Unified RAW Path Oblivious RAM

by

Ling Ren

B.S. in Electronic Engineering, Tsinghua University, Beijing, China, 2012

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Department of Electrical Engineering and Computer Science

August 29, 2014

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
Unified RAW Path Oblivious RAM
by
Ling Ren

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Oblivious RAM (ORAM) is a cryptographic primitive that conceals memory access patterns
to untrusted storage. Its applications include oblivious cloud storage, trusted processors,
software protection, secure multi-party computation, and so on. This thesis improves the
state-of-the-art Path ORAM in several aspects.

On the theoretical side, we improve Path ORAM’s memory bandwidth overhead by a
factor of $O(\log \log N)$ when the block size is small. With this improvement, Path ORAM
is asymptotically the most efficient ORAM construction with constant or polylogarithmic
client storage under any block size. Our technique to achieve this improvement involves
using pseudorandom functions to compress the position map, a central component in Path
ORAM and other position-based ORAMs. With small block size, managing the position
map has huge overhead and is Path ORAM’s performance bottleneck. Our technique reduces
this overhead.

On the practical side, we propose Unified ORAM with a position map lookaside buffer
to utilize locality in real-world applications, while preserving access pattern privacy. We
also propose a new variant of Path ORAM named RAW Path ORAM, which achieves a
constant factor reduction in memory bandwidth and encryption overhead. It also features a
much simpler proof for correctness compared with Path ORAM. Combining our techniques
results in a roughly $2\times$ improvement in ORAM bandwidth, and over $1.43\times$ speedup on
SPEC benchmarks.

We also study how to efficiently verify the integrity of ORAM. Besides some customized
optimizations for Path ORAM and RAW Path ORAM, we present a novel integrity verifica-
tion scheme that works for any position-based ORAM and achieves an asymptotic reduction
in hashing overhead over prior solutions.

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

This thesis would not have been possible without the collaborative efforts of Srinivas Devadas, Marten van Dijk, Christopher Fletcher, Xiangyao Yu and Albert Kwon. I would like to first thank my advisor Srini. I find myself lucky to have a great advisor who is so energetic and passionate about research, and at the same time gives students complete freedom. It was also Srini who suggested that I work on this topic and more generally in computer security and applied cryptography which are my current interests. Srini, I thank you for the above along with the guidance and support, the careful proofreading, and the pool parties and ice-cream.

Marten, it has been an amazing experience to work with you. You have always been a source of new ideas. I learned from you how to think about security and cryptography, which I had no background in before joining MIT. Chris, you have been a great mentor as well as my closest colleague. I highly admire your dedication and diligence. Many a time, it is seeing you at your desk that motivated me to keep up the hard work. Xiangyao, it is great to have you to work with on everything from research projects, mentoring students to course assignments and exams. Albert, I am amazed by your abilities as a first year student, and particularly enjoyed going over with you the math and subtleties in what became part of this thesis. I learned a lot from all the four of you!

I would like to thank Elaine Shi and Emil Stefanov. Their seminal works laid the foundation of this thesis. Many ideas in this thesis also arise from discussion with them. I have been very lucky to have the chance to collaborate with them.

I would also like to express my gratitude to many people at CSAIL who have helped me in various aspects. Thanks to Rachael Harding, Ilia Lebedev and Nirav Dave for the help and information when I first got here. Thanks to Cheng Cheng and Madars Virza for the help in the cryptography class and the insightful discussion about my research. Thanks to Charles Herder for his knowledge on seemingly everything, to George Kurian, Keun Sup Shim and Omer Khan for their help and contribution, and to the entire Hornet group for sharing exciting ideas and amazing work.

Finally, I would like to thank my parents and my girlfriend. I am grateful to my parents for too many things; I can only say thank you for your support in every aspect of my life. To my girlfriend Hua, thank you for sticking with me despite the distance and time difference.
## Contents

1 Introduction ........................................... 7
   1.1 Thesis Contribution .................................. 9
   1.2 Thesis Organization .................................... 9
   1.3 Notations and Assumed Background Knowledge ................. 10

2 Related Work ........................................ 11
   2.1 Oblivious RAM ........................................ 11
   2.2 ORAM Applications ..................................... 13
      2.2.1 Oblivious Storage .................................. 13
      2.2.2 Trusted Processors ................................ 14
      2.2.3 Other Applications ................................. 14

3 Preliminaries and Background .............................. 15
   3.1 ORAM Definition and Threat Model ......................... 15
   3.2 Basic Path ORAM ....................................... 16
      3.2.1 Construction ....................................... 16
      3.2.2 Privacy ........................................... 18
      3.2.3 Correctness: Stash Overflow Probability ............... 19
      3.2.4 Performance ....................................... 19
   3.3 Recursive Path ORAM ................................... 20
      3.3.1 Block size for PosMap ORAMs ....................... 21
   3.4 Position-based ORAM Interface .......................... 22
      3.4.1 Generalizability of the Techniques ................. 24

4 PosMap Lookaside Buffer .................................. 25
   4.1 Scalability of Recursive ORAM ......................... 25
   4.2 PosMap Lookaside Buffer ............................... 26
      4.2.1 High-Level Ideas .................................. 26
      4.2.2 PLB (In)security .................................. 26
   4.3 Security Fix: Unified ORAM ............................ 27
      4.3.1 Blocks in ORamU .................................... 27
      4.3.2 ORAM readrmv and append Operations ............... 27
      4.3.3 Detailed Algorithm ................................ 27
      4.3.4 Unified ORAM Privacy ............................... 28
   4.4 Comparison with Related Work .......................... 30
5 **Compressed PosMap**  
5.1 Preliminary: Pseudo Random Functions 31  
5.2 Construction 32  
5.3 Asymptotic Improvement 32  
5.4 Practical Improvement 33  

6 **RAW Path ORAM**  
6.1 Overview 35  
6.2 Construction 35  
6.3 Memory Bandwidth and Encryption Overhead 36  
6.3.1 Design Space and Parameter Recommendation 37  
6.4 Correctness: Stash Occupancy Analysis 38  
6.4.1 Proof outline 39  
6.4.2 ∞-ORAM 39  
6.4.3 Average Bucket and Rooted Subtree Load 41  
6.4.4 Chernoff Bound 42  
6.5 Optimization: Non-integer A 43  
6.6 Privacy 44  

7 **Integrity Verification**  
7.1 Overview 45  
7.2 ORAM Integrity Definition 45  
7.3 Recursive ORAM Privacy under Active Adversaries 46  
7.4 For Path ORAM 47  
7.4.1 Basic Path ORAM 47  
7.4.2 Recursive Path ORAM 48  
7.5 For RAW Path ORAM 50  
7.5.1 Preliminary: Message Authentication Codes 50  
7.5.2 Construction 50  
7.5.3 Security 51  
7.6 For All Position-based ORAMs: PosMap MAC 52  
7.6.1 Construction 52  
7.6.2 Adding Encryption: Subtle Attacks and Defences 52  
7.6.3 Security 53  

8 **Evaluation**  
8.1 Methodology and Parameters 55  
8.2 Path ORAM on DRAM 55  
8.3 PLB Design Space 57  
8.4 Final Results 57  

9 **Conclusion**  
59
List of Figures

3-1 A Path ORAM with $L = 3$. ........................................ 17
3-2 An example of a Path ORAM access. ............................... 18
3-3 A recursive Path ORAM access. .................................. 20
4-1 The percentage of data read from PosMap ORAMs. .............. 25
4-2 Unified ORAM frontend with PLB and $\chi = 4$. ................. 28
6-1 Data read vs. decrypted on a RAW Path ORAM RO access and a Path ORAM access. .................................................. 37
6-2 Stash overflow probability for RAW Path ORAM. ............... 38
6-3 Memory bandwidth and encryption overhead of RAW Path ORAM. 38
6-4 Memory bandwidth and encryption overhead of RAW Path ORAM with non-integer $A$. ............................................. 43
7-1 Integrity verification of a single Path ORAM. ....................... 48
8-1 Illustration of subtree locality. ....................................... 56
8-2 ORAM latency in nano seconds assuming one to eight DRAM channel(s). 57
8-3 Sweep study for direct-mapped PLB capacity. Runtime is normalized to the 8 KB PLB point. ............................................. 57
8-4 Performance of PLB, Compressed PosMap and PMMAC. Slowdown is relative to a conventional system with DRAM. ..................... 58
8-5 Scalability to large ORAM capacities. White shaded regions indicate data movement from PosMap ORAMs. Slowdown is relative to a conventional system with DRAM. ..................... 58
List of Tables

1.1 Asymptotic and empirical memory bandwidth overhead after applying the techniques in this thesis. The empirical results come from Section 8.4. . . . 9
1.2 Variables with consistent meaning throughout the thesis. . . . . . . . . . . . 10
2.1 Performance of previous ORAM constructions. . . . . . . . . . . . . . . . . 12
5.1 Asymptotic improvement from compressed PosMap. . . . . . . . . . . . . 31
8.1 Experimental setup. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
Chapter 1

Introduction

Outsourcing storage and computation to cloud service providers is becoming more and more popular. Such services include Amazon S3, Google Drive and Dropbox for cloud storage, and Amazon EC2, Google Compute Engine and Microsoft Azure for cloud computing, just to name a few. While these services offer cost-effectiveness, robustness (due to redundancy) and convenience like mobility, they also pose new risks to users’ privacy. Users now have to trust the service providers not to leak their private data. According to a recent survey [44], in 2011 alone there were 71 reported cloud vulnerability incidents, and this number has been doubling roughly every two years since 2008. To make matters worse, service providers themselves can be malicious. “Malicious insiders” is listed among the top threats to cloud computing [23]. Moreover, without being intentionally malicious, service providers could mistakenly leak users’ private data, due to bad protocol design or buggy implementation for example.

The first countermeasure that comes to mind is encryption. In the cloud storage case, a user Alice can encrypt all her data using a private key known only to herself. For computation outsourcing, it is less straightforward to apply encryption since the server cannot compute on encrypted data (unless one uses fully homomorphic encryption [21] which is very expensive). One solution is to have trusted and tamper-resistant secure processors on the server side [32]. In this model, the user Alice sets up a secure channel with a trusted processor and sends her private data. The trusted processor decrypts the data, performs the computation and then sends the encrypted results back to Alice. It is assumed that adversaries cannot look inside the tamper-resistant secure processor. Apart from the communication channel to the user, the secure processor also needs to protect its interaction with external memory, by encrypting all the data sent to the external memory [10, 13] for example.

However, encryption alone is not sufficient to protect privacy. Access patterns, i.e., the addresses of memory accesses, also leak information. Below is a very simple example program that leaks sensitive data through its memory access pattern. Suppose \( x \) is private user data and \( a \) is some array with public starting address.

```plaintext
function Leaky(x, a):
    read a[x]
```

Even if the processor encrypts the external memory, the address of this memory access leaks the value of \( x \). For more realistic attacks, Zhuang et al. [15] showed that memory access patterns can leak a program’s control flow graph, and therefore leak sensitive data through conditional branches. Islam et al. [36] demonstrated that an adversary can infer 80% of the
search queries to an encrypted email repository from access patterns.

Oblivious RAM (ORAM) is a cryptographic primitive that conceals memory access patterns. ORAM continuously reshuffles the memory and translates the address of each memory access to a set of randomized memory locations. Provably, these randomized memory locations are guaranteed to be independent of, and thus leak no information about, the logical addresses that are actually requested. In the cloud storage scenario, if Alice accesses her private data on a remote server using an ORAM algorithm, the server or any other eavesdroppers in the network cannot learn any information about Alice’s data through her access pattern. Likewise in computation outsourcing, a server-side trusted secure processor that accesses external memory through an ORAM interface does not need to trust the external memory and will not leak any information from memory access patterns.

The strong security guarantee of ORAM comes with great cost. ORAM has long been assumed to be too expensive for practical applications. Besides, most early ORAM constructions were quite complicated and hard to implement, which also limits their use in practice. Only recently, ORAM’s performance and usability has been significantly improved. We review the most relevant previous works on ORAM in Section 2.1. Particularly interesting among them is a class of ORAM constructions called tree-based ORAMs or more generally position-based ORAMs, first proposed by Shi et al. [31]. Position-based ORAMs enjoy remarkable simplicity as well as competitive performance, and therefore show great potential for real-world applications. Position-based ORAMs have recently been adopted in both oblivious remote storage [46, 53, 58], and secure processor proposals [32, 56, 49, 48]. Notably, Path ORAM [55, 54] is currently the most efficient position-based ORAM, and is also conceptually simple and easy to implement.

This thesis further improves position-based ORAMs in several aspects. Before presenting the contributions of this thesis, let us first take a look at the current inefficiencies and bottlenecks of ORAM.

First of all, memory bandwidth is always a bottleneck of ORAM. All ORAMs are subject to a $O(\log N)$ lower bound on memory bandwidth overhead [4, 7], where $N$ is the number of blocks in ORAM. Although Path ORAM has already achieved the $O(\log N)$ lower bound in some settings [55, 54], it is still desirable to improve the hidden constant factor in it, which can make a big difference in practice.

Moreover, the above memory bandwidth is achieved either at the cost of huge client storage, or assuming a large block size. Basic position-based ORAMs need to store in trusted storage a large data structure called Position Map (PosMap for short), whose size is proportional to the number of blocks $N$. To achieve small client storage, they need a technique called recursion, or recursive ORAM. The idea is to recursively store the large PosMap in additional ORAMs to reduce the client storage. With a large data block size, recursion does not increase memory bandwidth overhead too much. But when the data block size is small, recursion dominates the memory bandwidth overhead, and makes position-based ORAMs asymptotically less efficient. Section 4.1 gives some concrete numbers: with reasonable parameters for secure processors, recursion can account for more than 60% of the memory bandwidth overhead of Path ORAM.

This thesis also considers integrity of ORAM. Despite being very important to ORAM constructions, most previous works simply commented that integrity could be achieved using standard solutions like Merkle trees [2]. While this is certainly true, naively using a Merkle tree can lead to great performance loss.
Table 1.1: Asymptotic and empirical memory bandwidth overhead after applying the techniques in this thesis. The empirical results come from Section 8.4.

<table>
<thead>
<tr>
<th>ORAM scheme</th>
<th>Asymptotic</th>
<th>Empirical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive Path ORAM [54]</td>
<td>$O \left( \log N + \frac{\log^3 N}{B} \right)$</td>
<td>236×</td>
</tr>
<tr>
<td>Unified Path ORAM + PLB</td>
<td>$O \left( \log N + \frac{\log^3 N}{B} \right)$</td>
<td>167×</td>
</tr>
<tr>
<td>Unified Path ORAM + PLB + Compression</td>
<td>$O \left( \log N + \frac{\log^3 N}{B \log \log N} \right)$</td>
<td>145×</td>
</tr>
<tr>
<td>Unified RAW Path ORAM + PLB + Compression</td>
<td>$O \left( \log N + \frac{\log^3 N}{B \log \log N} \right)$</td>
<td>~121×</td>
</tr>
</tbody>
</table>

### 1.1 Thesis Contribution

This thesis proposes the following three techniques to improve the state-of-the-art position-based ORAMs.

