remove that block from the ORAM tree. This corresponds to Steps 2-4 in Section 5.4 with three important changes. First, we will only decrypt the minimum amount of information needed to *find* the requested block and add it to the stash. Precisely, we decrypt the Z block addresses stored in each bucket header (Section 5.4), to identify the requested block, and then decrypt the requested block itself (if it is found). The amount of data read vs. decrypted is illustrated in Figure 5-7. Note that unlike Ring ORAM, we read the whole path as opposed to a random block in each bucket. This was done to keep the design simple.

Second, we add only the requested block to the stash (as opposed to the whole path). Third, we update the bucket header containing the requested block to indicate a block was removed (e.g., by changing its program address to $\perp$), and re-encrypt/write back to memory the corresponding state for each bucket. To re-encrypt header state only, we encrypt that state with a second initialization vector denoted IV_2. The rest of the bucket is encrypted under IV_1. A strawman design may store both IV_1 and IV_2 in the bucket header (as in Figure 5-7). We describe an optimized design at the end of this section.

Evict Path. EvictPath performs a normal but *dummy* Path ORAM access to a static sequence of leaves corresponding to the *reverse lexicographic order* of paths (as with Ring ORAM, Chapter 3). By dummy access, we simply mean reading and evicting a path to push out blocks in the stash that have accumulated over the A ReadPath operations.

Security. The security analysis is very similar (and simpler, even) to that in Ring ORAM. ReadPath accesses always read paths in the ORAM tree at random, just like Path ORAM. Further, EvictPath accesses occur at predetermined times and are to predictable/data-independent paths.

Performance and Area Characteristics

Assume for simplicity that the bucket header is the same size as a data block (which matches our evaluation). Then, each ReadPath access reads $(L + 1)Z$ blocks on the path, but only decrypts 1 block; it also reads/writes and decrypts/re-encrypts the $L + 1$ headers/blocks. An EvictPath reads/writes and decrypts/re-encrypts all the $(L+1)(Z+1)$ blocks on a path.

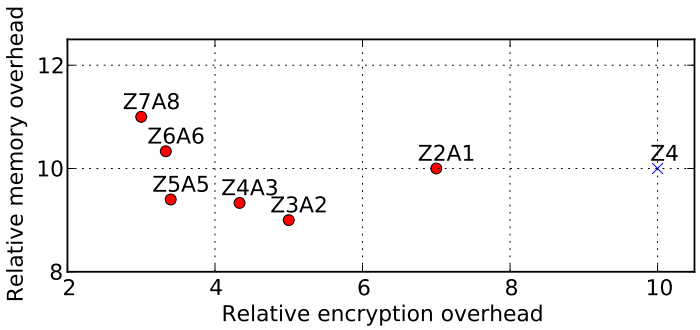


Figure 5-8: The relative memory and encryption bandwidth overhead of RAW ORAM with different parameter settings.

Thus, in RAW ORAM the relative memory bandwidth per bucket is $Z + 2 + \frac{2(Z+1)}{A}$, and the relative encryption bandwidth per bucket is roughly $1 + \frac{2(Z+1)}{A}$.

The remaining question for RAW ORAM is: what A and Z combinations result in a stash that will not overflow, yet at the same time minimize encryption and memory bandwidth? In Figure 5-8, we visualize the relative memory and encryption bandwidth of RAW ORAM with different parameter settings that have been shown (in Section 3.6) to give negligible stash overflow probability. We find $Z = 5, A = 5$ (Z5A5) to be a good trade-off as it achieves 6% memory bandwidth improvement and $\sim 3\times$ encryption reduction over Path ORAM. We will use Z5A5 in the evaluation and remark that this configuration requires $T = 64$ for a 2^{-80} stash overflow probability.

Exploiting Path Eviction Predictability

Despite RAW ORAM's theoretic area savings for encryption units, careful engineering is needed to prevent that savings from turning into performance loss. The problem is that by reducing encryption units (i.e., AES) to provide "just enough" bandwidth for ReadPath accesses, we are forced to wait during EvictPath accesses for that reduced number of AES units to finish decrypting/re-encrypting the entire path. Further, since all AES IVs are stored externally with each bucket, the AES units can't start working on a new EvictPath until that access starts.

To remove the above bottleneck while maintaining the AES unit reduction, we make the following key observation: **Since EvictPath operations occur in a predictable, fixed order, we can determine exactly how many times any bucket along any path has been *written* in the past.**

Suppose that, as in Section 5.6, G is a counter that tracks the number of EvictPath accesses made so far. Also as before, we know the next leaf (and its corresponding path) being evicted is given precisely by $G \mod 2^L$. (We allocate a 64-bit counter in the ORAM controller to store G.) Now, due to load-balancing nature of reverse lexicographic order, if $\mathcal{P}(l)[i]$ has been evicted to g_i times in the past, then $\mathcal{P}(l)[i+1]$ has been evicted $g_{i+1} = \lfloor (g_i + 1 - l_i)/2 \rfloor$ where l_i is the i-th bit in leaf l. This can be easily computed in hardware as $g_{i+1} = (g_i + \sim l_i) \gg 1$, where $\gg$ is a right bit-shift and $\sim$ is bit-wise negation.

Using eviction predictability, we will *pre-compute* the AES-CTR initialization vector IV_1. Simply put, this means the AES units can do all decryption/encryption work for EvictPath accesses "in the background" during concurrent ReadPath accesses.

To decrypt the i-th 128-bit ciphertext chunk of the bucket with unique ID $BucketID$

at level j in the tree, we XOR it with the following mask: $\mathsf{AES}_K(g_j \,||\, BucketID \,||\, i)$ where g_j is the bucket eviction count defined above. Correspondingly, re-encryption of that chunk is done by generating a new mask where the write count has been incremented by 1. We note that with this scheme, g_j takes the place of IV_1 and since g_j can be derived internally, we need not store it externally.

On both ReadPath and EvictPath operations, we must decrypt the program addresses and valid bits of all blocks in each bucket. For this we may apply the global counter scheme from Section 5.5.3 or use the mask as in Ren et al. [69], namely $\mathsf{AES}_K(\mathsf{IV}_2 \,||\, BucketID \,||\, i)$, where IV_2 is stored externally as part of each bucket's header.

At the implementation level, we time-multiplex an AES core between generating masks for IV_1 and IV_2. The AES core prioritizes IV_2 operations; when the core is not servicing IV_2 requests, it generates masks for IV_1 in the background and stores them in a FIFO.

5.7 Evaluation (Simulation)

In this section we will thoroughly evaluate our proposals in software simulation.

5.7.1 Methodology

We perform simulations using Graphite [39] with the processor parameters listed in Table 5.1. Parameters are chosen to approximate a mid-range system. The core and cache model remain the same in all experiments; unless otherwise stated, we assume the ORAM parameters from the table. We use a subset of SPEC06-int benchmarks [31] with reference inputs. All workloads are warmed up over 1 billion instructions and then run for 3 billion instructions. Averages reported are geometric means.

We derive AES/SHA3 latency, Frontend and Backend latency directly from our hardware prototype (Sections 5.8-5.10). Unless otherwise specified, the simulations assume a basic Path ORAM Backend as opposed to RAW ORAM. Frontend latency is the time to evict and refill a block from the PLB (Section 5.5.1) and occurs at most once per Backend call. Backend latency (approximately) accounts for the cycles lost due to hardware effects such as serializers/buffer latency/etc and is added on top the time it takes to read/write an ORAM tree path in DRAM, which is given in Section 5.7.2. Both of these delays are small relative to ORAM access latency, indicating the hardware prototype is mostly bottlenecked by available memory bandwidth (our intent).

We model DRAM and ORAM accesses on top of commodity DRAM using DRAMSim2 [45] and its default DDR3_micron configuration with 8 banks, 16384 rows and 1024 columns per row. Each DRAM channels runs at 667 MHz DDR with a 64-bit bus width and provides $\sim$ 10.67 GB/s peak bandwidth. All ORAM configurations assume 50% DRAM utilization (meaning a 4 GB ORAM requires 8 GB of DRAM) and use the subtree layout scheme from [69] to achieve nearly peak DRAM bandwidth.

5.7.2 ORAM Latency and DRAM Channel Scalability

ORAM latency is sensitive to DRAM bandwidth and for this reason we explore how changing the channel count impacts ORAM access time in Table 5.2. ORAM Tree latency refers to the time needed for the Backend to read/write a path in the Unified ORAM tree, given the ORAM parameters in Table 5.1. All latencies are in terms of *processor* clock cycles, and

Table 5.1: Processor configuration for evaluation.

Core, on-chip cache and DRAM	
core model	in order, single issue, 1.3 GHz
add/sub/mul/div	1/1/3/18 cycles
fadd/fsub/fmul/fdiv	3/3/5/6 cycles
L1 I/D cache	32 KB, 4-way, LRU
L1 data + tag access time	1 + 1 cycles
L2 Cache	1 MB, 16-way, LRU
L2 data + tag access time	8 + 3 cycles
cache line size	64 B
Path ORAM/ORAM controller	
ORAM controller clock frequency	1.26 GHz
data block size	64 B
data ORAM capacity	4 GB ($N = 2^{26}$)
block slots per bucket (Z)	4
AES-128 latency	21 cycles (Section 5.8)
SHA3-224 latency (PMMAC)	18 cycles (Section 5.8)
Frontend latency	20 cycles (Section 5.8)
Backend latency	30 cycles (Section 5.8)
Memory controller and DRAM	
DRAM channels	2 ($\sim$ 21.3 GB peak bandwidth)
DRAM latency	given by DRAMSim2 [45]

Table 5.2: ORAM access latency by DRAM channel count.

DRAM channel count	1	2	4	8
ORAM Tree latency (cycles)	2147	1208	697	463

represent an average over multiple accesses. For reference, a DRAM access for an insecure system without ORAM takes on average 58 processor cycles.

Generally, ORAM latency decreases with channel count as expected but the effect becomes increasingly sub-linear for larger channel counts due to DRAM channel conflicts. Since 2 channels represent realistic mid-range systems and do not suffer significantly from this problem, we will use that setting for the rest of the evaluation unless otherwise specified.

5.7.3 PLB Design Space

Figure 5-9 shows how direct-mapped PLB capacity impacts performance. For a majority of benchmarks, larger PLBs add small benefits ($\leq$ 10% improvements). The exceptions are bzip2 and mcf, where increasing the PLB capacity from 8 KB to 128 KB provides 67% and 49% improvement, respectively. We tried increasing PLB associativity and found that, with a fixed PLB capacity, a fully associative PLB improves performance by $\leq$ 10% when compared to direct-mapped. To keep the architecture simple, we therefore assume direct-mapped PLBs from now on. Going from a 64 KB to 128 KB direct-mapped PLB, average performance only increases by 2.7%, so we assume a 64 KB direct-mapped PLB for the rest of the evaluation.

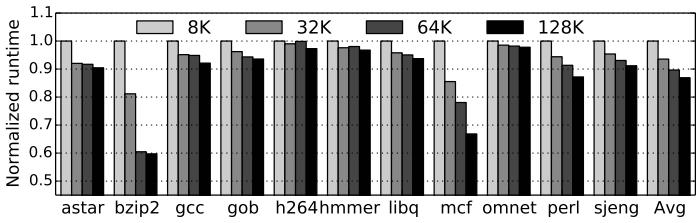


Figure 5-9: PLB design space, sweeping direct-mapped PLB capacity. Runtime is normalized to the 8 KB PLB point.

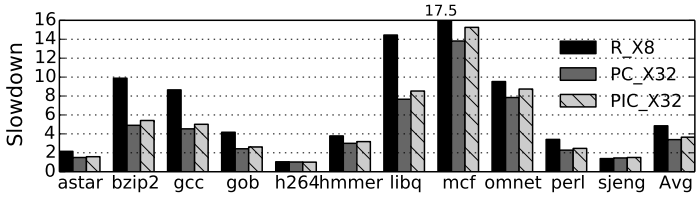


Figure 5-10: Performance of PLB, Compressed PosMap and PMMAC. Slowdown is relative to an insecure system without ORAM.

5.7.4 Scheme Composability

We now present the impact on performance when we compose PLB (Section 5.5.1), PosMap compression (Section 5.5.2) and PMMAC (Section 5.5.3). To name our schemes in the discussion, we use the letters P, I and C to indicate the **P**LB, **I**ntegrity verification (PMMAC) and **C**ompressed PosMap, respectively. For example, PC_X32 denotes PLB+Compressed PosMap with $X = 32$. PI_X8 is the flat-counter PMMAC scheme from Section 5.5.3. For PC_X32 and PIC_X32, we apply recursion until the on-chip PosMap is $\leq$ 128 KB in size, yielding 4 KB on-chip PosMaps for both points. R_X8 is a Recursive ORAM baseline with $X = 8$ (32-Byte PosMap ORAM blocks following [69]) and $H = 4$, giving it a 272 KB on-chip PosMap.

Despite consuming less on-chip area, PC_X32 achieves a 1.43× speedup (30% reduction in execution time) over R_X8 (geomean). To provide integrity, PIC_X32 only adds 7% overhead on top of PC_X32, which is due to the extra bandwidth needed to transfer per-block MACs. This overhead will decrease with larger block sizes.