- **Unified ORAM** with a PosMap Lookaside Buffer (PLB) to reduce the number of PosMap accesses,
- **PosMap compression** using pseudorandom functions, which improves memory bandwidth both asymptotically and empirically, and
- **RAW Path ORAM**, which has a constant-factor improvement in memory bandwidth and encryption overhead over Path ORAM.

Table 1.1 summarizes the improvements from the above techniques. On the theoretical side, PosMap compression improves Path ORAM’s memory bandwidth by a $O(\log \log N)$ factor when the block size is small. With this improvement, Path ORAM with compressed PosMap has the best memory bandwidth among ORAM constructions with constant or polylogarithmic client storage under any block size.

On the practical side, Unified ORAM with PLB and compressed PosMap eliminates over 80% of the PosMap accesses, reducing overall ORAM memory bandwidth overhead by 38%, and leads to a 1.43× average speedup on SPEC benchmarks. RAW Path ORAM with a competitive parameter setting further improves memory bandwidth by 31% and at the same time reduces encryption overhead by 69%. In addition, RAW Path ORAM features a much simpler proof for correctness compared to Path ORAM.

Additionally, we present a novel integrity verification scheme called PosMap MAC, that works for any position-based ORAM and achieves an asymptotic reduction in hashing. We also present customized optimizations on integrity verification for Path ORAM and RAW Path ORAM.

### 1.2 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 reviews related work. Chapter 3 gives ORAM’s definition and threat model, and introduces Path ORAM as an example of position-based ORAMs. Chapter 4 presents PosMap Lookaside Buffer and Unified ORAM. Chapter 5 describes compressed PosMap and shows its improvements. Chapter 6 presents RAW Path ORAM. Chapter 7 proposes optimizations on ORAM integrity verification,
Table 1.2: Variables with consistent meaning throughout the thesis.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Defined in Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Number of blocks in an ORAM</td>
<td>2.1</td>
</tr>
<tr>
<td>$B$</td>
<td>Data block size in bits of an ORAM</td>
<td>2.1</td>
</tr>
<tr>
<td>$L$</td>
<td>Depth of Path ORAM tree</td>
<td>3.2</td>
</tr>
<tr>
<td>$Z$</td>
<td>Number of blocks in a bucket</td>
<td>3.2</td>
</tr>
<tr>
<td>$R$</td>
<td>Maximum number of blocks in the stash</td>
<td>3.2</td>
</tr>
<tr>
<td>$P(l)$</td>
<td>Path to leaf $l$ in Path ORAM</td>
<td>3.2</td>
</tr>
<tr>
<td>$P(l)[i]$</td>
<td>$i$-th bucket on Path $P(l)$ in Path ORAM</td>
<td>3.4</td>
</tr>
<tr>
<td>$\chi$</td>
<td>Number of labels in a PosMap block</td>
<td>3.3</td>
</tr>
<tr>
<td>$H$</td>
<td>Number of ORAMs in recursion</td>
<td>3.3</td>
</tr>
<tr>
<td>$B_p$</td>
<td>PosMap block size in bits</td>
<td>3.3.1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Group counter width in compressed PosMap</td>
<td>5.2</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Individual counter width in compressed PosMap</td>
<td>5.2</td>
</tr>
<tr>
<td>$A$</td>
<td>Ratio of RO and RW accesses in RAW Path ORAM</td>
<td>6.2</td>
</tr>
</tbody>
</table>

including PosMap MAC. Chapter 8 evaluates the performance of PLB, compressed PosMap and PosMap MAC. Chapter 9 concludes the thesis.

1.3 Notations and Assumed Background Knowledge

The variables in Table 1.2 (mostly capital letters) will have consistent meanings throughout the thesis. For example, $N$ is always the number of blocks and $B$ always the block size in an ORAM. All the symbols of Path ORAM are also used in RAW Path ORAM, while the inverse is not true. Other variables (mostly lower cases) are temporary variables that may have different meanings in different contexts/sections.

This thesis requires no background on Oblivious RAM. One focus of the thesis is to analyze and improve the complexity of ORAM algorithms, so we expect the readers to be familiar with the notations in algorithm complexity, especially the “big O” notations. We also assume basic knowledge in cryptography like secret keys, encryption and hash functions. For more advanced cryptographic primitives including pseudorandom functions, Merkle trees and message authentication codes, we will give a brief introduction before using them, but it is still recommended that readers refer to more detailed materials.

While this thesis optimizes ORAM as a primitive, parts of this thesis including the evaluation will assume (or draw analogy to) a real-world processor setting. For a better understanding of these contents, it will be helpful to have basic knowledge in computer architecture, such as cache, page tables and TLB, memory controllers and DRAM.
Chapter 2

Related Work

2.1 Oblivious RAM

The first ORAM construction is the “square-root” solution by Goldreich [4], which was based on hash tables and oblivious shuffling. Goldreich’s paper also established an $O(\log N)$ lower bound on ORAM’s memory bandwidth overhead. Since then, there have been numerous follow-up works that significantly improved ORAM’s efficiency in the past three decades. In this section, we review the most relevant works, and list their performance in Table 2.1.

ORAM performance is usually measured by three metrics: memory bandwidth overhead, client storage and server storage. Memory bandwidth overhead, or simply bandwidth overhead, is the extra amount of data an ORAM algorithm has to access in order to achieve obliviousness. Client storage is the amount of private and trusted storage an ORAM algorithm needs. The ORAM threat model assumes that adversaries cannot see the contents of the client storage, nor the access pattern to it. Server storage is the amount of public and untrusted storage an ORAM algorithm requires. Adversaries can observe the contents of the server storage and the access pattern to it at any time. Suppose the logical memory space has $N$ addresses (also often referred to as blocks, items or words), and each address stores $B$ bits of data. Memory bandwidth overhead, client storage and server storage are usually functions of $N$ and $B$. Following conventions in prior work, client storage and server storage are measured in the unit of $B$-bit blocks in Table 2.1 and throughout the thesis.

Ostrovsky [5] (and later with Goldreich [7]) presented the famous ‘hierarchical’ solution, which is the first ORAM construction with polylogarithmic memory bandwidth overhead. It achieved $O(\log^3 N)$ memory bandwidth overhead, with $O(1)$ client storage and $O(N \log N)$ server storage. This construction got the name ‘hierarchical’ because it uses a hierarchy of buffers whose sizes grow at a geometric rate. Each buffer is a hash table that keeps the blocks in it randomly shuffled. Each access goes through all the buffers, retrieves the requested block from its shuffled location (determined by the hash tables), and puts it in the smallest buffer. Smaller buffers are shuffled into large ones as they fill up.

Ostrovsky’s hierarchical construction inspired many subsequent works. Williams and Sion [19] reduced the bandwidth overhead to $O(\log^2 N)$ using oblivious sorting, at the cost of $O(\sqrt{N})$ client storage. Goodrich, Mitzenmacher and others [28, 34] used cuckoo hashing to further reduce the bandwidth overhead to $O(\log N)$ with $O(N^\epsilon)$ ($\epsilon > 0$) client storage. The authors also presented a construction with constant client storage and $O(\log^2 N)$ bandwidth overhead. Using similar techniques as Goodrich and Mitzenmacher [28], Kushilevitz et al. [37] improved the memory bandwidth to $O(\log^2 N / \log \log N)$ under constant client storage.
Table 2.1: Performance of previous ORAM constructions.

<table>
<thead>
<tr>
<th>ORAM scheme</th>
<th>Avg Memory Bandwidth Overhead</th>
<th>Client Storage</th>
<th>Server Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hierarchical ORAMs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GO [7]</td>
<td>$O \left( \log^{3} N \right)$</td>
<td>$O(1)$</td>
<td>$O(N \log N)$</td>
</tr>
<tr>
<td>WS [19]</td>
<td>$O \left( \log^{2} N \right)$</td>
<td>$O(\sqrt{N})$</td>
<td>$O(N \log N)$</td>
</tr>
<tr>
<td>GM [28]</td>
<td>$O \left( \log N \right)$</td>
<td>$O(N^\epsilon)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td></td>
<td>$O \left( \log^2 N \right)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>KLO [37]</td>
<td>$O \left( \log^2 N/\log \log N \right)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td><strong>Position-based ORAMs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSS [39]</td>
<td>$O \left( \log^2 N \right)$</td>
<td>$O(\sqrt{N})$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>SCSL [31]</td>
<td>$O \left( \log^3 N \right)$</td>
<td>$O(1)$</td>
<td>$O(N \log N)$</td>
</tr>
<tr>
<td>Gentry et al. [43]</td>
<td>$O \left( \log^3 N/\log \log N \right)$</td>
<td>$O \left( \log^2 N \right)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Path ORAM [54]</td>
<td>$O \left( \log N + \frac{\log^3 N}{B \log \log N} \right)$</td>
<td>$O \left( \log N \right) \omega(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>This work</td>
<td>$O \left( \log N + \frac{\log^3 N}{B \log \log N} \right)$</td>
<td>$O \left( \log N \right) \omega(1)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

† Several notes about the table. While we put SSS under position-based ORAMs, it actually combined the ideas of hierarchical solution and position-based ORAMs. For SCSL and Gentry et al., we list their trivial bucket version with recursion. All the constructions with no explicit mention of block size $B$ at least assumed $B = \Omega(\log N)$. The $\omega(1)$ term in Path ORAM and this work will be explained in Section 3.2.3.

To summarize, in the hierarchical ORAM framework, blocks in each buffer are shuffled by applying pseudorandom permutations, standard hash functions or cuckoo hashes on their logical addresses, and all the buffers are reshuffled regularly, as the blocks are accessed and their randomized locations revealed. The best memory bandwidth overhead achieved by hierarchical solutions is $O(\log N)$ under large client storage [28], and $O(\log^2 N/\log \log N)$ under small (constant in this case) client storage [37].

While most ORAM constructions followed Ostrovsky’s hierarchical ORAM idea, Stefanov, Shi and others [39, 31] took a very different approach. We use the binary-tree ORAM of Shi et al. [31] to explain their high-level ideas. In this construction, each data block is mapped to a random path of a binary tree via a huge table, known as the position map or PosMap for short (though it got that name from the other paper [39]). For this reason, we also call these ORAM constructions position-based ORAMs. In this framework, a block can be reshuffled simply by updating its corresponding entry in the PosMap. This approach by design needs no hash tables, cuckoo hashing, or the expensive reshuffling and oblivious sorting operations. When a block is accessed, it is put to the root of the tree. Hence, some eviction procedures are needed to percolate the blocks towards the leaves of the tree and prevent the root from overflowing. A downside of position-based ORAMs is the $\omega(N)$ size PosMap. A core idea of Shi et al. [31] is the recursion technique to reduce client storage: the huge PosMap is recursively stored in additional ORAMs until it finally becomes $O(1)$ in size. Although these two early position-based ORAMs did not beat the mature hierarchical solutions like Kushilevitz et al. [37], they opened new directions and inspired follow-up work.

Gentry et al. [43] optimized the tree-based ORAM [31] in several ways, and obtained
an $O(\log \log N)$ factor improvement. In their work, the most relevant technique to us is an eviction strategy called reverse lexicographic order. This strategy greatly improved the eviction quality and inspired our RAW Path ORAM in Chapter 6.

The breakthrough of position-based ORAMs comes in Path ORAM [38, 55, 54], an extremely simple and efficient ORAM protocol. In Table 2.1, we report its memory bandwidth overhead in the most general form $O(\log N + \frac{\log^3 N}{B})$, which depends on both $N$ and $B$. Path ORAM for the first time achieved $O(\log N)$ memory bandwidth overhead under small client storage, though requiring the block size to be at least $B = \Omega(\log^2 N)$. Here we define ‘small’ to be constant or polylogarithmic. It is worth pointing out that the other ORAMs in the table also have requirements on the block size $B$. All of them at least assumed $B = \Omega(\log N)$.

Yet it should be noted that Path ORAM does not always have the best memory bandwidth even among constructions with small client storage. When the block size is small $B = o(\log N \log \log N)$ but still at least $\Omega(\log N)$, Path ORAM’s memory bandwidth is worse than that of Kushilevitz et al. [37]. This thesis improves Path ORAM’s memory bandwidth to $O(\log N + \frac{\log^3 N}{B \log \log N})$. This result always achieves the best memory bandwidth (possibly breaking even with other schemes) under small client storage. For $B = \Theta(\log N)$, it is a tie with Kushilevitz et al. [37], though admittedly requiring more client storage; for $B = \Omega(\log^2 N)$, it breaks even with the original recursive Path ORAM; for anything in between, it beats all the other constructions with small client storage.

We omitted some works in this brief review for various reasons. Two hierarchical ORAM solutions [20, 25] suffered from some security flaws [37]. A lot of concurrent works come out as online reports during the preparation of this thesis. While some of them have interesting ideas, we will skip them as they have not been peer-reviewed. Ajtai [22] and Damg˚ard et al. [27] assumed a very different model: they investigated how to construct ORAM without cryptographic assumptions. Other works had a different focus from us, such as minimizing the number of round-trips [26, 33, 40] or amortizing the worst-case overhead [29].

## 2.2 ORAM Applications

Small client storage and large client storage are obviously two very different cases for ORAM algorithms. They also correspond to different applications. In trusted processors, since processors’ on-chip storage is so expensive, small client storage is strongly preferred. In the oblivious storage case, however, clients can use the main memory or the disk of their computers, and may be willing to spend more client storage in exchange for lower memory bandwidth overhead.

### 2.2.1 Oblivious Storage

Notable oblivious storage implementations include PrivateFS [41], Shroud [46] and ObliviStore [53]. PrivateFS used a hierarchical ORAM solution while the other two adopted position-based ORAMs. All three works looked into parallelism in ORAM. ObliviStore, which is based on the the ORAM algorithm of Stefanov et al. [39] with careful engineering claims the best memory throughput. Stefanov and Shi later showed that ObliviStore’s performance can be significantly improved assuming two non-colluding servers in the cloud [52]. A recent work named Burst ORAM [58], also built on top of ObliviStore, optimizes response time instead of throughput.
2.2.2 Trusted Processors

We have introduced in Chapter 1 the computation outsourcing use case for trusted processors. Aegis [13, 16] aimed to verify its computation and protect user’s private data. Earlier than that, researchers considered using trusted processors for software protection, i.e., to prevent unauthorized copy of software. This was the motivation of Goldreich’s initial ORAM paper [4]: to make a trusted processor’s interaction with memory oblivious, i.e., independent of the software running in the processor. XOM (eXecute Only Memory) [10, 11, 12] took a step by encrypting the memory and adding related architectural support. As access patterns leak information in both applications [15], Aegis and XOM would have stronger security had they been equipped with ORAMs. It is worth mentioning that HIDE [15] applied random shuffling to small chunks of memory to mitigate access pattern leakage. It is much more efficient than ORAM but provides weaker (not cryptographic-level) security.