To give more insight, Figure 5-11 shows the average data movement per ORAM access (i.e., per LLC miss+eviction). We give the Recursive ORAM R_X8 up to a 256 KB on-chip PosMap. As ORAM capacity increases, the overhead from accessing PosMap ORAMs grows quickly for R_X8. All schemes using a PLB have much better scalability. For the 4 GB ORAM, on average, PC_X32 reduces PosMap bandwidth overhead by 82% and overall ORAM bandwidth overhead by 38% compared with R_X8. At the 64 GB capacity, the reduction becomes 90% and 57%. Notably the PMMAC scheme without compression (PI_X8) causes nearly half the bandwidth to be PosMap related, due to the large counter width and small X (Section 5.5.3). The compressed PosMap (PIC_X32) solves this problem.

5.7.5 Comparison to Non-Recursive ORAM with Large Blocks ([66])

In Figure 5-12, we compare our proposal to the parameterization used by Phantom [66]. Phantom was evaluated with a large ORAM block size (4 KBytes) so that the on-chip

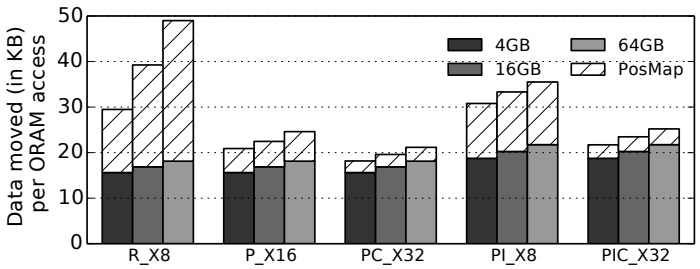


Figure 5-11: Scalability to large ORAM capacities. White shaded regions indicate data movement from PosMap ORAM lookups. Slowdown is relative to an insecure system without ORAM.

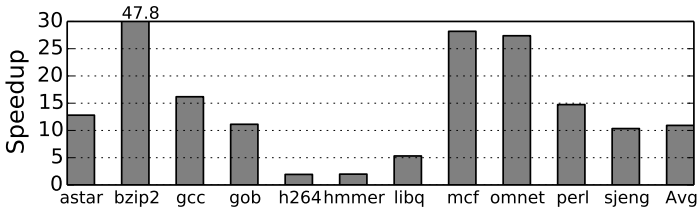


Figure 5-12: PC_$X32$ speedup relative to Phantom [66] w/ 4 KB blocks.

PosMap could be contained on several FPGAs without recursion for ORAM capacity. (Of course, as the working set size increases, the PosMap size will increase proportionally.)

To match Section 5.7.4, we model the Phantom parameters on 2 DRAM channels (which matches the DRAM bandwidth reported in [66]) and with a 4 GB ORAM ($N = 2^{20}, L = 19$) $Z = 4$, 4 KB blocks and no recursion. For these parameters, the on-chip PosMap is ~ 2.5 MB (which we evaluate for on-chip area in Section 5.9.4). To accurately reproduce Phantom's system performance, we implemented the Phantom block buffer (Section 5.7 of that work) as a 32 KB memory with the CLOCK eviction strategy and assume the Phantom processor's cache line size is 128 Bytes (as done in [66]).

On average, PC_$X32$ from Section 5.7.4 achieves 10× speedup over the Phantom configuration with 4 KB blocks. The intuition for this result is that Byte movement per ORAM access for our scheme is roughly $(26 * 64)/(19 * 4096) = 2.1\%$ that of Phantom. While PC_$X32$ needs to access PosMap blocks due to recursion, this effect is outweighed by the reduction in Data ORAM Byte movement.

5.7.6 Performance Evaluation Using A Ring ORAM Backend

The starting point for our hardware ORAM was Path ORAM due to that design's conceptual simplicity. We will now simulate performance as if our starting point was Ring ORAM (Chapter 3). Since we already know Ring ORAM's analytic bandwidth relative to Path ORAM, the purpose of this study is to see how *online bandwidth reduction* translates to performance gain.

We use the same processor/cache architecture as previous experiments. To give a more complete picture, we additionally evaluate two common database workloads tpcc and ycsb. Due to the small block size, we parameterize Ring ORAM at $Z = 5$, $A = 5$, $S = 7$ to reduce metadata overhead. To isolate the bandwidth improvement due to Ring ORAM, all configurations use the optimized ORAM recursion techniques from [69] (i.e., without

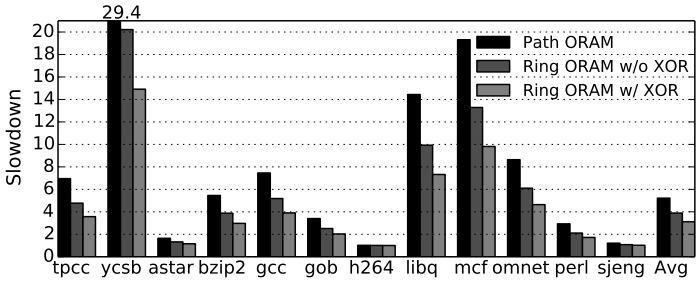


Figure 5-13: Performance improvement of Ring ORAM vs. Path ORAM.

the PLB): we apply recursion three times with 32-Byte position map block size and get a 256 KB final position map. We do not use tree-top caching or the mechanisms from Section 5.5 since it proportionally benefits both Ring ORAM and Path ORAM. Today's DRAM DIMMs cannot perform any computation, but it is not hard to imagine having simple XOR logic either inside memory, or connected to $O(\log N)$ parallel DIMMs so as not to occupy processor-memory bandwidth. Thus, we show results with and without the XOR technique.

Figure 5-13 shows program slowdown over an insecure DRAM. The high order bit is that using Ring ORAM with XOR results in a geometric average slowdown of 2.8× relative to an insecure system. This is a 1.5× improvement over Path ORAM. If XOR is not available, the slowdown over an insecure system is 3.2×.

For completeness, we have also repeated the experiment with the Unified ORAM and PLB techniques (Section 5.5.1). The geometric average slowdown over an insecure system in that case is 2.4× (2.5× without XOR).

5.8 Evaluation (FPGA Prototype)

We now describe our hardware prototype of Tiny ORAM on a Virtex-7 VC707 FPGA board and analyze its area and performance characteristics. Our main reason for hardware prototyping is to tape-out in ASIC. With that in mind, the FPGA evaluation has two primary objectives. First, we wish to compare against Phantom (which was optimized for FPGA) in as apples-to-apples a comparison as possible. Second, we wish to demonstrate our design working under 'high memory bandwidth' conditions.[9]

The entire design (as well as the extension to ASIC) is open source at `http://kwonalbert.github.io/oram`. For the FPGA and ASIC evaluation sections, Table 5.3 may be used as a guide for which and how optimizations were evaluated.

5.8.1 Metrics and Baselines

The entire design is written in plain Verilog and was synthesized using the Xilinx Vivado flow (version 2013.4). Performance is measured as the latency (in FPGA cycles or real time) between when an FPGA user design requests a block and Tiny ORAM *returns* that block. Area is calculated in terms of FPGA lookup-tables (LUT), flip-flops (FF) and Block RAM (BRAM), and is measured post place-and-route (i.e., represents final hardware area numbers). For the rest of the paper we count BRAM in terms of 36 Kbit BRAM.

[9]The FPGA fabric's internal clock frequency is lower than an ASIC's, yet the memory clock is unchanged, giving the FPGA design the impression of higher memory bandwidth.

Table 5.3: Which optimizations were implemented/evaluated in hardware (and are a part of the open source release)? Which of those were evaluated in the ASIC flow and included in the final tape-out?

Optimization	Section	Implemented in hardware	Included in tape-out
PLB+Unified ORAM	5.5.1	✓	✓
PosMap Compression	5.5.2		
PMMAC	5.5.3	✓	✓
Subtree address layout	5.6.1	✓	✓
Bit-based stash management	5.6.2	✓	✓
RAW ORAM	5.6.3	✓	

We compare Tiny ORAM with two baselines shown in Table 5.4. The first one is Phantom [66], which we normalize to our ORAM capacity and the 512 bits/cycle DRAM bandwidth of our VC707 board. We further disable Phantom's tree top caching. Phantom's performance/area numbers are taken/approximated from the figures in their paper, to our best efforts. The second baseline is a basic Path ORAM with our stash management technique, to show the area saving of RAW ORAM.

5.8.2 Implementation

Organization. We built the design hierarchically as three main components: the Frontend, stash (Backend) and AES units used to decrypt/re-encrypt paths (Backend). We evaluate both Path ORAM and RAW ORAM Backend designs (Section 5.6.3). The Path ORAM Backend is similar to the Phantom Backend.

Unlike Phantom, our design does not have a DRAM buffer (see [66]). We remark that if such a structure is needed it should be much smaller than that in Phantom (<10 KBytes as opposed to hundreds of KBytes) due to our 64 Byte block size.

Parameterization. Both of our designs (Path ORAM and RAW ORAM) use $B = 512$ bits per block and $L = 20$ levels. Our choice of $B = 512$ (64 Bytes) shows that Tiny ORAM can run even very small block sizes without imposing hardware performance bottlenecks. We are constrained to set $L = 20$ because this setting fills the VC707's 1 GByte DRAM DIMM.

Using the notation from Section 5.7.4, the Frontend we evaluate is P_X16. We do not evaluate the cost of integrity (PMMAC) in the FPGA prototype as integrity was not considered by the Phantom design and does not impact memory throughput (Section 5.7.4). Integrity will be added in the final ASIC evaluation (Sections 5.9-5.10).

Clock regions. The DRAM controller on the VC707 board runs at 200 MHz and transfers 512 bits/cycle. To ensure that DRAM is Tiny ORAM's bottleneck, we optimized our design's timing to run at 200 MHz.

DRAM controller. We interface with DDR3 DRAM through a stock Xilinx on-chip DRAM controller with 512 bits/cycle throughput.[10] From when a read request is presented

[10]Given the expected ASIC clock frequency from the simulations, this is similar to having 8 DRAM channels without channel conflicts.

to the DRAM controller, it takes $\sim$ 30 FPGA cycles to return data for that read (i.e., without ORAM). The DRAM controller pipelines requests. That is, if two reads are issued in consecutive cycles, two 512 bit responses arrive in cycle 30 and 31. As mentioned before, the subtree layout scheme allows us to achieve near-optimal DRAM bandwidth.

Encryption. We use "tiny aes," a pipelined AES core that is freely downloadable from Open Cores [1]. Tiny aes has a 21 cycle latency and produces 128 bits of output per cycle. One tiny aes core costs 2865/3585 FPGA LUT/FF and 86 BRAM. To implement the time-multiplexing scheme from Section 5.6.3, we simply add state to track whether tiny aes's output (during each cycle) corresponds to IV_1 or IV_2.

Given our DRAM bandwidth, RAW ORAM requires 1.5 (has to be rounded to 2) tiny aes cores to completely hide mask generation for EvictPath accesses at 200 MHz. To reduce area further, we optimized our design to run tiny aes and associated control logic at 300 MHz. Thus, our final design requires only a single tiny aes core. Basic Path ORAM would require 3 tiny aes cores clocked at 300 MHz, which matches our 3$\times$ AES saving in the analysis from Section 5.6.3. We did not optimize the tiny aes clock for basic Path ORAM, and use 4 of them running at 200 MHz.

5.8.3 Access Latency Comparison

For the rest of the FPGA evaluation, all access latencies are averages when running *on a live hardware prototype*. Table 5.4 gives a summary of results. Our RAW ORAM Backend can finish an access in 276 cycles ($1.4\mu s$) on average. This is very close to basic Path ORAM; we did not get the 6% theoretical performance improvement because of the slightly more complicated control logic of RAW ORAM.

After normalizing to our DRAM bandwidth and ORAM capacity, Phantom should be able to fetch a 4 KByte block in $\sim 60\mu s$. This shows the large speedup potential for small blocks. Suppose the program running has bad data locality (i.e., even though Phantom fetches 4 KBytes, only 64 Bytes are touched by the program). In this case, Tiny ORAM using a 64 Byte block size improves ORAM latency by 40$\times$ relative to Phantom with a 4 KByte block size. We note that Phantom was run at 150 MHz: if optimized to run at 200 MHz like our design, our improvement is $\sim$ 32$\times$. Even with perfect locality where the entire 4 KByte data is needed, using a 64 Byte block size introduces only $1.5 - 2\times$ slowdown relative to the 4 KByte design.[11] More data on the block size trade-off is given in Section 5.7.5.

5.8.4 Hardware Area Comparison

In Table 5.4, we also see that the RAW ORAM Backend requires only a small percentage of the FPGA's total area. The slightly larger control logic in RAW ORAM dampens the area reduction from AES saving. Despite this, RAW ORAM achieves an $\geq 2\times$ reduction in BRAM usage relative to Path ORAM. Note that Phantom [66] did not implement encryption: we extrapolate their area by adding 4 tiny aes cores to their design and estimate a BRAM savings of 4$\times$ relative to RAW ORAM.

[11] This slowdown is due to the Path ORAM algorithm: with a fixed memory size, a larger block size results in a shorter ORAM tree (i.e., L decreases which improves performance).