Ascend [32, 56] is a holistic secure processor proposal the author is currently involved with. Ascend extends the threat model of computation outsourcing to untrusted programs: the server can run any program of its choosing, so the program can be intentionally malicious and leaky like the one in the introduction. For this reason, Ascend adopts memory obfuscation by ORAM as well as timing obfuscation [59]. Phantom [49, 48] is the first hardware implementation of Path ORAM on FPGA. Both Ascend and Phantom use Path ORAM [55] because it is simple and suitable for hardware implementation.

2.2.3 Other Applications

Apart from oblivious storage and secure processors, ORAM also finds applications in secure multi-party computation using RAM programs [35, 47], private information retrieval (PIR) [8, 14] and proof of retrievability [42].
Chapter 3

Preliminaries and Background

3.1 ORAM Definition and Threat Model

At a high level, Oblivious RAM (ORAM) is an algorithm that hides access pattern to untrusted storage. Intuitively, it should not leak 1) which address is being accessed; 2) how old it is (when it was last accessed); 3) access pattern (sequential, random, etc); or 4) whether the access is a read or a write.

In this section, we formally define ORAM. To do so, we first need to define RAM (Random Access Memory) operation and sequence.

Definition 1 (RAM request). A RAM request is an \((\text{addr}, \text{op}, \text{data})\) triplet. The operation \(\text{op}\) is either ‘read’ or ‘write’. On a read, RAM returns the data stored at the requested address \(\text{addr}\) (data is ignored on a read). On a write, RAM updates the data stored at address \(\text{addr}\) to \(\text{data}\).

A sequence of RAM requests form a RAM request sequence, \(\text{\tilde{x}} = (x_m, x_{m-1}, \cdots, x_2, x_1)\), where \(x_i = (\text{addr}_i, \text{op}_i, \text{data}_i)\). \(x_1\) is the first operation that’s sent to the RAM, and \(x_m\) is the last.

ORAM is an algorithm that responds to an input RAM request sequence \(\text{\tilde{x}}\), with the help of an external untrusted RAM along with some internal trusted storage. ORAM sends a (obfuscated) RAM request sequence \(\text{\text{ORAM}}(\text{\tilde{x}})\) to the external untrusted RAM and gets back responses. The external untrusted RAM, also known as server storage, can be read by adversaries at any time, and the request sequence \(\text{\text{ORAM}}(\text{\tilde{x}})\) is also observable to adversaries. ORAM is allowed to keep some secret states in its internal trusted storage, also known as client storage. Adversaries cannot see the content of client storage or the access pattern to it. We want an ORAM algorithm to provide correctness and access pattern privacy.

Correctness requires that ORAM responds to the input RAM request sequence as a valid RAM with all but negligible probability. A RAM is valid if upon any request in a RAM request sequence, it always returns the most recent data that is written to the corresponding address.

Definition 2 (Valid RAM). A RAM is a valid RAM, if given an input RAM request sequence \(\text{\tilde{x}}\), the RAM responds (to read operations) in the following way: \(\forall i, \text{if op}_i = \text{‘read’},\) the RAM returns \(d_i = \text{data}_j\) where \(j\) is the maximum integer such that \(j < i\), \(\text{op}_j = \text{‘write’}\) and \(\text{addr}_j = \text{addr}_i\).
Access pattern privacy (under passive adversaries) requires that the output ORAM sequence $\text{ORAM}(\vec{x})$ reveals no information about the input sequence $\vec{x}$, except a small amount of information related to the input length $|\vec{x}|$.

**Definition 3 (ORAM privacy).** An ORAM algorithm achieves access pattern privacy if for any two input sequences $\vec{x}$ and $\vec{x}'$ of equal length (i.e., $|\vec{x}| = |\vec{x}'|$), $\text{ORAM}(\vec{x})$ and $\text{ORAM}(\vec{x}')$ are computationally indistinguishable.

This definition captures the intuition at the beginning of this section. For any access pattern, whether it is random, sequential or any other pattern, its resulting sequence coming out of ORAM is indistinguishable from that of any other input sequence of the same length. But ORAM does not protect the length of the input sequence. $|\text{ORAM}(\vec{x})|$ is usually solely determined by $|\vec{x}|$ and vice versa. Therefore, ORAM only leaks the length of the input sequence. A good analogy is that encryption leaks only the length of the message.

We would like to remark that the indistinguishability of $\text{ORAM}(\vec{x})$ and $\text{ORAM}(\vec{x}')$ can also be interpreted in a probabilistic sense. By that we mean $|\text{ORAM}(\vec{x})|$ does not have to be completely determined by $|\vec{x}|$; instead, $|\text{ORAM}(\vec{x})|$ can be a random variable whose distribution is fully determined by $|\vec{x}|$. We will see in Section 6.5 that this interpretation enables a simple optimization that leads to a constant-factor improvement.

In some cases, we also want an ORAM to have integrity under active adversaries that may tamper with server storage. Chapter 7 will focus on integrity. But before that chapter, we assume passive adversaries who do not tamper with server storage.

As mentioned, the focus of this thesis is a class of ORAM algorithms called position-based ORAMs. In the next section, we introduce Path ORAM as an example of position-based ORAMs to provide readers with more concrete background.

## 3.2 Basic Path ORAM

### 3.2.1 Construction

#### 3.2.1.1 Components

In Path ORAM, the untrusted server storage is logically structured as a binary tree, as shown in Figure 3-1. We refer to it as Path ORAM tree or ORAM tree for short. The root of the tree is referred to as level 0, and the leaf level as level $L$ (we say such a tree has depth $L$). Each node in the tree is called a *bucket* and holds $Z$ blocks. If a bucket has less than $Z$ real blocks, the rest of the slots are filled with *dummy blocks*. A dummy block is conceptually an empty slot that can be taken up by a real block at any time. All the blocks in the tree including the dummy blocks are encrypted with probabilistic encryption, so any two blocks (dummy or real) are indistinguishable after encryption. Each leaf node has a unique leaf label $l$.

The Path ORAM controller has a *position map*, a *stash* and associated control logic. The position map (PosMap for short) is a lookup table that associates each block with a random leaf in the ORAM tree. The stash is a memory that stores up to a small number of blocks (denoted as $R$) that temporarily cannot be put back into the ORAM tree.
3.2.1.2 Invariant and Operation

At any time, each block in Path ORAM is mapped to a random leaf via the PosMap. Path ORAM maintains the following invariant: If a block is mapped to leaf \( l \), then it must be either in some bucket along the path from the root to leaf \( l \) or in the stash. In Figure 3-1 for example, blocks mapped to leaf \( l = 5 \) can be located in any of the shaded structures at any time. Since a leaf uniquely determines a path in a binary tree, we will use leaf and path interchangeably. We also refer to the path from the root to leaf \( l \) as path \( l \), or \( P(l) \).

The ORAM controller serves RAM requests \((a, op, d')\) where \( a \) is the address of the requested block, \( op \) is either read or write, and \( d' \) is the new data if \( op = \text{write} \). To serve such a request, the ORAM controller carries out the following steps, which we denote as \( \text{accessORAM}(a, op, d') \),

1. Look up the PosMap with \( a \), yielding the corresponding leaf label \( l \).
2. Assign a new random leaf \( l' \) to block \( a \) and update the PosMap.
3. 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.
4. Update block \( a \) in the stash to have leaf \( l' \). If \( op = \text{read} \), return block \( a \); if \( op = \text{write} \), replace the data of block \( a \) with \( d' \).
5. Evict and encrypt as many blocks as possible from the stash to path \( l \) in the ORAM tree to keep the stash occupancy low. This means for every block in the stash, put the block to the furthest bucket (closest to the leaf) that it can reside in without violating the invariant. Fill any remaining space on the path with encrypted dummy blocks.

Figure 3-2 gives an example of Path ORAM operations with \( Z = 1 \) and \( R = 4 \). Each block has the format: (block identifier, leaf label). For example, \((b, 3)\) means block \( b \) is
mapped to path 3. Steps 1-3 read block $b$ and steps 4-6 write block $b'$ to $b'$. We elaborate on the path write back operation using the top right box (# 3) in Figure 3-2 as an example. $(b, 1)$ can only go to the root bucket since it only shares the root bucket in common with path 3; $(c, 2)$ can no longer be written back to the tree at all since it only shares the root bucket with path 3, and the root bucket is now full; $(d, 4)$ can be mapped back to the bucket between the leaf and the root; no block goes to the leaf bucket on path 3, so that bucket needs to be filled up with the encryption of a dummy block. To guarantee obliviousness, the ORAM controller should write back the path in a data-independent way (e.g., always from the root to the leaf), after all of the above computation is done. An adversary only sees the ORAM controller reads and writes two random paths (in this case path 3 and path 1).

3.2.1.3 Probabilistic Encryption

The protocol requires probabilistic encryption over each block (including dummy blocks) in external memory. This ensures that ciphertexts along the accessed path will always change, no matter whether or not the plaintexts stay the same.

We will use the following AES counter-mode [9] probabilistic encryption scheme. To encrypt a bucket, break up the plaintext into chunks, and apply the following one-time pad

\[
\text{AES}_K(BID || Ctr || i) \oplus \text{chunk}_i
\]

In the above formula, $BID$ is a unique identifier for each bucket in the ORAM tree. $Ctr$ is a counter that increments every time the bucket is re-encrypted. $Ctr$ can be either a per-bucket counter or a global counter shared by all buckets. It is required that we never reuse the same counter value, so $Ctr$ should be large enough (64-bit or 128-bit in practice) such that it never rolls over. The encrypted bucket is the concatenation of each one-time padded chunk along with the $Ctr$ value in the clear. This probabilistic encryption scheme adds minimum storage overhead to each bucket.

3.2.2 Privacy

The path read and write operation (Step 3 and 5) are done in a data-independent way (e.g., always from the root to the leaf); and due to probabilistic encryption (Section 3.2.1.3), all the ciphertexts along the path change. Therefore, the leaf label $l$ is the only information revealed to an observer on an ORAM access.
Step 2 is the key to Path ORAM’s privacy: a block is remapped to a new random leaf whenever it is accessed. This guarantees that the PosMap always contains fresh random leaf labels for all the blocks. Then, a random path \( l \), retrieved from the PosMap, is read and written on every access regardless of the actual program address requested. Thus, Path ORAM satisfies the ORAM privacy definition (Definition 3).

### 3.2.3 Correctness: Stash Overflow Probability

While Path ORAM obviously achieves access pattern privacy, we still need to prove its correctness, which requires upper bounding the stash occupancy. We say Path ORAM fails when its stash overflows. In the case of a failure, Path ORAM may lose some blocks and return functionally incorrect results at any point in the future. In order for Path ORAM to qualify as a valid memory (Definition 2), we need to show that the stash overflow probability is negligible.

This step, however, turns out to be non-trivial. Stefanov and Shi came up with the Path ORAM algorithm in early 2012 [38], but were not able to bound the stash overflow probability at the time. This prevented the paper’s publication till late 2013, when it was finally proved that for Path ORAM with \( Z = 6 \) the stash overflow probability decreases exponentially with the stash size [55]. This means we can choose the stash size to be \( \Theta(\log N)\omega(1) \) to get a stash overflow probability of \( O(N^{-\omega(1)}) \), which is negligible in \( N \) and the security parameter. Later, \( Z = 5 \) was also proven to have an exponentially decreasing stash overflow probability [54]. Both proofs are highly skillful and complicated.

Note that \( Z \) directly determines the hidden constant factor in Path ORAM complexity (Section 3.2.4) and the practical performance, so it is desirable to prove correctness for smaller \( Z \). Experimental results [51, 49] suggest that \( Z = 4 \) also achieves negligible stash overflow probability. Yet, this remains an open problem up to the date this thesis is written.

### 3.2.4 Performance

In this thesis, we care about both theoretical (asymptotic) and practical (empirical) performance.

#### 3.2.4.1 Storage

Path ORAM’s server storage is proven to be just a constant factor worse than \( N \cdot B \) bits [55]. Path ORAM’s client storage includes two parts: the PosMap and the stash. The PosMap for a basic Path ORAM is \( O(N \log N) \) bits; with recursion, it can be reduced to \( O(1) \) (Section 3.3). The stash has to be at least \( \Theta(\log N)\omega(1) \) to ensure negligible overflow probability. This explains the \( \omega(1) \) factor in Path ORAM’s client storage in Table 2.1. In practice, experiments suggest that a stash size around 100 is usually enough. So Path ORAM’s client storage, though not \( O(1) \), is not a big problem in practice.

#### 3.2.4.2 Memory Bandwidth

The Path ORAM tree and the stash have to store a (address, leaf, data) triplet for each data block. Let each block contain \( B \) bits of data, and be stored alongside with its \( U_1 \)-bit program address and \( U_2 \)-bit leaf label. Each bucket contains \( Z \) blocks and a counter for probabilistic encryption (Section 3.2.1.3). On each Path ORAM access, the ORAM controller has to read and write an entire path. We define \( \text{PathLen} \) to be the amount of data on a path. In an
ORAM tree with depth \( L \), each path contains \( L + 1 \) buckets. So the amount of data on a path (in bit) is

\[
\text{PathLen} = (L + 1)[Z(B + U_1 + U_2) + \text{CtrWidth}]
\]  

(3.1)

Path ORAM latency and energy overhead in practice directly relates to the PathLen metric. The dominant part of Path ORAM latency is the time it takes to transfer PathLen bits of data between the external ORAM tree and the ORAM controller. Path ORAM energy mainly consists of three parts: (1) to access the external memory that stores the ORAM tree, (2) to transfer the data to and from ORAM controller and (3) to decrypt/encrypt the data. All three parts are linear in PathLen.

Now we analyze the asymptotic bandwidth overhead of Path ORAM. Let \( N \) be the number of data blocks in ORAM. Then we need \( U_1 = U_2 = \log N \). Now it makes sense to use counters also of \( O(\log N) \) bits not to further increase the asymptotic complexity. Path ORAM’s asymptotic bandwidth overhead is then given by \( 2 \cdot \text{PathLen}/B = O(\log^2 N/B) \). If we assume the block size \( B = \Omega(\log N) \), which is usually true in practice, Path ORAM’s asymptotic bandwidth overhead is then \( O(\log N) \).

We remark that counters of \( O(\log N) \) bits do overflow in polynomial time. If that happens, all blocks are copied into a new Path ORAM and the algorithm resumes. This process takes \( O(\log N) \) Path ORAM operations, and can be amortized to \( o(1) \). This was also implicitly assumed in the original Path ORAM paper [55, 54]. We also remark that this is mostly a theoretical construction to achieve better asymptotic results. In practice, we can simply use counters wide enough (like 64-bit) so they never overflow.