Table 5.4: Parameters, performance and area summary of different designs. Access latencies for Phantom are normalized to 200 MHz. All %s are relative to the Xilinx XC7VX485T FPGA. For Phantom area estimates, "$\sim 235 + 344$" BRAM means 235 BRAM was reported in [66], plus 344 for tiny aes.

Design	Phantom	Path ORAM	RAW ORAM
Parameters			
Z, A	4, N/A	4, N/A	5, 5
Block size	4 KByte	64 Byte	64 Byte
# of tiny aes cores	4	4	1
Performance (cycles)			
Access 64 B	$\sim$ 12000	270	276
Access 4 KB	$\sim$ 12000	17280	17664
ORAM Backend Area			
LUT (%)	$\sim 6000 + 11460$	18977 (7%)	14427 (5%)
FF (%)	not reported	16442 (3%)	11359 (2%)
BRAM (%)	$\sim 172 + 344$	357 (34%)	129 (13%)
Total Area (Backend+Frontend)			
LUT (%)	$\sim 10000 + 11460$	22775 (8%)	18381 (6%)
FF (%)	not reported	18252 (3%)	13298 (2%)
BRAM (%)	$\sim 235 + 344$	371 (36%)	146 (14%)

5.8.5 Full System Evaluation

We now evaluate a complete ORAM controller by connecting our RAW ORAM Backend to the optimized ORAM Frontend from Section 5.5. For completeness, we also implemented and evaluated a baseline Recursive Path ORAM. (To our knowledge, we are the first to implement any form of Recursive ORAM in hardware.) We call configurations with our optimized Frontend "Freecursive" to distinguish them from the baseline Frontend. For our $L = 20$, we add 2 PosMap ORAMs, to attain a small on-chip position map (< 8 KB).

Figure 5-14 shows the average memory access latency of several real SPEC06-int benchmarks. Due to optimizations from Section 5.5, performance depends on program locality. For this reason, we also evaluate two synthetic traces: scan which has perfect locality and rand which has no locality. We point out two extreme benchmarks: libq is known to have good locality, and on average our ORAM controller can access 64 Bytes in 490 cycles. sjeng has bad (almost zero) locality and fetching a 64 Byte block requires $\sim$ 950 cycles (4.75 μs at 200 MHz). Benchmarks like sjeng reinforce the need for small blocks: setting a larger ORAM block size will strictly decrease system performance since the additional data in larger blocks won't be used.

5.9 Evaluation (ASIC Prototype, post Synthesis)

We now evaluate the Tiny ORAM design pushed through Synopsis' ASIC synthesis tool Design Compiler. This work is the first to prototype any ORAM through an ASIC hardware flow. The main objective in this section is to show design scalability (through area consumption), as a function of available DRAM bandwidth. Results post-Place and Route (via IC Compiler) and post tape-out are given in the next section.

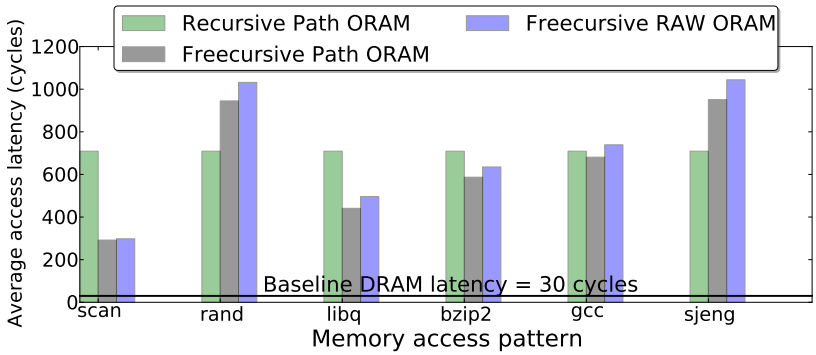


Figure 5-14: Average number of FPGA clock cycles needed to complete an ORAM access. We call configurations with our optimized Frontend "Freecursive" to distinguish them from the baseline Frontend.

Table 5.5: ORAM area breakdown post-synthesis.

		DRAM channels (nchannel)		
		1	2	4
Area (% of total)	Frontend	31.2	30.0	22.5
	PosMap	7.3	7.0	5.3
	PLB	10.2	9.7	7.3
	PMMAC	12.4	11.9	8.8
	Misc	1.3	1.4	1.1
	Backend	68.8	70.0	77.5
	Stash	28.3	28.9	21.9
	AES	40.5	41.1	55.6
Total cell area (mm^2)		.316	.326	.438

5.9.1 Metrics

The design (organization and codebase) is derived from the FPGA implementation (Section 5.8). We give area results for different DRAM bandwidths, and say the ORAM (Backend) has a $64 * \text{nchannel}$ bit/cycle datapath to/from DRAM.

The design was synthesized using a 32 nm commercial standard cell library and memory (SRAM and register file) generator. For synthesis results, we report total cell area; i.e., the minimum area required to implement the design. For layout results, we set a bounding box for each major block that maximized utilization of available space while meeting timing requirements.

5.9.2 Implementation

Organization. The entire design required 5 SRAM/RF memories (which we manually placed during layout): the PLB data array, PLB tag array, on-chip PosMap, stash data array and stash tag array. Numerous other (small) buffers were needed and implemented in standard cells.

Parameterization. ORAM parameters follow Table 5.1 except that we use an 8 KB PosMap and 8 KB direct-mapped PLB by default (we discuss a 64 KB PLB design in Section 5.9.4). All ASIC Frontend results include the PMMAC integrity verifier (Section 5.5.3).

In particular, the Frontend we evaluate is PI_$X8$. For PMMAC, we use flat 64 bit counters to check freshness. PosMap compression (Section 5.5.2) was not implemented in any form, but we expect its hardware area to be negligible. Due to the increased technical sophistication (and therefore validation effort) of the RAW ORAM design post tape-out, we chose to not evaluate that scheme in ASIC. We remark on its expected efficiency below.

Encryption/Hashing units. To implement cryptographic operations, we used two AES cores from OpenCores [1]: the 21-cycle pipelined core "tiny aes" to implement the Backend's read/write path decryptions/encryptions and a non-pipelined 12 cycle core to implement $\mathsf{PRF}_K()$ (Section 5.5.2). We used a SHA3-224 core from OpenCores to implement $\mathsf{MAC}_K()$ for PMMAC (Section 5.5.3).

5.9.3 Results

Table 5.5 shows ORAM area across several DRAM channel (nchannel) counts. All three configurations met timing using up to a 1.3 GHz clock with a 100 ps uncertainty. Notice that the Frontend constitutes a minority of the total area and that this percentage decreases with nchannel. Area decreases with nchannel because the Frontend performs a very small DRAM bandwidth-independent amount of work (~ 50 clock cycles including integrity verification) per Backend access. The Backend's bits/cycle throughput (AES, stash read/write, etc), on the other hand, must rate match DRAM.

We note the following design artifact: since we use AES-128, area increases only slightly from nchannel $= 1$ to 2, because both 64-bit and 128-bit datapaths require the same number of AES units. If RAW ORAM were used in the tape-out, Section 5.8 tells us that AES area would not increase between nchannel $= 1$ to nchannel $= 8$ (inclusive). (When nchannel $= 8$, DRAM bandwidth is upper-bounded by 512 bits/cycle, which is equivalent to the FPGA design's bandwidth.)

5.9.4 Alternative Designs

To give additional insight, we estimate the area overhead of *not* supporting recursion (i.e., like Phantom) or having a larger PLB with recursion. For a 4 GB ORAM, the PosMap must contain 2^{26} to 2^{20} entries (for block sizes 64 Bytes to 4 KB). With a 2^{20}-entry on-chip PosMap, without Recursion, the nchannel $= 2$ design requires ~ 5 mm^2 area — an increase of over $10\times$ our post-layout result. Doubling the ORAM capacity (roughly) doubles this cost. Further, for performance reasons, we prefer smaller block sizes which exacerbates the area problem (Section 5.7.5). On the other hand, we calculate that using Recursion with a 64 KB PLB (to match experiments in Section 5.7) increases the area for the nchannel $= 1$ configuration by 29% (and is 26% of total area).

5.10 Evaluation (ASIC Prototype, post Layout/Tape-out)

We now evaluate a Tiny ORAM prototype, taped-out and integrated into the Princeton Piton/Ascend processor [93, 49]. A diagram of the chip and lab setup for bring-up tests is given in Figure 5-15. Overall, the chip is composed of 25 cache-coherent SPARC T1 cores and Tiny ORAM serves as a memory controller for when these cores have LLC misses. The design was taped-out March 2015 in a 32 nm commercial technology, and was successfully tested in January 2017. The main objective is to validate the prototype's functionality,

performance and power – and to prove that the design is suitable for integration into a single-chip secure processor.

Parameterization. The taped-out design matches the nchannel = 2 point from the previous section. Fixed ORAM parameters for the tape-out were $B = 512$ bits and $Z = 4$.

5.10.1 Tape-out Area and Performance

For the final chip's place-and-route and layout (via IC Compiler), we adopted a hierarchical work flow. We divided Tiny ORAM into three logical modules: Frontend, Backend, and Encryption (the AES units). We placed and routed the three modules separately. Their respective dimensions and post-layout areas are given in Table 5.6.

The bounding box for the entire design was set to be 2 mm × 0.5 mm. This is due to an early design decision to put Tiny ORAM at the top edge of the chip as well as artificial constraints imposed by SRAM dimensions. Therefore, while the bounding box occupies ∼1 mm^2 area, Tiny ORAM post-layout area is more accurately represented by the combined post-layout area of the three modules, which sum to ∼0.51 mm^2. Each module met timing at a target clock frequency of 1 GHz.

Remark. Recall that post-synthesis Tiny ORAM had a total area of 0.326 mm^2. It is common for design area to increase post-layout, as module bounding boxes must be set conservatively to meet timing.

Table 5.6: Dimensions (width × height) and area of the three modules in Tiny ORAM.

Module	Frontend	Backend	Encryption
Dimensions (μm)	636.7 × 218.7	346.6 × 364.5	669.0 × 364.5
Area (mm^2)	0.139	0.126	0.244

The nchannel = 2 configuration, above, can complete an ORAM access for 512 bits of user data (one cache line) in ∼ 1275 cycles (not including the initial round-trip delay to retrieve the first word of data from external memory). Recall in Section 5.7.2 we reported a very similar ORAM latency of 1208 cycles per access for nchannel = 2, which leads to an average slowdown of ∼ 4× on SPEC-Int-2006 benchmarks with a typical cache hierarchy.

5.10.2 Functional Tests and Power Measurements in Silicon

We now test ORAM functionality and measure its power consumption on first silicon. Functionality was tested over a stride and random client access pattern. Client requests are delivered to Tiny ORAM, and Tiny ORAM makes requests to external memory over a UART connected to a host machine (which emulates real main memory). In all tests, the series of requests made by Tiny ORAM match simulation which proves correct operation.

Table 5.7 shows Tiny ORAM's dynamic power consumption across a range of voltages and clock frequencies. Dynamic power includes transistor switching power for Tiny ORAM's logic, the SRAMs and the clock. For each test, core voltage (VDD) is set to the values shown in the table and SRAM voltage is set to 0.05 V higher than VDD. The power numbers do not include the power consumption from non-ORAM logic on the chip, the I/O pins or the external memory.

Table 5.7: ORAM controller power consumption (mW) under different frequencies and voltages.

V \ MHz	250	500	750	857
0.7	29.5			
0.75	32.4			
0.8	36.8			
0.85	43.2	74.8		
0.9	50.7	84.9		
0.95	57.6	97.9		
1.0			150	
1.1			208	299

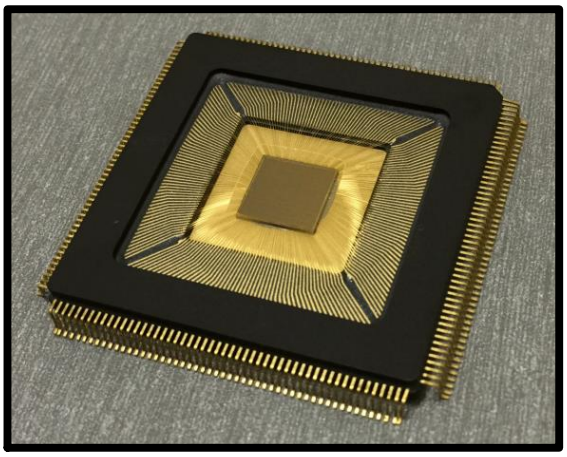
To measure peak power consumption, we need to keep the ORAM controller busy servicing memory requests at its highest throughput possible. Therefore, during power tests, we feed the ORAM controller with a synthetic memory request trace from an on-chip traffic generator, and use an on-chip buffer mimicking an external memory that has zero latency and can fully utilize the chip pin bandwidth. Thus, each measurement gives an upper bound on the chip power consumption in a real deployment.