### 3.3 Recursive Path ORAM

The \( N \log N \)-bit position map is usually too large, especially for a secure processor’s on-chip storage. For example, a 4 GB Path ORAM with a block size of 128 bytes and \( Z = 4 \) has a position map of 93 MB. Recursion [31] solves this problem by storing the PosMap in additional ORAMs.
We refer to the original data Path ORAM as the data ORAM denoted as \( \text{ORam}_0 \), and the second Path ORAM as a PosMap ORAM denoted as \( \text{ORam}_1 \). Suppose each block in \( \text{ORam}_1 \) contains \( \chi \) leaf labels for blocks in \( \text{ORam}_0 \). Then, for a block with address \( a_0 \) in \( \text{ORam}_0 \), its leaf label is in block \( a_1 = a_0 / \chi \) of \( \text{ORam}_1 \). Accessing data block \( a_0 \) now involves two steps. First, the ORAM controller accesses the PosMap ORAM to get block \( a_1 = a_0 / \chi \), from which the ORAM controller retrieves the leaf label of \( a_0 \) and replaces it with a new random leaf label. Then it loads block \( a_0 \) from the data ORAM. The steps in Figure 3-3 align with Step 1-5 from the basic Path ORAM case in Section 3.2. Of course, the new PosMap might still be too large. In that case, additional PosMap ORAMs (\( \text{ORam}_2, \text{ORam}_3, \cdots, \text{ORam}_{H-1} \)) can be added to further shrink the PosMap.

We observe that the recursive PosMaps here are very similar to multi-level page tables in conventional computers, where the leaf labels are pointers to the next-level page tables or the pages. A recursive ORAM access is similar to a full page table walk. Note that in Figure 3-3, a block in a PosMap ORAM stores leaf labels for several blocks (blocks with address \( a, a+1, \) etc.). These observations motivated the scheme in Chapter 4.

Apparently, the trade-off of recursive ORAM is larger performance overhead. Now the ORAM controller has to access each ORAM in the recursion to serve a request. The total amount of data movement in recursive ORAM is the sum of \( \text{PathLen} \) (defined in Section 3.2.4) for all the ORAMs. We will analyze recursive ORAM performance after deciding on an important parameter—PosMap block size.

### 3.3.1 Block size for PosMap ORAMs

While the block size of data ORAM should match the need of the application (e.g. the cache line size of a secure processor), all the PosMap ORAMs should use a smaller block size, since all we need from a PosMap ORAM on an access is a leaf label of \( L = O(\log N) \) bits, which is typically less than 4 Bytes in practice. This idea led to both asymptotic [54] and empirical [51] improvements over the strawman scheme in the original Path ORAM paper [55] where every ORAM in the recursion uses a uniform block size.

Let us still denote the total number of data blocks as \( N \). Suppose every ORAM in the recursion uses the same block size \( B = \Omega(\log N) \). Then each PosMap block can store \( \chi = B/\log N \) leaf labels. To have a \( O(1) \) size on-chip PosMap, we need \( H = O(\log N / \log \chi) \) levels of recursion\(^2\). The resulting overhead of such a recursive ORAM is then \( \frac{1}{B} H \cdot O(B \log N) = O\left(\frac{\log^2 N}{\log B \cdot \log \log N}\right) \). If \( B \) is polylogarithmic in \( N \) and is at least \( \Omega(\log^2 N) \), then the result is \( O\left(\frac{\log^2 N}{\log \log N}\right) \), breaking even with the best hierarchical ORAM [37].

The better strategy is to reduce the block size of PosMap ORAMs. Let us denote the PosMap ORAM block size as \( B_p \), to distinguish from the data ORAM block size. Let \( B_p = \chi \log N \) where \( \chi \) is a constant (\( \geq 2 \)). Then, the resulting bandwidth overhead becomes

\[
\frac{2}{B} \sum_{h=0}^{H-1} \text{PathLen}_h = \frac{2}{B} \cdot O(B \log N + H \cdot B_p \cdot \log N) = O\left(\frac{\log N + \frac{\log^2 N}{B}}{B}\right) \quad (3.2)
\]

\(^1\)All divisions in this thesis are program division, \( a/\chi \) meaning \( \lfloor a/\chi \rfloor \). We omit the flooring for simplicity.

\(^2\)We remark that since the smaller ORAMs in the recursion have fewer levels, they could have a slightly larger \( \chi \), resulting in a slightly smaller \( H \), but this does not change the asymptotic result.
When the data block size is at least $B = \Omega(\log^2 N)$, recursive Path ORAM achieves $O(\log N)$ memory bandwidth overhead.

We comment that this observation (use small block size for PosMap ORAMs) also improves other position-based ORAMs. Take the trivial bucket version of Shi et al. [31] for example. Its non-recursive construction has $O(\log^2 N)$ memory bandwidth overhead and $O(N)$ PosMap. Choosing $B_p = \chi \log N$ with $\chi$ being a constant again, and carrying out the same analysis, we obtain that its recursive construction has $O\left(\log^2 N + \frac{\log^3 N}{B}\right)$ bandwidth overhead (and $O(1)$ final PosMap of course). With data block size at least $B = \Omega(\log^2 N)$, this bound is $O(\log N)$ better than the original $O(\log^3 N)$ bound.

We remark, however, that in practice the block size for PosMap ORAMs should not be too small for the following two reasons. First, other storage in buckets (the addresses, the leaf labels and the 64-bit counter) will dominate, especially if the buckets have to be padded to a certain size for memory alignment. Moreover, the trade-off of a smaller block size is that as $\chi$ decreases, more PosMap ORAMs are needed into the recursion; adding one more ORAM into recursion introduces one decryption latency and one DRAM row buffer miss latency (the first access to an ORAM is sure to be a row buffer miss).

### 3.4 Position-based ORAM Interface

We have presented the steps of a Path ORAM access $\text{accessORAM}(a, op, d')$ in Section 3.2. Now we are going to split $\text{accessORAM}(a, op, d')$ into two subroutines: $(l, l') = \text{Frontend}(a)$ and $\text{Backend}(a, op, d', l, l')$. $(l, l') = \text{Frontend}(a)$ corresponds to Step 1 to 2 in that section: looking up the PosMap and remapping the requested block. $\text{Backend}(a, op, d', l, l')$ corresponds to Step 3 to 5: reading and writing the path that $\text{Frontend}(a)$ yields. With the frontend-backend separation, the basic (non-recursive) Path ORAM can be characterized by Algorithm 1. In the algorithm, $\mathcal{P}(l)$ is the content on path $l$ where $\mathcal{P}(l)[i]$ is the $i$-th bucket towards the root on $\mathcal{P}(l)$. $S$ is the sequence of buckets to be written back to $\mathcal{P}(l)$ from the stash.

The reason we separate these two steps is to expose the interface of position-based ORAMs, i.e., $\text{Backend}(a, op, d', l, l')$. We define the position-based ORAM interface to be a standard RAM interface $(a, op, d')$ plus a current label $l$ and a new label $l'$ corresponding to the requested block $a$. This interface characterizes all existing position-based ORAMs [31, 39, 43, 53] including Path ORAM. In the Path ORAM case, $l$ and $l'$ are the current and remapped random leaf labels of the address $a$. But we remark that a label $l$ does not have to correspond to a static set of memory locations [39].

A position-based ORAM backend should also achieve correctness and privacy, on the condition that the input current label $l$ matches the (at-the-time) new label $l'$ on the last access to the requested address. If this condition is met, the definitions of correctness and privacy will basically follow those in Section 3.1. Correctness still means that the position-based ORAM backend responds to the input request sequence (in the form of position-based ORAM interface) as a valid RAM with all but negligible probability; and privacy still requires that given two input request sequences of the same length, the two resulting sequences produced by the position-based ORAM backend are computationally indistinguishable.
Algorithm 1 A basic Path ORAM access.

1. **Inputs:** Address \( a \), Operation \( op \), Write Data \( d' \)
2. **function** \( ACCESSORAM(a, op, d') \)
3. \( l, l' \leftarrow \text{Frontend}(a) \)
4. return \( \text{Backend}(a, op, d', l, l') \)
5. **function** \( \text{FRONTEND}(a) \)
6. \( l' \leftarrow \text{PRNG}_K() \mod 2^L \)
7. \( l \leftarrow \text{PosMap}[a] \)
8. \( \text{PosMap}[a] \leftarrow l' \) \( \triangleright \) remap block
9. return \( l, l' \)
10. **function** \( \text{BACKEND}(a, op, d', l, l') \)
11. \( \text{ReadPath}(l) \)
12. \( r \leftarrow \text{FindBlock}(\text{Stash}, a) \) \( \triangleright r \) points to block \( a \)
13. \( (a, l, d) \leftarrow \text{Stash}[r] \)
14. if \( op \neq \text{write} \) then
15. \( \text{Stash}[r] \leftarrow (a, l', d') \)
16. else if \( op \neq \text{read} \) then
17. \( \text{Stash}[r] \leftarrow (a, l', d) \)
18. \( \text{WritePath}(l) \)
19. return \( d \)
20. **function** \( \text{ReadPath}(l) \)
21. for \( i \leftarrow 0 \) to \( L \) do \( \triangleright \) read path
22. \( \text{bucket} \leftarrow \text{Decrypt}_K(\mathcal{P}(l)[i]) \)
23. \( \text{InsertBlocks}(	ext{Stash}, \text{bucket}) \)
24. **function** \( \text{WritePath}(l) \)
25. \( S \leftarrow \text{PushToLeaf}(	ext{Stash}, l) \) \( \triangleright \) see Section 3.2
26. for \( i \leftarrow 0 \) to \( L \) do \( \triangleright \) write path back
27. \( \text{RemoveBlocks}(	ext{Stash}, S[i]) \)
28. \( \mathcal{P}(l)[i] \leftarrow \text{Encrypt}_K(S[i]) \)
3.4.1 Generalizability of the Techniques

The frontend/backend separation also helps better modularize position-based ORAM algorithms. By definition all position-based ORAMs only differ in their backend protocols. Recursion, for example, only involves the frontend and applies to all position-based ORAMs. Unified ORAM and PosMap Lookaside Buffer in Chapter 4, compressed PosMap in Chapter 5 and PosMap MAC integrity verification scheme in Section 7.6 also involve only the frontend, and therefore apply to all position-based ORAMs.

On the other hand, in Chapter 6, we will propose a new position-based ORAM backend called RAW Path ORAM and some optimizations specific to it. The integrity verification scheme in Section 7.4 and 7.5 and the subtree locality memory placement in Section 8.2 are customized for (RAW) Path ORAM and may not work for other constructions.
Chapter 4

PosMap Lookaside Buffer

This chapter presents PosMap Lookaside Buffer, (PLB), a mechanism to reduce the overhead of recursion by exploiting locality in PosMap accesses. First, in Section 4.1, we identify the problem: PosMap ORAM accesses can potentially contribute more than half of the total bandwidth overhead. We then present the main idea of PLB in Section 4.2. Unfortunately, unless care is taken, PLB breaks ORAM access pattern privacy. The solution, Unified ORAM, is presented in Section 4.3 with the detailed algorithm and a discussion on its privacy guarantees. We explore the PLB design space and evaluate its effectiveness in Section 8.3 and 8.4. Unified ORAM and PLB only change the frontend, and can work with any position-based ORAM backend.

4.1 Scalability of Recursive ORAM

In Figure 4-1 we plot the percentage of memory bandwidth overhead due to PosMap ORAMs under realistic parameters. Memory bandwidth overhead is calculated based on Equation (3.1), with all buckets padded to 512 bits to estimate the effect of DRAM bursts. We assume Path ORAM with $U_1 = U_2 = 32$ bits, $\text{CtrWidth} = 64$ bits, $B_p = 256$ bits, $\chi = 8$ and $Z = 4$. The notation $b_{64}.pm8$ in the figure means the ORAM data block size $B$ is 64 Bytes and the final PosMap is at most 8 KBytes. Note that almost all modern processors use a cache line size of 64 Bytes. Abrupt kinks in the graph indicate when another PosMap ORAM is added. Not surprisingly given our previous asymptotic analysis, with a small data block size and a

Figure 4-1: The percentage of data read from PosMap ORAMs.
large ORAM capacity, PosMap ORAMs can contribute to more than half of the memory bandwidth overhead. Allowing a larger final PosMap only slightly dampens the effect.

4.2 PosMap Lookaside Buffer

Given our understanding of recursive PosMap as a multi-level page table for ORAM (Section 3.3), a natural optimization is to cache PosMap blocks so that ORAM accesses exhibiting locality in addresses require less PosMap ORAM accesses. This idea is the essence of the PosMap Lookaside Buffer, or PLB, whose name originates from the Translation Lookaside Buffer (TLB) in conventional systems.

4.2.1 High-Level Ideas

A key observation from Section 3.3 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 cache to the ORAM Frontend called the PLB. Suppose the frontend received a request for block $a_0$ at some point. Recall that the PosMap block $a_i = a_0/\chi^i$ is needed from ORam$_i$ (Section 3.3). If this PosMap block is in the PLB when block $a_0$ is requested, the ORAM controller has the leaf needed to lookup ORam$_{i-1}$, and can skip ORam$_i$ and all the smaller PosMap ORAMs. Otherwise, block $a_i$ is retrieved from 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.

4.2.2 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. Consider two example programs in a system with a recursive ORAM with $H = 2$ where each PosMap block in ORam$_1$ stores leaf labels for $\chi = 4$ blocks in the data ORAM 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 $\chi$ (e.g., touches $a, a+\chi, a+2\chi, \ldots$). 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 ORam$_0$, and 1 denotes an access to ORam$_1$. However, with the PLB, the adversary sees the following access sequences (0 denotes an access to ORam$_0$ on a PLB hit):

Program A : $1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, \ldots$
Program B : $1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \ldots$

Program A hits in the PLB three times out of four accesses while Program B constantly misses in the PLB and needs to access ORam$_1$ on every access. Clearly, this leaks information about the two programs’ access pattern.
4.3 Security Fix: Unified ORAM

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 ORamU. Organizationally, the PLB and on-chip PosMap become the new Path ORAM Frontend, which interacts with a single ORAM Backend (see Section 3.4 for the frontend and backend separation). Security-wise, accesses to data blocks and PosMap blocks are made indistinguishable by ORamU. Both programs from the previous section access only ORamU, and the adversary cannot tell them apart. Section 4.3.4 will give a more detailed discussion on security.

4.3.1 Blocks in ORamU

Data blocks and PosMap blocks (which were originally stored in the PosMap ORAMs) are now stored in the single ORAM ORamU and all accesses are made to this one ORAM. Both data and PosMap blocks now have the same size. PosMap blocks do not take much space since the number of blocks that used to be stored in some ORam_i is \( \chi \) times smaller than the number of blocks stored in ORam_{i-1}.

Each set of PosMap blocks must occupy a disjoint address space so that they can be disambiguated. For this purpose we re-apply the addressing scheme introduced in Section 4.2. Given data block \( a_0 \), the address for the PosMap block originally stored in ORam_i for block \( a_0 \) is given by \( i||a_i \), where \( a_i = a_0/\chi^i \). This address \( i||a_i \) is used to fetch the PosMap block from ORamU 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 \( a_i \).

4.3.2 ORAM readrmv and append Operations

We use two new flavors of ORAM access to support PLB refills/evictions (i.e., op in the RAM request triplet in Section 3.1): read-remove and append. 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. ORamU 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 ORamU must be known so that the block can be written back to the ORAM tree during later ORAM accesses.

The benefit of using readrmv and append is that it saves an ORAM access when a block later needs to be put back to ORAM. Unlike read, readrmv does not leave behind a stale copy of the requested block in the ORAM tree. So when that block is later written back to ORAM, we can directly append it to the stash without invoking a full ORAM backend access.

4.3.3 Detailed Algorithm

The PLB is an conventional cache that stores PosMap blocks. Each PosMap block is also tagged with its block address \( i||a_i \). The steps to read/write a data block with address \( a_0 \) are given below (shown pictorially in Figure 4-2).

1. (PLB lookup) For \( i = 0, \ldots, H - 2 \), look up the PLB for the leaf label of block \( a_i \) (see Section 4.3.1 for the addressing scheme of \( a_i \)). If one access hits, set \( h = i \) and go
to Step 2; else, continue. If no access hits for $i = 0, \ldots, H - 2$, look up the on-chip PosMap at $a_{H - 1}$ and set $h = H - 1$.

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

3. (Data block access) Perform an ordinary \texttt{read} or \texttt{write} to \texttt{ORam$_U$} for block $a_0$.

The detailed pseudo-code is given in Algorithm 2. The PLB supports three operations, \texttt{Lookup Remap} and \texttt{Refill}. \texttt{Lookup}(a) (Line 11) looks for the content associated with address $a$, in our case the leaf label of block $a$. If the leaf exists in the PLB (a ‘hit’), it is returned; otherwise (a ‘miss’) $\perp$ is returned. On a PLB hit, we call \texttt{Remap}(a, l') to change the leaf label of $a$ to $l'$. \texttt{Refill} (Line 20) happens when there is a PLB miss. The missed PosMap block will be read-removed from \texttt{ORam$_U$} and brought into the PLB. This possibly evicts another block from PLB (Line 21), which is appended to \texttt{ORam$_U$}. Thus, each block is stored in the PLB alongside its current label. Aside from adding support for \texttt{readrmv} and \texttt{append}, accessing \texttt{ORam$_U$} requires no other changes to the ORAM Backend.

4.3.4 Unified ORAM Privacy

We now give a proof sketch that our Unified ORAM with PLB achieves the privacy requirement for a position-based ORAM backend in Section 3.4. We use the fact that the PLB interacts with a normal Path ORAM Backend and make the following observations.

\textbf{Observation 1.} If all leaf labels $l_i$ used in \{\texttt{read}, \texttt{write}, \texttt{readrmv}\} calls to \texttt{Backend} are random and independent of other $l_j$ for $i \neq j$, the \texttt{Backend} meets the position-based ORAM backend privacy definition.

\textbf{Observation 2.} If an \texttt{append} is always preceded by a \texttt{readrmv}, stash overflow probability does not increase (since the resulted net change in stash occupancy is the same as that by a \texttt{read} or \texttt{write} operation).
Algorithm 2 Unified ORAM frontend.

1: function AccessORAM(a, op, d')
2:     l, l' ← UORAMFrontend(a)
3:     return Backend(a, op, d', l, l')
4: function UORAMFrontend(a)
5:     for h ← 0 to H − 1 do
6:         l_h' ← PRNG() mod 2^L
7:         if h = H − 1 then
8:             l_h ← PosMap[a_h]  ▶ will always hit
9:             PosMap[a_h] ← l_h'
10:         else
11:             l_h ← PLB.Lookup(a_h)  ▶ may miss
12:             if l_h ≈ ⊥ then
13:                 PLB.Remap(a_h, l_h')  ▶ hit, start access
14:                 break
15:     for i ← h to 1 do
16:         d_i ← Backend(a_i, readrmv, ⊥, l_i, l_i')  ▶ PosMap block accesses
17:         j ← a_i−1 mod χ  ▶ l_{i−1} is j-th leaf in d_i
18:         l_{i−1} ← d_i[j]
19:         d_i[j] ← l_{i−1}'
20:         (a_e, l_e, d_e) ← PLB.Refill(a_i, l_i, d_i)
21:         if a_e ≈ ⊥ then
22:             Backend(a_e, append, d_e, ⊥, l_e)  ▶ PLB eviction
23:     return l_0, l_0'
24: function Backend(a, op, d', l, l')
25:     if op ≈ append then
26:         InsertBlocks(Stash, (a, l', d'))  ▶ append and return
27:     return
28:     ReadPath(l)  ▶ same as in Algorithm 1
29:     r ← FindBlock(Stash, a)
30:     (a, l, d) ← Stash[r]
31:     if op ≈ write then
32:         Stash[r] ← (a, l', d')
33:     else if op ≈ read then
34:         Stash[r] ← (a, l', d)
35:     else if op ≈ readrmv then
36:         Stash[r] ← ⊥  ▶ remove block
37:     WritePath(l)  ▶ same as in Algorithm 1
38:     return d

Theorem 1. The privacy and correctness of Unified ORAM with PLB reduces to the ORAM Backend.

Proof. When the Unified ORAM Frontend requests a block from Backend, the leaf label l sent to Backend comes from either a PosMap block stored in the PLB, or the on-chip PosMap.
In both cases, leaf \( l \) was remapped the instant the requested block was last accessed. We conclude that all \{read, write, readrmv\} commands to Backend are to random/independent leaves and Observation 1 applies. Further, an append commands can only be issued by a PLB eviction, which is the result of a PLB refill, which is further the result of a readrmv operation. Thus, Observation 2 applies.

The PLB, however, will influence the ORAM sequence length \(|\text{ORAM}(x)|\) by, conceptually, filtering out some calls to Backend for PosMap blocks. This is why Unified ORAM with PLB only achieves the position-based ORAM backend access pattern privacy, but not the original ORAM privacy definition (Definition 3 in Section 3.1).

We remark that this level of security is sufficient for secure processors (and maybe some other applications). In a “processor running program” setting, it is most natural to define \( x \) to be the sequence of load/store instructions in the program. Processors have several levels of on-chip cache (usually three for modern processors). When a program requests data, the processor first looks for the data in its Level-1 (L1) cache; on an L1 miss, it accesses the L2 cache, and so on. Only when it misses all of the on-chip cache will the processor issue a request to the external memory. Whether it’s a cache hit or miss depends on the actual access pattern. Thus, given a sequence of load/store instructions of a program, only an access pattern dependent fraction of them will be sent by ORAM. This means \(|\text{ORAM}(x)|\) is now determined by, and thus reveals, the number of Last-Level-Cache (LLC) misses, but not the total number of memory load/store instructions. After adding the PLB, \(|\text{ORAM}(x)|\) is determined by, and thus reveals, the sum of LLC misses and PLB misses. in both cases, the total number of ORAM accesses leaks and is the only thing that leaks. In contrast, the leakage in Section 4.2.2 (insecurely using PLB without Unified ORAM) is the set of PosMap ORAMs needed on every recursive ORAM access, which grows linearly with \(|\text{ORAM}(x)|\).

### 4.4 Comparison with Related Work

Liu et al. [45] proposed allocating different program variables into different ORAMs to reduce each ORAM’s size and therefore improve performance. Given their work, our PLB and Unified ORAM scheme is very counter-intuitive: we show that using a single ORAM tree (instead of multiple potentially smaller ones) can unlock great performance improvement in real-world applications. Note that Liu et al. can only apply their optimization when they see that accesses to multiple ORAM trees do not leak privacy. Otherwise, they still keep the variables in the same ORAM tree. One may argue that our idea to store data and PosMap blocks in the same tree is similar to their not doing anything. To this, we remark that Liu et al. did not mention PosMap as it’s irrelevant, and each ORAM in their work has to be recursive if large enough.

Wang et al. [60] also develop mechanisms to reduce recursive ORAM overhead. Their work only applies when the data stored in ORAM has common data structures, such as a tree of bounded degree or a list. Another two works related to exploiting locality in ORAM are Maas et al. [49] and Yu et al. [61]. The former only works for programs with extremely good locality in large (4 KB) chunks of data. The latter studied prefetching in ORAM and only exploited locality in adjacent blocks. The PLB does not assume any program characteristics and can exploit any irregular locality.
Chapter 5

Compressed PosMap

In this section, we propose a technique to compress the position map of Path ORAM using Pseudo Random Function (PRF). We briefly introduce PRF [3] in Section 5.1, before describing the key ideas of our techniques in Section 5.2. We then parameterize our construction in both a theoretical setting (Section 5.3) and a practical setting (Section 5.4). The theoretical construction achieves an asymptotic improvement over recursive ORAM, as summarized in Table 5.1. In the table, $N$ is again the total number of data blocks and $B$ is the data block size. This is currently the best asymptotic bandwidth overhead under any data block size among ORAMs with at most polylogarithmic client storage. We will evaluate the practical construction in Section 8.4.

<table>
<thead>
<tr>
<th>ORAM scheme</th>
<th>Asymptotic bandwidth overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kushilevitz et al. [37]</td>
<td>$O\left(\frac{\log^2 N}{\log \log N}\right)$ ($B = \Omega(\log N)$)</td>
</tr>
<tr>
<td>Recursive Path ORAM [54]</td>
<td>$O\left(\log N + \frac{\log^2 N}{B}\right)$</td>
</tr>
<tr>
<td>Unified Path ORAM with compressed PosMap</td>
<td>$O\left(\log N + \frac{\log^2 N}{B\log \log N}\right)$</td>
</tr>
</tbody>
</table>

5.1 Preliminary: Pseudo Random Functions

A Pseudo Random Function (PRF) family $y = \text{PRF}_K(x)$ is a collection of efficiently computable functions, where $K$ is a random secret key. It 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 [3].

We could use pseudo random numbers instead of truly random numbers for PosMap. In this case, PosMap stores a monotonically increasing counter $c$ for each block. To generate the leaf label for a block, we input the block’s address $a$ and its access count $c$ into a PRF, i.e., $l = \text{PRF}_K(a||c) \mod 2^L$. This, however, does not do any compression. The counters probably have to be even larger than the original leaf labels of $\log N$ bits. Our construction deals with this issue.
5.2 Construction

Suppose each PosMap block contains $\chi$ leaf labels for the next ORAM. Then, the leaf labels for block \{a, a + 1, \cdots, a + \chi - 1\} will be stored in the same PosMap block. We store in their PosMap block a $\alpha$-bit group counter (GC) and $\chi \beta$-bit individual counters (IC):

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

where $||$ is concatenation. The leaf label of block $a + j$ is defined to be

$$\text{PRF}_K(GC \, || \, IC_j \, || \, a + j) \mod 2^L.$$ 

The output (we call it the uncompressed format) is computationally indistinguishable from a truly random number in $[0, 2^L)$ as long as we never feed the same input to the PRF. Block identifier $a + j$ is included in the input, so different blocks will always use different inputs to the PRF. If we further ensure that the concatenation of the group counter and the individual counter strictly increases, then the PRF always receives new input. 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 overflows (becomes zero again), the ORAM controller will increment the group counter $GC$. This will change the leaf label for all the blocks in the group, so we reset all the individual counters to zero, load all of them from their current paths and remap them to their new paths given by $\text{PRF}_K(GC + 1 \, || \, 0 \, || \, a + j) \mod 2^L$. In the worst case where the program always requests the same block in a group, we need to reset all the $\chi$ individual counters in the group every $2^\beta$ accesses.

This would be very expensive in recursive ORAM. In that case, ORAM controller has to make $\chi$ full recursive ORAM accesses to reset individual counters in a certain ORAM. Otherwise, it reveals that individual counters have overflowed in that particular ORAM, leaking the access pattern. As in the case of PLB, Unified ORAM helps by making an access for resetting an individual counter indistinguishable from a normal access. So the amortized cost to reset individual counters is at most $\chi/2^\beta$ for Unified ORAM.

We note that each block in the ORAM tree or in the stash also has its own leaf label stored beside itself. These leaf labels will still be in the uncompressed format, because they are used in path write-back, where the ORAM controller does not have a chance to reference the PosMap.

5.3 Asymptotic Improvement

Compressed PosMap can lead to an asymptotic bandwidth improvement by reducing the levels of recursion needed. We choose group counter width to be $\alpha = \Theta(\log N)$, the same length as the counters for probabilistic encryption, so group counter overflows can be handled in the same way as in Section 3.2.4. Let individual counter width be $\beta = \log \log N$, and $\chi = \frac{\log N}{\log \log N}$. Then PosMap block size is $B_p = \alpha + \chi \cdot \beta = \Theta(\log N)$. Note that such a compressed PosMap block contains $\chi = O\left(\frac{\log N}{\log \log N}\right)$ leaf labels. Without compression, a PosMap block of size $O(\log N)$ can only store a constant number of leaf labels. These

\footnote{a is a multiple of $\chi$ here so that these $\chi$ blocks share the same PosMap block. The blocks $a, a + 1, \cdots$ can be either data blocks or PosMap blocks, so we omit the subscript $i.$}
parameters make the levels of recursion \( H = O \left( \frac{\log N}{\log \chi} \right) \). This improvement is a factor of 
\( \log \chi = O(\log \log N - \log \log \log N) = O(\log \log N) \).

In Unified ORAM, the overhead of resetting individual counters is \( \chi/2^\beta \) in the worst case: every \( 2^\beta \) accesses to the Unified ORAM tree cause at most 1 individual counter to underflow, which requires \( \chi \) accesses to the Unified ORAM tree. With \( \beta = \log \log N \), and \( \chi = \frac{\log N}{\log \log N} \), this overhead is \( o(1) \). Now we study the asymptotic bandwidth overhead of Unified ORAM with compressed PosMap. In the asymptotic analysis, we assume there is no PLB or PLB never hits, since there is no good way to model locality.

We start by examining a simple case where data block size \( B = B_p = \Theta(\log N) \). In this case we simply use a Unified Path ORAM with block size \( B \). For each ORAM access, we need to make \( H \) accesses to the Unified ORAM, and each time transfer \( O(B \cdot \log N) \) bits. So the asymptotic bandwidth overhead is

\[
\left(1 + \frac{\chi}{2^\beta}\right) \cdot \frac{2}{B} (B \cdot \log N) \cdot H = O \left( \frac{\log^2 N}{\log \log N} \right).
\]

This bound outperforms that of recursive Path ORAM for \( B = \Theta(\log N) \) by a factor of \( O(\log \log N) \), and breaks even with Kushilevitz et al. \[ 37 \] (currently the best ORAM scheme with small client storage under small block size).

For the general case where \( B \neq B_p \), we are faced with the problem of choosing Unified ORAM block size. Unified ORAM should still use block size \( B_p \), because a larger block size for PosMap is sub-optimal as Section 3.3.1 shows. Then, we need to break each data block into sub-blocks of size \( B_p \), and store them in Unified ORAM as independent blocks. We let these sub-blocks share a single individual counter; the uncompressed leaf for each sub-block is obtained by including the sub-block index \( k \) in the PRF input, \( \text{PRF}_K(GC || IC_j || a + j || k) \mod 2^L \). Now a full ORAM access involves \( H \) accesses to Unified ORAM to load the above PosMap block, and another \( \lceil B/B_p \rceil \) accesses to Unified ORAM to load all the sub-blocks of the data block. The asymptotic bandwidth overhead is

\[
\left(1 + \frac{\chi}{2^\beta}\right) \cdot \frac{2}{B} (B_p \cdot \log N) \cdot \left( \left\lceil \frac{B}{B_p} \right\rceil + H \right) = O \left( \log N + \frac{\log^3 N}{B \log \log N} \right).
\]

This bound is also asymptotically better than that of recursive Path ORAM when \( B = o(\log^2 N) \). When we have large data block size \( B = \Omega(\log^2 N) \), recursive and Unified ORAM both achieve \( O(\log N) \) asymptotic bandwidth overhead and outperform Kushilevitz et al. \[ 37 \].