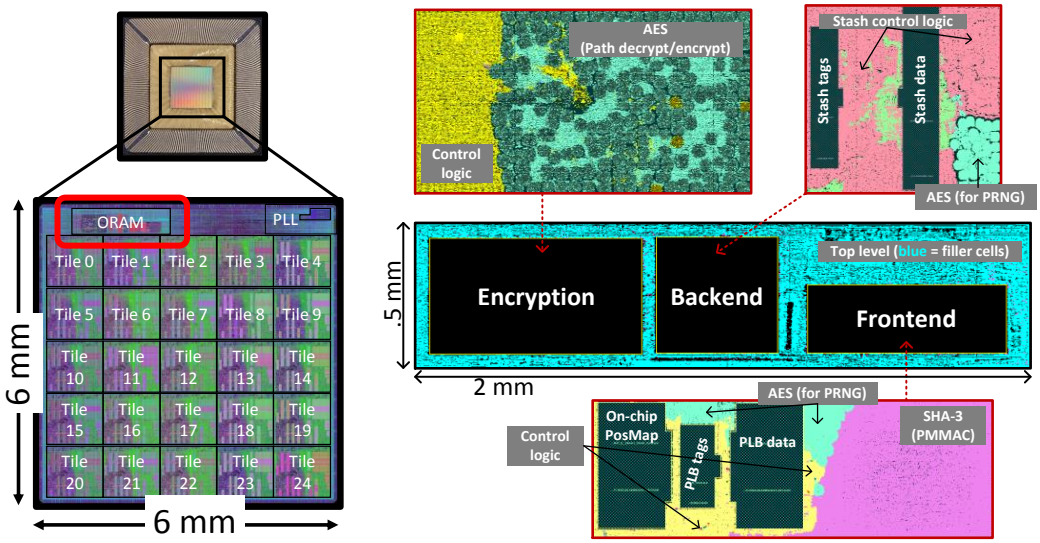
In each test, we sample the chip current draw 100 times in 16 seconds and compute the average power. It is worth noting that the ORAM power consumption will gradually increase with time as the chip temperature increases. At lower voltage (< 1 V), the effect of temperature increase is not noticeable and we start sampling the current 5 seconds after power on. At high voltage (≥ 1 V), this effect cannot be ignored, and we wait for the current draw to stabilize before sampling current.

Generally, running at a higher clock frequency requires a higher voltage to make transistors toggle faster. For each frequency in Table 5.7, the ORAM logic will stop functioning (not meet timing) below a certain voltage, at which point we stop measuring power. For each frequency, the ideal point to run the ORAM controller is the lowest recorded voltage, which is the point that ORAM functions and consumes the least power. Since increasing voltage beyond the threshold strictly consumes more power, we omit the 1 V and 1.1 V measurements for frequencies 250 MHz and 500 MHz.

Our test setup constrained us to test voltages $\leq$1.1 V. This is why we were only able to test frequencies up to 857 MHz. If equipped with a more effective cooling solution, the chip may function beyond 857 MHz with >1.1 V voltage. We repeat the test at 500 MHz and 0.9 V across three different chips. Dynamic power consumption across chips varies by about 7%.

We also measure the power consumption from the clock tree. For these tests, the ORAM controller receives the clock and is ready to service memory requests, but no memory request is made. For the frequencies and voltages tested in Table 5.7, the clock tree accounts for around 40% of the total dynamic power.





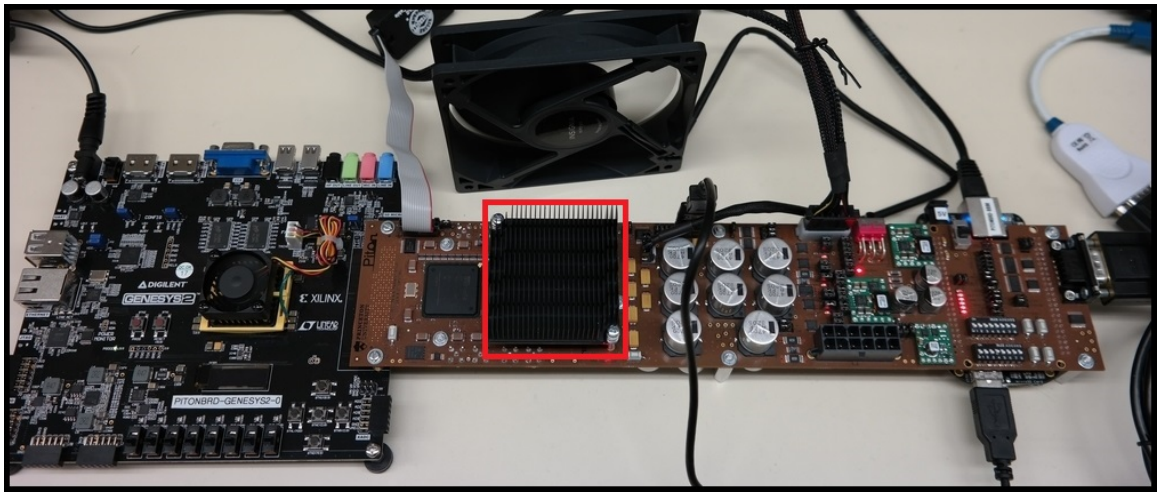


Figure 5-15: Chip die and package photo (top), whole-chip tape-out diagram (mid left), Tiny ORAM tape-out diagram broken up into the three logical top modules (mid right), bring-up lab setup (bottom - chip under heat sink in red).

Chapter 6

Conclusion

This thesis developed new ORAM constructions that only require a small amount of client storage, and taped-out the first small client storage hardware ORAM controller in 32 nm silicon. Our first construction, Ring ORAM, achieves constant online bandwidth and outperforms all prior small client storage schemes, up to constant factors. Due to its performance across different client storage regimes, we view this work as a leading candidate for practical ORAM that can be deployed today. Our second construction, Onion ORAM, is the first ORAM to achieve constant overall bandwidth without relying on heavy weight cryptography such as fully homomorphic encryption (FHE). Due to its reliance on cheaper cryptographic techniques, we view this work as taking an important step towards *practical* constant bandwidth blowup ORAMs. Finally, we present the first hardware ORAM that is implementable in a single processor chip. This work proves the viability of a single-chip secure processor which can protect the privacy of software IP or user data, as it interacts with an external memory device.

We will conclude by discussing two challenges left open in the server computation ORAM line of work (Chapter 4). First, while the block size in Onion ORAM is poly-logarithmic, the exponent is rather large (especially for our malicious construction). Subsequent to our work, Moataz et al. [90] combined our bounded feedback ORAM with an optimized merge procedure for evictions which reduces server computation and block size for the semi-honest construction. With this development, we argue that semi-honest constant bandwidth blowup ORAM is (nearly) practical. The extent to which we can tighten up poly-logarithmic factors for all constructions (especially the malicious construction) is left open.