### 5.4 Practical Improvement

Currently, ORAMs in secure processors or other practical applications are usually not large enough to see the asymptotic improvement of our theoretical construction. But compressed PosMap still has benefits in practice. First, it reduces the final on-chip PosMap size. Second, it allows us to cache more leaf labels in PLB and to bring in more leaf labels per PosMap block access. This helps increase PLB hit rate and reduce PosMap accesses.

The choice of these parameters largely depends on the system and applications. We give an example assuming ORAM in processors below. In this scenario, the block size is often 64-Byte to match the cache line size. We can choose \( \alpha = 64 \) (which never overflows), \( \beta = 14, \chi = 32 \) forming a 64-Byte block. With these parameters, we are able to fit 32 leaf
labels in each 64-Byte PosMap block. Effectively, each leaf label is now only 16-bit, and the overhead of resetting individual counters is at most $\chi/2^\beta = 0.2\%$. Originally, each leaf label in the uncompressed format must be at least $\log N$ bits ($\approx 25$).

The compressed PosMap does not lower the security level of Path ORAM or add additional hardware. Even without compressed PosMap, a PRF like AES will be used in practice to generate fresh leaf labels. Compressed PosMap, however, does add an additional AES latency to the critical path. In the PosMap lookup step, the ORAM controller has to perform an AES operation (evaluate the PRF) to get the uncompressed random leaf. This overhead is small compared to the hundreds of cycles of ORAM latency. We will evaluate the practical benefits of compressed PosMap in Section 8.4.
Chapter 6

RAW Path ORAM

6.1 Overview

This chapter proposes a new variant of Path ORAM which we call RAW Path ORAM. RAW Path ORAM is a position-based ORAM algorithm. It can be used with or without recursion and the other frontend related optimizations (PLB and compressed PosMap) in this thesis.

The major benefit of RAW Path ORAM is that it reduces memory bandwidth overhead and encryption overhead by a constant factor. Encryption overhead received relatively less attention in prior ORAM works. Path ORAM requires the same encryption bandwidth as the external memory bandwidth. That is to say, all data read from external memory are decrypted and all data written to external memory are re-encrypted. Yet, in some systems encryption may be even more expensive than moving data to and from external memory, and can potentially become another bottleneck.

We first present the construction of RAW Path ORAM (algorithmic changes from Path ORAM) in Section 6.2. Then we demonstrate RAW Path ORAM’s reduction in bandwidth overhead and encryption in Section 6.3. As a second benefit, we show in Section 6.4 that RAW Path ORAM dramatically simplifies the stash occupancy analysis, which is fairly complicated for Path ORAM. We propose a further optimization in Section 6.5 and discuss RAW Path ORAM’s privacy guarantees in Section 6.6.

6.2 Construction

Parameter $A$. RAW Path ORAM has two types of accesses: read-only (RO) and read-write (RW) accesses. We introduce a new parameter $A$. RAW Path ORAM obeys a strict schedule that it always performs one RW access after every $A$ RO accesses.

An RO access serves the requests to the position-based ORAM backend. It reads the requested block into the stash and removes it from the ORAM tree, but does not evict any blocks from the stash to the ORAM tree. The requested block is then remapped and returned/updated as with basic Path ORAM. This corresponds to Step 3 and 4 in Section 3.2, or Lines 11-17 in Algorithm 1 with two important changes. First, we only decrypt the data of the requested block and add it to the stash, while discarding the rest of the blocks. To do so, we also need to decrypt the address of each block in their (address, leaf, data) triplets. Second, we update/re-encrypt the address part of each block in each bucket along the path. We remove the requested block from the path, by changing its address field to ⊥.

An RW access performs a normal (read+writeback) but dummy ORAM access to a
Algorithm 3 RAW Path ORAM backend.

1: **Initial:** $ROCnt = 0, RWCnt = 0$

2: **function** RAWORAMBackend($a, op, d', l, l'$)

3: \[ ROCnt \leftarrow ROCnt + 1 \] \quad \triangleright \text{RAW counter}

4: if $ROCnt \mod A \neq 0$ then

5: \[ l_g \leftarrow RWCnt \mod 2^L \]

6: RWAccess($l_g$)

7: \[ RWCnt \leftarrow RWCnt + 1 \]

8: return ROAccess($a, l, l', op, d'$)

9: **function** RWAccess($l$)

10: \[ \text{ReadPath}(l) \]

11: \[ \text{WritePath}(l) \] \quad \triangleright \text{read/write path, same as Algorithm 1}

12: **function** ROAccess($a, l, l', op, d'$)

13: for $i \leftarrow 0$ to $L$ do \quad \triangleright \text{read path}

14: \quad \text{bucket} \leftarrow P(l)[i]

15: for $j \leftarrow 0$ to $Z - 1$ do

16: \quad \text{addr} \leftarrow \text{Decrypt}_K(\text{bucket}[j].\text{addr}) \quad \triangleright \text{decrypt address}

17: \quad if $\text{addr} = a$ then

18: \quad \quad \text{block} \leftarrow \text{Decrypt}_K(\text{bucket}[j]) \quad \triangleright \text{decrypt block of interest}

19: \quad \quad \text{InsertBlocks}(\text{Stash}, \text{block})

20: \quad \quad \text{addr} \leftarrow \perp \quad \triangleright \text{remove block of interest from the path}

21: \quad \quad \text{bucket}[j].\text{addr} \leftarrow \text{Encrypt}_K(\text{addr}) \quad \triangleright \text{re-encrypt address}

22: \quad P(l)[i] \leftarrow \text{bucket} \quad \triangleright \text{update address stored externally}

23: Access the block ... Same as line 12-17 in Algorithm 1.

24: return $d$

static sequence of paths corresponding to a *reverse lexicographic ordering*. The only purpose of dummy accesses is to evict blocks from the stash back to the ORAM tree. Hence they only read and write paths without returning or remapping any block.

**Reverse lexicographic ordering.** Gentry et al. [43] proposed writing paths in reverse lexicographic order to simplify the analysis on the stash overflow probability. Let $RWCnt$ be the number of RW accesses made so far. The ORAM controller initializes $RWCnt = 0$ and increments $RWCnt$ by 1 after each RW access. The path for each RW access is then simply the low-order $L$ bits in $RWCnt$, namely $RWCnt \mod 2^L$. Intuitively, reverse lexicographic order minimizes the common sub-paths in consecutive accesses. This improves the eviction quality by load-balancing between paths.

Putting the above ideas together, we have the RAW Path ORAM algorithm in Algorithm 3. The name RAW comes from the fact that the ORAM controller always performs Read-only accesses $A$ times before a read-Write access.

### 6.3 Memory Bandwidth and Encryption Overhead

We compare the memory bandwidth and encryption overhead of RAW Path ORAM and Path ORAM. In both constructions, there are $Z(L + 1)$ blocks on a path. We assume that
in the (address, leaf, data) triplet, the storage of data dominates the other two.

For Path ORAM, all the blocks read in are decrypted and all the blocks written out are re-encrypted. So memory bandwidth and encryption overhead (in terms of blocks) are the same: $2Z(L + 1)$. $Z$ must be large enough to keep the stash overflow probability negligible. As mentioned, $Z = 5$ is proven [54] and $Z = 4$ is experimentally verified for Path ORAM in prior work [49].

For RAW Path ORAM, there are two types of accesses. An RW access is the same as a Path ORAM access. An RO access reads (but does not write) a path and decrypts only one block. The amount of data read vs. decrypted is illustrated in Figure 6-1. So the average memory bandwidth overhead is $\frac{A + 2Z(L + 1)}{A}$; the average encryption overhead is $\frac{2Z(L + 1)}{A}$. Obviously we prefer smaller $Z$ and larger $A$ for RAW Path ORAM, on the condition that stash overflow probability is negligible.

### 6.3.1 Design Space and Parameter Recommendation

Similar to prior work on Path ORAM, we will explore viable $A$ and $Z$ combinations using both experiments and mathematical analysis. For each $Z$, we look for the largest $A$ that makes stash overflow probability decrease exponentially in the stash size. We take the experimental approach in this section, and leave the theoretical analysis to Section 6.4.

Through experiments, we find the following viable combinations: $Z2A1$, $Z3A2$, $Z4A3$ and $Z5A5$ ($Z2A1$ means $Z = 2, A = 1$). Figure 6-2 gives the experimental results and shows that the stash overflow probability drops exponentially with the stash size for these configurations. We experiment with $L = 20$, simulate for over 100 million RO accesses that scan the memory, and sample the stash occupancy before each RW access. The stash size does not include the transient path.

Plugging in parameters from the above analysis, we visualize in Figure 6-3 the bandwidth and encryption overhead of RAW Path ORAM and compare with Path ORAM. It can be seen from the figure that RAW Path ORAM provides a easy way to trade off memory bandwidth and encryption by choosing $Z$ and $A$. Among the viable configurations, $Z3A2$ achieves the best memory bandwidth, a 25% reduction over Path ORAM $Z = 4$, and at the same time a 2.67× reduction in encryption overhead. Points like $Z7A8$ have lower encryption overhead, but worse bandwidth. $Z4A3$ is a good compromise between bandwidth and encryption, achieving 16% reduction in bandwidth overhead and 3× reduction in encryption over Path ORAM $Z = 4$. Moreover, we can rigorously prove that $Z4A3$ has negligible stash overflow probability in Section 6.4; we plot it as a dot to distinguish from others. So for a fair comparison, we should compare $Z4A3$ to Path ORAM with $Z = 5$, also a dot in the figure.
which makes the improvements even larger.

### 6.4 Correctness: Stash Occupancy Analysis

Now we analyze the stash occupancy for a non-recursive RAW Path ORAM. Interestingly, the deterministic write pattern dramatically simplifies the proof, compared to the 8-page proof for Path ORAM [54], which is not as detailed as the proof for RAW Path ORAM here.

Following the notations in Path ORAM [55, 54], by $\text{ORAM}^{Z,A}_L$ we denote a non-recursive RAW Path ORAM with depth $L$ levels and bucket size $Z$, which does one RW access per $A$ RO accesses. The root is at level 0 and the leaves are at level $L$. We define the stash occupancy $\text{st}(S_Z)$ to be the number of real blocks in the stash after a sequence of ORAM accesses (this notation will be further explained later). We will prove that $\Pr[\text{st}(S_Z) > R]$ decreases exponentially in $R$ for certain $Z$ and $A$ combinations.
6.4.1 Proof outline

The proof consists of several steps. The first two steps are similar to Path ORAM [55]. We introduce $\infty$-ORAM, which has an infinite bucket size and after the post-processing algorithm $G$ has exactly the same distribution of blocks over all buckets and the stash. The stash usage of $\infty$-ORAM after post-processing is greater than $R$ if and only if there exists a subtree $T$ in $\infty$-ORAM whose “usage” exceeds its “capacity” by more than $R$. Then we calculate the average usage of subtrees in $\infty$-ORAM. Finally, we apply a Chernoff-like bound on their actual usage to complete the proof.

6.4.2 $\infty$-ORAM

Our proof needs Lemma 1 and Lemma 2 for Path ORAM in Stefanov et al. [55], which we restate in this subsection. We notice that the proof for Lemma 2 there is not very detailed, so we make the necessary changes and give a more rigorous proof here.

We first introduce $\infty$-ORAM, denoted as $\text{ORAM}_{L,A}^{\infty}$. Its buckets have infinite capacity, i.e., an infinite $Z$. It receives the same input request sequence as a RAW Path ORAM. 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 $\text{ORAM}_{L,A}^{\infty}$ as $b_i^\infty$, and $b_i$ of $\text{ORAM}_{L,A}^Z$ 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 $S_\infty$ be state of $\text{ORAM}_{L,A}^{\infty}$ and $S_Z$ be the state of $\text{ORAM}_{L,A}^Z$.

We now propose a greedy post-processing algorithm $G$, which by reassigning blocks in buckets makes each bucket $b_i^\infty$ in $\infty$-ORAM contain the same set of blocks as $b_i^Z$. Formally $G$ takes as input $S_\infty$ and $S_Z$ after the same access sequence with the same randomness. For $i$ from $2^{L+1} – 1$ down to $1$, $G$ processes the blocks in bucket $b_i^\infty$ in the following way:

1. For those blocks that are also in $b_i^Z$, keep them in $b_i^\infty$.

2. For those blocks that are not in $b_i^Z$ but in some ancestors of $b_i^Z$ (the parent of $b_i^\infty$, and note that the division includes flooring). If such blocks exist and the number of blocks remaining in $b_i^\infty$ is less than $Z$, raise an error.

3. If there exists a block in $b_i^\infty$ that’s in neither $b_i^Z$ nor any ancestor of $b_i^Z$, raise an error.

We say $S_\infty$ is post-processed to $S_Z$, denoted by $G_{S_Z}(S_\infty) = S_Z$, if no error occurs during $G$ and $b_i^\infty$ after $G$ contains the same set of blocks as $b_i^Z$ for $i = 0, 1, \ldots, 2^{L+1}$.

**Lemma 1.** $G_{S_Z}(S_\infty) = S_Z$ after the same ORAM access sequence with the same randomness.

To prove this lemma, we made a little change to RAW Path ORAM algorithm. In RAW Path ORAM, an RO access adds the block of interest to the stash and replaces it with a dummy block in the tree. Instead of making the block of interest in the tree dummy, we turn it into a stale block. On an RW access to path $l$, all the stale blocks that are mapped to leaf $l$ are turned into dummy blocks. Stale blocks are treated as real blocks in both $\text{ORAM}_{L,A}^Z$ and $\text{ORAM}_{L,A}^{\infty}$ (including $G_Z$) until they are turned into dummy blocks. Note that this trick of stale blocks is only to make the proof go through. It hurts the stash occupancy and we will not use it in practice. With the stale block trick, we can use induction to prove Lemma 1.

\footnote{Note that the decreasing order ensures that a parent is always processed later than its children.}
Proof. Initially, the lemma obviously holds. Suppose \( G_{S_Z}(S_{\infty}) = S_Z \) after some accesses. We need to show that \( G_{S'_Z}(S_{\infty}) = S'_Z \) where \( S'_Z \) and \( S'_{\infty} \) are the states after the next access (either RO or RW). An RO access adds a block to the stash (the root bucket) for both \( ORAM^Z_A \) and \( ORAM^Z_{\infty, A} \), and does not move any blocks in the tree except turning a real block into a stale block. Since stale blocks are treated as real blocks, \( G_{S'_Z}(S_{\infty}) = S'_Z \) holds.

Now we show the induction holds for an RW access. Let \( RW_i^Z \) be an RW access to \( P(l) \) (path \( l \)) in \( ORAM^Z_{\infty, A} \) and \( RW_i^\infty \) be an RW access to \( P(l) \) in \( ORAM^\infty_{\infty, A} \). Then, \( S'_Z = RW_i^Z(S_Z) \) and \( S'_{\infty} = RW_i^\infty(S_{\infty}) \). Note that \( RW_i^Z \) has the same effect as \( RW_i^\infty \) followed by post-processing, so

\[
S'_Z = RW_i^Z(S_Z) = G_{S'_Z}(RW_i^\infty(S_Z)) = G_{S'_Z}(RW_i^\infty(G_{S_Z}(S_{\infty}))).
\]

The last equation is due to the induction hypothesis.

It remains to show that \( G_{S'_Z}(RW_i^\infty(G_{S_Z}(S_{\infty}))) = G_{S'_Z}(RW_i^\infty(S_{\infty})) \), which is \( G_{S'_Z}(S'_{\infty}) \). To show this, we decompose \( G \) into steps for each bucket, i.e., \( G_{S_Z}(S_{\infty}) = g_1g_2 \cdots g_{2^L+1}(S_{\infty}) \) where \( g_i \) processes bucket \( b_i^\infty \) in reference to \( b_i^Z \). Similarly, we decompose \( G_{S'_Z} \) into \( g'_1g'_2 \cdots g'_{2^L+1} \) where each \( g'_i \) processes bucket \( b'_i^\infty \) of \( S'_{\infty} \) in reference to \( b'_i^Z \) of \( S'_Z \). We now show that for any \( 0 < i < 2^L+1 \), \( G_{S'_Z}(RW_i^\infty(g_1g_2 \cdots g_i(S_{\infty}))) = G_{S'_Z}(RW_i^\infty(g_1g_2 \cdots g_{i-1}(S_{\infty}))) \). This is obvious if we consider the following three cases separately:

1. If \( b_i \in P(l) \), then \( g_i \) before \( RW_i^\infty \) has no effect since \( RW_i^\infty \) moves all blocks on \( P(l) \) into the stash before evicting them to \( P(l) \).
2. If \( b_i \notin P(l) \) and \( b_{i/2} \notin P(l) \) (neither \( b_i \) nor its parent is on Path \( l \)), then \( g_i \) and \( RW_i^\infty \) touch non-overlapping buckets and do not interfere with each other. Hence, their order can be swapped, \( G_{S'_Z}(RW_i^\infty(g_0g_1g_2 \cdots g_i(S_{\infty}))) = G_{S'_Z}g_i(RW_i^\infty(g_0g_1g_2 \cdots g_{i-1}(S_{\infty}))) \). Furthermore, \( b'_i^Z = b'_i^Z \) (since \( RW_i^\infty \) does not change the content of \( b_i \)), so \( g_i \) has the same effect as \( g'_i \) and can be merged into \( G_{S'_Z} \).
3. If \( b_i \notin P(l) \) but \( b_{i/2} \in P(l) \), the blocks moved into \( b_{i/2} \) by \( g_i \) will stay in \( b_{i/2} \) after \( RW_i^\infty \). Since \( b_{i/2} \) is the highest intersection (towards the leaf) that these blocks can go to. So \( g_i \) can be swapped with \( RW_i^\infty \) and can be merged into \( G_{S'_Z} \) as in the second case.

We remind the readers that because we only remove stale blocks that are mapped to \( P(l) \), the first case is the only case where some stale blocks in \( b_i \) may turn into dummy blocks. And the same set of stale blocks are removed from \( ORAM^Z_{\infty, A} \) and \( ORAM^\infty_{\infty, A} \).

This shows \( G_{S'_Z}(RW_i^\infty(G_{S_Z}(S_{\infty}))) = G_{S'_Z}(RW_i^\infty(S_{\infty})) = G_{S'_Z}(S'_{\infty}) \) and completes the proof.

Now we investigate what state \( S_{\infty} \) will lead to the stash usage of more than \( R \) blocks in a post-processed \( \infty \)-ORAM. We say a subtree \( T \) is a rooted subtree, denoted as \( T \in ORAM^\infty_{\infty, A} \) if \( T \) contains the root of \( ORAM^\infty_{\infty, A} \). This means that if a node in \( ORAM^\infty_{\infty, A} \) 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 RAW Path ORAM \( c(T) = n(T) \cdot Z \). Lastly, we define the usage \( X(T) \) to be the actual number of real blocks that are stored in \( T \). The following lemma characterizes the stash size of a post-processed \( \infty \)-ORAM:

\[ \]
Lemma 2. \( \text{st}(G_{S_Z}(S_\infty)) > R \) if and only if \( \exists T \in \text{ORAM}_L^{\infty, A} \) such that \( X(T) > c(T) + R \) before post-processing.

Proof. If part: Suppose \( T \in \text{ORAM}_L^{\infty, A} \) and \( X(T) > c(T) + R \). Observe that \( G \) can assign the blocks in a bucket only to an ancestor bucket. Since \( T \) can store at most \( c(T) \) blocks, more than \( R \) blocks must be assigned to the stash by \( G \).

Only if part: Suppose that \( \text{st}(G_{S_Z}(S_\infty)) > R \). Let \( T \) be the maximal rooted subtree such that all the buckets in \( T \) contain exactly \( Z \) blocks after post-processing \( G \). Suppose \( b \) is a bucket not in \( T \). By the maximality of \( T \), there is an ancestor (not necessarily proper ancestor) bucket \( b' \) of \( b \) that contains less than \( Z \) blocks after post-processing, which implies that no block from \( b \) can go to the stash. Hence, all blocks that are in the stash must have originated from \( T \). Therefore, it follows that \( X(T) > c(T) + R \). \( \square \)

By Lemma 1 and Lemma 2, we have

\[
\Pr[\text{st}(S_Z) > R] = \Pr[\text{st}(G_{S_Z}(S_\infty)) > R] \\
\leq \sum_{T \in \text{ORAM}_L^{\infty, A}} \Pr[X(T) > c(T) + R] \\
< \sum_{n \geq 1} 4^n \max_{T : n(T) = n} \Pr[X(T) > c(T) + R] \tag{6.1}
\]

6.4.3 Average Bucket and Rooted Subtree Load

The following lemma will be used in the next subsection:

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

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

Proof. For a bucket \( b \) in \( \text{ORAM}_L^{\infty, A} \), define \( Y(b) \) to be the number of blocks in \( b \) before post-processing. It suffices to prove that \( \forall b \in \text{ORAM}_L^{\infty, A}, E[Y(b)] \leq A/2 \).

If \( b \) is a leaf bucket, the blocks in it are put there by the last RW access to that leaf. Note that only real blocks could be put in \( b \) on that last access (stale blocks could not\(^2\)), though some of them may have turned into stale blocks. There are at most \( N \) distinct real blocks and each block 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 not a leaf bucket, we define two variables \( m_1 \) and \( m_2 \): the last RW access to \( b \)’s left child is the \( m_1 \)-th RW access, and the last RW access to \( b \)’s right child is the \( m_2 \)-th RW access. Without loss of generality, assume \( m_1 < m_2 \). We then time-stamp the blocks as follows. When a block is accessed and remapped, it gets time stamp \( m^* \), which is the number of RW accesses 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 access to \( b \) (\( m_2 \)-th) has already passed. Therefore, only blocks with time stamp

\(^2\)Stale blocks can never be moved into a leaf by an RW access, because that RW access would remove all the stale blocks mapped to that leaf.
Thus, we can apply a Chernoff-like bound \(^1\) to get an exponentially decreasing bound on the overflow probability. Therefore, \(E[Y(b)] \leq d \cdot 2^{-(i+1)} = A/2\) for any non-leaf bucket as well.  

\[\text{6.4.4 Chernoff Bound}\]

Let \(b\) be the \(m\)-th block in \(b\) may be accessed again after the \(m\)-th access. Notably, \(Z_{4A3}\) is the most competitive configurations. 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 (6.1) we have \(t = \ln(Z/a) > 0\) and that the stash overflow probability decrease exponentially in the stash size \(R\).

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

Trying different \(Z\) and \(A\), we get some configurations that satisfy those two conditions, including \(Z_{3A1}, Z_{4A3}, Z_{5A4}\) and \(Z_{6A5}\). Notably, \(Z_{4A3}\) is the among the most competitive ones so far.

\(^3\)Only real or stale blocks with the right time stamp will be put in \(b\) by the \(m_2\)-th access. Some of them may be accessed again after the \(m_2\)-th access and become stale. But this does not affect the total number of blocks in \(b\) as stale blocks are treated as real blocks.

\(^4\)One way to see this is that a bucket \(b\) at level \(i\) will be written every \(2^i\) RW accesses, and two consecutive RW accesses to \(b\) always travel down the two different children of \(b\).
6.5 Optimization: Non-integer $A$

In this section, we further improve memory bandwidth and encryption of RAW Path ORAM by extending $A$ to non-integers. The idea is to interpret the RW frequency $A$ in a probabilistic sense. For example, if for every access, we make it an RO access with 80% probability and an RW access with 20% probability, it would be equivalent to $A = 4$ in the proof of Lemma 3 and the bound.

Therefore, by adjusting the probability of RO accesses vs. RW accesses, we can effectively make $A$ any real number. Plugging non-integer $A$ into Inequality (6.3), $Z2A0.9$, $Z3A1.9$, $Z5A4.4$ result in negligible the stash overflow probability. These three are very close to the experimentally verified and competitive settings with integer $A$ in Section 6.3.1.

This extension also allows us to use more aggressive parameter settings in practice. We remark that in real implementation, we do not need to make every access probabilistic. Following the notation in Algorithm 3, we simply need to make sure $ROCnt/RWCnt$ is always close to $A$. To achieve this property, the ORAM controller can issue an RW access whenever $ROCnt > RWCnt \cdot A$. This corresponds to a one-line change to Algorithm 3.

With this change, we repeat the experiments in Section 6.3.1. for every $Z$, we look for the maximum $A$ (with 0.1 step) that makes the stash overflow probability decrease exponentially with the stash size. We show the most competitive parameter settings in Figure 6-4. Dots represent the proven ones, and crosses represent the experimentally verified ones. We pick some promising configurations of RAW Path ORAM to compare with Path ORAM. For the proven ones, $Z4A3$ achieves 33% memory bandwidth reduction and 73% encryption reduction over Path ORAM $Z = 5$. For the experimentally verified ones, $Z3A2.4$ achieves 31% memory bandwidth reduction and 69% encryption reduction over Path ORAM $Z = 4$.  

Figure 6-4: Memory bandwidth and encryption overhead of RAW Path ORAM with non-integer $A$. 

43
6.6 Privacy

RAW Path ORAM backend (with integer $A$ or non-integer $A$) satisfies the original ORAM privacy definition (Definition 3) and of course also the position-based ORAM backend privacy definition. RO accesses always read paths in the ORAM tree at random, just like Path ORAM. The total number of RO accesses is equal to the length of the input request sequence. RW accesses occur at predetermined time and are to predictable/data-independent paths.

Even if we consider the theoretical construction where every access is either RO or RW with some probability, the lengths of the two ORAM sequences will have the same distribution. We can still say they have the same length in a probabilistic interpretation.
Chapter 7

Integrity Verification

7.1 Overview

In this chapter, we consider integrity verification for ORAM. We first give the ORAM integrity definitions in Section 7.2: an ORAM integrity verification scheme should guarantee both integrity and access pattern privacy under active (tampering) adversaries. In Section 7.3, we show an attack that breaks recursive ORAM privacy, if no integrity checking is added. We then describe customized optimization to integrity verification for Path ORAM (Section 7.4) and RAW Path ORAM (Section 7.5). Lastly in Section 7.6, we present PosMap MAC, an integrity verification scheme that works for any position-based ORAM and achieves an asymptotic reduction in hashing overhead.

7.2 ORAM Integrity Definition

The definition for integrity itself is straightforward. It requires that even if the external RAM does not behave honestly, ORAM should behave as a valid RAM (defined in Definition 2) or detect the dishonest responses of the external RAM. What complicates the problem is that we want an ORAM to maintain access pattern privacy even if the external untrusted RAM is dishonest, e.g., tampered with by active adversaries. We call it access pattern privacy against active adversaries. In Section 7.3, we show that recursive position-based ORAMs cannot maintain access pattern privacy against active adversaries.

In the presence of active adversaries, ORAM’s output sequence may depend on responses from the external RAM. We write it \( \text{ORAM}(\overleftarrow{x}, \overleftarrow{y}) \), where \( \overleftarrow{y} \) is the response from the external RAM. We can define access pattern privacy against active adversaries in a similar way to Definition 3. To do that, we first define \( \overleftarrow{y}_v \), the maximum prefix of valid responses in \( \overleftarrow{y} \). In other words, the \((|\overleftarrow{y}_v| + 1)\)-th element of \( \overleftarrow{y} \) is the first dishonest response in it.

**Definition 4** (ORAM privacy under active adversaries). An ORAM algorithm achieves access pattern privacy against active adversaries if for any two input sequences \( \overleftarrow{x} \) and \( \overleftarrow{x}' \) of equal length and external RAM responses \( \overleftarrow{y} \) and \( \overleftarrow{y}' \) with equal length of valid prefix (i.e., \( |\overleftarrow{x}| = |\overleftarrow{x}'| \) and \( |\overleftarrow{y}_v| = |\overleftarrow{y}'_v| \)), \( \text{ORAM}(\overleftarrow{x}, \overleftarrow{y}) \) and \( \text{ORAM}(\overleftarrow{x}', \overleftarrow{y}') \) are computationally indistinguishable.

An integrity verification scheme for ORAM has to meet both goals: 1) ensure ORAM to be a valid RAM or detect the tampering under active adversaries, and 2) achieve access
pattern privacy against active adversaries. Apon et al. [57] defined ORAM integrity plus privacy under active adversaries in a more formal way, though the essence is the same as the definitions here.

To meet Definition 4, ORAM usually needs an integrity verification scheme that detects dishonest responses from the external RAM right away, or after a fixed amount of time once they occur. We are also interested in another case with perhaps a slightly relaxed privacy guarantee. We want to allow ORAM to detect dishonest responses after a potentially unpredictable amount of time, as long as it maintains indistinguishability of its output sequence until the detection point. In this case, ORAM leaks some information through the detection time. Unfortunately, this model right now does not fit in a formal framework like Apon et al. [57]. But we find it interesting and promising, because it enables many optimizations including the asymptotically more efficient PMMAC scheme in Section 7.6.

7.3 Recursive ORAM Privacy under Active Adversaries

Before describing our integrity verification scheme, we first point out that integrity is vital for recursive ORAMs to maintain privacy under active adversaries. Specifically, we show that an active adversary can launch an attack for recursive ORAMs that forces the ORAM controller to behave differently under the following two access patterns: (a) repeatedly accessing the same data block, and (b) accessing different data blocks.