Second, beyond tightening parameters, we pose the question of whether constant bandwidth blowup ORAMs can be constructed from non-homomorphic encryption schemes. In Onion ORAM, the computational complexity of the Damgård-Jurik cryptosystem (which relies on modular exponentiation for homomorphic operations), or even more efficient SWHE schemes may be a bottleneck in practice. Can we construct constant bandwidth ORAM using simple computation such as XOR and any semantically secure encryption scheme with small ciphertext blowup? A partial result in this direction is discussed in Chapter 3 and is due to Burst ORAM [73]: simple computation on ciphertexts (mod 2 XOR) enables a family of schemes to achieve constant online bandwidth blowup. Whether similar ideas can lead to constant *overall* bandwidth blowup is unclear.

Bibliography

[1] Open cores. http://opencores.org/.

[2] Sean Gallagher. Your usb cable, the spy: Inside the nsas catalog of surveillance magic. *Ars Technica*.

[3] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345.

[4] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple Oblivious RAM protocol. Cryptology ePrint Archive, Report 2013/280.

[5] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. Cryptology ePrint Archive, Report 2014/672.

[6] R. Rivest, L. Adleman, and M.L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 1978.

[7] Ralph C. Merkle. Protocols for public key cryptography. In *Oakland*, 1980.

[8] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 1986.

[9] O. Goldreich. Towards a theory of software protection and simulation on Oblivious RAMs. In *STOC*, 1987.

[10] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.

[11] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *FOCS*, 1991.

[12] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, 1996.

[13] O. Goldreich and R. Ostrovsky. Software protection and simulation on Oblivious RAMs. In *Journal of the ACM*, 1996.

[14] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS'97*, pages 364–373, 1997.

[15] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, 1997.

[16] Markus G. Kuhn. Cipher instruction search attack on the bus-encryption security microcontroller ds5002fp. *IEEE Trans. Comput.*, 47(10):1153–1157, October 1998.

[17] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[18] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 2000.

[19] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[20] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[21] Ivan Damgard and Mads Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *PKC*, 2001.

[22] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In *CRYPTO*, 2001.

[23] Andrew "bunnie" Huang. Hacking the xbox: An introduction to reverse engineering. 2003.

[24] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[25] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 178–192, 2003.

[26] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17^{th} *ICS (MIT-CSAIL-CSG-Memo-474 is an updated version)*, New-York, June 2003. ACM.

[27] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: An infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS*, 2004.

[28] H. Lipmaa. An Oblivious Transfer protocol with log-squared communication. In *ISC*, 2005.

[29] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the* 32^{nd} *ISCA'05*, New-York, June 2005. ACM.

[30] Amazon. Amazon simple storage service developer's guide. Amazon, 2006.

[31] John L Henning. Spec cpu2006 benchmark descriptions. *Computer Architecture News*, 2006.

[32] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *STC*, 2006.

[33] Alon Itai and Michael Slavkin. Detecting data structures from traces. In *Workshop on Approaches and Applications of Inductive Programming*, 2007.

[34] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *MICRO*, 2007.

[35] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.

[36] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[37] Rafal Wojtczuk and Alexander Tereshkin. Attacking intel bios. *Blackhat*, 2009.

[38] Dan Hubbard and Michael Sutton. Top threats to cloud computing v1. 0. Cloud Security Alliance, 2010.

[39] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA*, 2010.

[40] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In *CRYPTO'11*, 2011.

[41] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, 2011.

[42] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[43] IBM. Ibm 4765 description. Technical report, 2011.

[44] R. Lindner and C. Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Topics in Cryptology - CT-RSA*, 2011.

[45] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 2011.

[46] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log n)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[47] Z. Brakerski, G. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012*, 2012.

[48] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A software-hardware architecture for self-protecting data. In *CCS*, 2012.

[49] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[50] C. Gentry, S. Halevi, and N.P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, 2012.

[51] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In *PKC*, 2012.

[52] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. pages 513–524, 2012.

[53] Mohammad Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[54] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based Oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[55] Adriana Lopez-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*, 2012.

[56] E. Stefanov, E. Shi, and D. Song. Towards practical Oblivious RAM. In *NDSS*, 2012.

[57] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.

[58] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.

[59] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. In *Advances in Cryptology–EUROCRYPT 2013*, pages 279–295. Springer, 2013.

[60] Christopher W. Fletcher. Ascend: An architecture for performing secure computation on encrypted data. In *MIT CSAIL CSG Technical Memo 508 (Master's thesis)*, April 2013.

[61] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PETS*, 2013.

[62] Intel. Software guard extensions programming reference. Intel, 2013.

[63] Seny Kamara. How to search on encrypted data: Oblivious rams (part 4). Blog post, 2013.

[64] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *FAST*, 2013.

[65] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, 2013.

[66] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[67] Ewen Macaskill and Gabriel Dance. The nsa files: Decoded. The Guardian, 2013.

[68] Ling Ren, Christopher Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity verification for Path Oblivious-RAM. In *HPEC*, 2013.

[69] Ling Ren, Xiangyao Yu, Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of Path Oblivious RAM in secure processors. In *ISCA*, 2013.

[70] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *S&P*, 2013.

[71] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple Oblivious RAM protocol. In *CCS*, 2013.

[72] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *PKC*. 2014.

[73] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *USENIX security*, 2014.

[74] Christopher Fletcher, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, 2014.

[75] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, 2014.

[76] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *FOCS*, 2014.

[77] Shai Halevi and Victor Shoup. Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106, 2014.

[78] Martin Maas. Phantom: Practical oblivious computation in a secure processor. In *UC Berkeley EECS Technical Memo 89 (Master's thesis)*, May 2014.

[79] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS*, 2014.

[80] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *POPL*, 2014.

[81] Jinsheng Zhang, Qiumao Ma, Wensheng Zhang, and Daji Qiao. Kt-oram: A bandwidth-efficient oram built on k-ary tree of pir nodes. Cryptology ePrint Archive, Report 2014/624, 2014.

[82] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *CCS*, 2015.

[83] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[84] Jonathan Dautrich and Chinya Ravishankar. Combining ORAM with PIR to minimize bandwidth costs. In *CODASPY*, 2015.

[85] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket oram: Single online roundtrip, constant bandwidth oblivious ram. Cryptology ePrint Archive, Report 2015/1065, 2015. http://eprint.iacr.org/.

[86] Christopher Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based Oblivious RAM. In *ASPLOS*, 2015.

[87] Christopher Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. A low-latency, low-area hardware Oblivious RAM controller. In *FCCM*, 2015.

[88] Shai Halevi and Victor Shoup. Bootstrapping for HElib. In *EUROCRYPT*, 2015.

[89] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. 2015.

[90] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication oram with small blocksize. 2015.

[91] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to Oblivious RAM. In *USENIX security*, 2015.

[92] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, 2015.

[93] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. In *ASPLOS*, 2016.

[94] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016.

[95] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. TCC, 2016.

[96] Ling Ren, Christopher Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Design and implementation of the ascend secure processor. In *TDSC*, 2017.

[97] David Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.