We use a 2-level recursive Path ORAM \{ORam_0, ORam_1\} as an example, but the attack works for any other recursive position-based ORAMs. We introduce the following notations. Let \(l_1\) and \(l_0\) be the paths in ORam_1 and ORam_0 read and written by the ORAM controller on the current access, and \(l^*_1\) and \(l^*_0\) be the paths in ORam_1 and ORam_0 read and written on the last access. The adversary always keeps a copy of \(P(l^*_1)\), the data seen by the ORAM controller on the previous access to ORam_1. On every recursive ORAM access, the adversary does the following:

1. If \(l_1 = l^*_1\), returns \(P(l^*_1)\) to the ORAM controller (old data on that path) and goes to the next step. Otherwise, saves \(P(l_1)\) and repeats this step on the next access to ORam_1.

2. The ORAM controller then accesses path \(l_0\) in ORam_0. If \(l_0 = l^*_0\), guess access pattern (a); otherwise, guess access pattern (b).

Let \(L_i\) be the height of the ORam_i tree \((i = 0, 1)\). Given a sequence of random leaf labels, for any two consecutive labels, \(l_1 = l^*_1\) happens with \(2^{-L_1}\) probability. If the sequence is long (on the scale of \(2^{L_1}\)), then with high probability the adversary can find \(l_1 = l^*_1\) somewhere in the sequence. Then, the ORAM controller gets \(P(l^*_1)\), the snapshot of path \(l_1\) before the last path write-back operation. In this case, if the same block is accessed, the ORAM controller must get the same leaf label \(l_0 = l^*_0\) from the snapshot. However, if a different block is accessed, \(l_0 \neq l^*_0\) with all but \(2^{-L_0}\) probability. Therefore, the adversary distinguishes memory access pattern (a) from (b) with non-negligible (actually quite high) probability.

Unified ORAM is also vulnerable to this attack: though there is only one ORAM tree, it logically contains the contents of multiple levels of PosMaps. Position-based ORAM without recursion is not vulnerable to this attack because its PosMap is securely stored within the ORAM controller.
We also learn from this attack that integrity verification cannot be done in the background for recursive or Unified ORAM. In other words, the ORAM controller cannot speculatively access ORam\(_i\) before verifying the leaf label it gets from ORam\(_{i+1}\). Security breaks when the ORAM controller exposes the replayed label (\(l_0\) in the attack).

7.4 For Path ORAM

7.4.1 Basic Path ORAM

A na"ive solution is to store a Merkle tree [2] in external memory, where each bucket in the ORAM tree is a leaf node of the Merkle tree. This scheme would work with any kind of ORAM, and is used in Shroud [46], but it has large overheads. In Path ORAM for example, all the \((L + 1)\) buckets on a path have to be verified through the Merkle tree on each ORAM access. This results in \(O(L^2)\) complexity, (one \(O(L)\) factor comes from Path ORAM and the other from Merkle tree). To reduce the integrity verification overhead, we exploit the fact that the basic operation of both the Merkle tree and Path ORAM is reading/writing paths through their tree structures.

We create a hash tree that has exactly the same structure as Path ORAM (shown mirrored in Figure 7-1). Let \(B_i\) be the \(i\)-th bucket in the ORAM tree, and \(h_i\) be the corresponding node in the hash tree. At a high level, each hash in the hash tree depends on its two children as well as the corresponding bucket in the Path ORAM tree, i.e., \(h_i = \text{hash}(B_i || h_{2i+1} || h_{2i+2})\). \((B_{2i+1} \text{ and } B_{2i+2} \text{ are the two children of bucket } B_i)\). The root hash \(h_0\) is stored inside the ORAM controller and cannot be modified by an adversary. Similar to a Merkle tree, the security of the proposed scheme can be reduced to the collision resistance of the hash function. Assuming the hash function is collision resistant, this scheme provides ORAM integrity and access pattern privacy under active adversaries. Note that this scheme detects any dishonest response from the external RAM right away.

Now we describe the steps of an ORAM access with this integrity verification scheme. Following Figure 7-1, we give an example access to a block mapped to path \(l = 2\).

1. ORAM path read: read buckets \(B_0\), \(B_2\) and \(B_5\) into the stash.
2. Read the hashes on path \(l = 2\) and the sibling hashes, i.e., \(h_1, h_2, h_5\) and \(h_6\).
3. If \(h_5 = \text{hash}(B_5)\) and for \(i = \{2, 0\}\), \(h_i = \text{hash}(B_i || h_{2i+1} || h_{2i+2})\), the path is authentic and fresh!
4. ORAM path writeback: write back bucket \(B_0\), \(B_2\), \(B_5\).
5. Re-compute \(h_5\), \(h_2\) and \(h_0\); write back \(h_5\) and \(h_2\).

All data touched in external memory is shaded in Figure 7-1.

The asymptotic complexity of this integrity verification scheme is \(O(L) = O(\log N)\). On each ORAM access \(L\) hashes (the sibling hashes) need to be read into the ORAM controller, and \(L\) hashes (along the path) need to be checked and then updated in the external hash tree. For practical parameterization and implementation considerations, readers can refer to our initial work on this scheme [50].
7.4.2 Recursive Path ORAM

To integrity-verify recursive Path ORAM, we can of course apply the above scheme in Section 7.4.1 to every Path ORAM in the recursion. But can we do better? As noted, verifying PosMap ORAMs is on the critical path. The latency of evaluating the hash function depends on the length of the input and is likely to become a bottleneck. In this section, we show that the input to the hash can be significantly reduced for PosMap ORAMs, if instead of using Definition 4 we allow the ORAM controller to not detect some dishonest responses from server storage as we mentioned in Section 7.2.

We first generalize the probabilistic encryption scheme in Section 3.2.1.3 to pseudorandom functions (PRF),

$$\text{encrypt}_K(X) = (s, \text{PRF}_K(s) \oplus X)$$

where $$\{\text{PRF}_K : \{0,1\}^{|s|} \rightarrow \{0,1\}^{|X|}\}$$ is a family of pseudorandom functions indexed by key $$K$$. To encrypt a message $$X$$, the encryption module feeds a seed $$s$$ to $$\text{PRF}_K$$, and then XORs the output (as a one-time pad) to $$X$$. The pseudorandomness of $$\text{PRF}_K$$ guarantees that $$\text{PRF}_K(s)$$ is computationally indistinguishable from truly random strings. It is required that the ORAM controller does not reuse the same seed (if the same seed is used in both $$\text{PRF}_K(s) \oplus X_1, \text{PRF}_K(s) \oplus X_2$$, then $$X_1 \oplus X_2$$ would be revealed to the adversary). This is guaranteed by the selection of seed in Section 3.2.1.3.

**Theorem 2.** To achieve integrity and access pattern privacy up to the detection point, it suffices to integrity-verify data ORAM and the seeds for PosMap ORAMs.

**Proof outline.** The proof of the theorem is based on two insights. First, data ORAM stores (address, data, leaf) triplets. If the ORAM controller gets the wrong leaf label from corrupted PosMap ORAMs, it can detect that in data ORAM. Second, since the seeds $$s$$ are integrity-verified, an adversary cannot steer the ORAM controller to re-using the
same seed with the same key. If the seeds in PosMap ORAMs are fresh and authentic, any ciphertext produced by an adversary will decrypt into random bits. In that case, the ORAM controller will behave indistinguishably—always accessing a random path in each ORAM—and the adversary cannot learn anything about the access pattern. We will prove these two insights as lemmas.

Consider a recursive Path ORAM with \( H \) levels \( (\text{ORam})_{i=0}^{H-1} \), where \( \text{ORam}_0 \) is the data ORAM. From an observer’s view, the ORAM controller exposes a leaf label (accesses a path) for each ORAM in the recursion. Let \( l_j^i \) be the path read and written for \( \text{ORam}_i \) on the \( j \)-th ORAM access, where \( i \in \{0, 1, \cdots, H-1\}, j \in \{0, 1, \cdots, m\} \). We can regard all the PosMap ORAMs combined as \( \text{PosMap} \), which returns a sequence of leaf labels \( (l_j^0)_{j=1}^m \) for accessing the data ORAM. For convenience, we denote \( (l_j^0)_{j=1}^m \) as the correct leaf labels returned by the original (unmodified) PosMap ORAMs, and \( (l_j^{0'})_{j=1}^m \) as the leaf labels returned by the modified PosMap ORAMs.

**Lemma 4.** Given \( \text{ORam}_0 \) is authentic and fresh, if \( \exists j \) where \( l_j^{0'} \neq l_j^0 \), then the ORAM controller will detect this when accessing \( \text{ORam}_0 \).

**Proof.** We show that the ORAM controller can detect the first time (the smallest \( j \)) that \( l_j^{0'} \neq l_j^0 \), which is sufficient to prove the lemma. Since data ORAM is authentic and fresh (it is integrity-verified using the method in Section 7.4.1), the Path ORAM invariant guarantees that a triplet \( (b_j, u_j, l_j^0) \) is stored somewhere along path \( l_j^0 \) in the \( \text{ORam}_0 \) tree or in the stash. If due to the wrong output of \( \text{PosMap} \), the ORAM controller accesses path \( l_j^{0'} \) in \( \text{ORam}_0 \), then either:

1. block \( b_j \) is not found along path \( l_j^{0'} \) or the stash; or

2. block \( b_j \) is found in the stash or on the common subpath of path \( l_j^{0'} \) and path \( l_j^0 \), but the ORAM controller compares \( l_j^{0'} \) with the leaf label stored in the triplet and finds \( l_j^{0'} \neq l_j^0 \).

In either case, the ORAM controller detects that the PosMap ORAMs have been modified.  

**Lemma 5.** Given the seeds are authentic and fresh, whichever way an adversary tampers with any \( \text{ORam}_i \), \( 1 \leq i \leq H-1 \), \( l_j^i \) is computationally indistinguishable from uniformly random for any \( i, j \) to the adversary.

**Proof.** First note that \( l_j^{H-1'} \), the path read/written in the smallest ORAM, is stored securely inside the ORAM controller, so it is uniformly random.

Leaf labels for the other ORAMs, \( l_j^i \) (\( i < H-1 \)), are produced during the execution of \( \text{PosMap} \). Suppose the original leaf label \( l_j^0 \) is part of a block \( X \) in \( \text{ORam}_{i+1} \). By \([X] \) we denote the restriction of \( X \) to the bits that represent the leaf labels for \( \text{ORam}_i \); this excludes the address and leaf for \( X \) itself in the triplet \( X \). At any time, \([X] \) is uniformly random because it always contains fresh leaf labels. (Recall that whenever a leaf label is used in ORAM operations, it will be replaced with a new random one). So the same restriction of the ciphertext \([Y] = [\text{PRF}_K(s) \oplus X] = [\text{PRF}_K(s)] \oplus [X] \) is also uniformly random, and statistically independent from \([\text{PRF}_K(s)] \). An adversary can change \( Y \) to \( Y' \). Since \( Y' \) only depends on \( Y \), \( [Y'] \) and \([\text{PRF}_K(s)] \) are also statistically independent. The seed \( s \) is authentic.

---

\(^1\)In practice, \([X] \) may be generated by some pseudorandom number generator. Then statistical independence in the proof becomes computational independence, and the lemma still holds.
and fresh, so the ORAM controller decrypts $Y'$ into $X' = \text{PRF}_K(s) \oplus Y'$. Now we have $\text{PRF}_K(s)$ that is computationally indistinguishable from uniformly random strings and is statistically independent from $[Y']$, so the resulting $[X']$ and $l_1^u$ (one of the leaf labels in $[X']$) are also indistinguishable from uniformly random strings. 

Lemma 4 states that the ORAM controller can detect any wrong leaf label it gets from PosMap when it accesses the authentic and fresh data ORAM. Note that Lemma 4 does not preclude the possibility that PosMap, though modified, still gives the correct leaf label $l_0^0$. This is allowed since the ORAM controller still gets the authentic and fresh data from the data ORAM in this case.

Lemma 5 states that as long as the seeds are verified, the paths accessed for each ORAM in the recursion by the ORAM controller will be computationally indistinguishable from uniformly random, regardless of how an adversary tampers with the rest of the contents in PosMap ORAMs. In the proof of Lemma 5, it is important that the seeds are verified. If $s$ can be tampered with, then $X'$ may not be $\text{PRF}_K(s) \oplus Y'$. The attack in Section 7.3 takes advantage of this security hole.

Combining the two lemmas completes the proof of Theorem 2.

### 7.5 For RAW Path ORAM

The deterministic nature of RW accesses in RAW Path ORAM allows us to use Message Authentication Codes (MAC) instead of Merkle trees to verify part of the data. We first give some background on MAC. Then, we present our construction and discuss its security.

#### 7.5.1 Preliminary: Message Authentication Codes

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 $d_i$ ($i = 0, \ldots$) can be modified by some adversary Eve. To guarantee message authenticity, Alice can send Bob tuples $(h_i, d_i)$ where $h_i = \text{MAC}_K(d_i)$ and $\text{MAC}_K()$ is a Message Authentication Code with secret key $K$, e.g., a keyed hash function [6]. The MAC scheme guarantees that Eve can only produce a message forgery $(h^*, d^*)$ with negligible probability, where $h^* = \text{MAC}_K(d^*)$ and $d^*$ 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 well-known fix for this problem is to embed a monotonic counter in each MAC [18], i.e., for message $i$, Alice transmits $(h'_i, d_i)$ where $h'_i = \text{MAC}_K(i||d_i)$. Eve can no longer replay an old packet $(h'_j, d_j)$ because $\text{MAC}_K(i||d_i) \neq \text{MAC}_K(j||d_j)$ with overwhelming probability.

#### 7.5.2 Construction

We first point out that in RAW Path ORAM, we can easily determine at any time how many times any bucket along any path has been written. Specifically, due to the static and load-balancing nature of the reverse lexicographic order, if $\mathcal{P}(l)[i]$ has been written $g_i$ times, then $\mathcal{P}(l)[i + 1]$ has been written $g_{i+1} = \lfloor (g_i + 1 - l_i)/2 \rfloor$ times. where $l_i$ is the $i$-th bit in $l$, the path in question. In Algorithm 3, the ORAM controller already keeps track of the total number of RW accesses $\text{RWCnt}$. At any time, the root bucket, or $\mathcal{P}(l)[0]$, has been written
\( g_0 = RWCnt \) times. Then, we can use the above load-balance relation to compute the write count of bucket \( P(l)[i] \) for any \( l \) and \( i \).

With the write count of every bucket known at any time, we can simply use the above replay-resistant MAC scheme with the write count serving as the monotonic counter. For a certain bucket, let \( d \) be the plaintext data of the bucket, including leaf labels and payloads for all the \( Z \) blocks, but not the block addresses. Let \( b \) be the bucket’s unique index (e.g., physical address), and \( g \) be the write count of the bucket. We compute the following MAC and store it alongside the bucket:

\[
h = MAC_K(g || b || d).
\]

When a bucket is read, we do the same computation and compare with the stored MAC.

However, RAW Path ORAM still updates block addresses non-deterministically (RO accesses are to random paths). So we still need a mirrored Merkle tree like the one Section 7.4.1 to verify the block addresses.

On an RW access, we check whether the MAC matches for each bucket read from the ORAM tree, and also generate the MAC for each bucket to be written out to the ORAM tree. The Merkle tree for the block addresses are also checked and updated accordingly. On an RO access, only block addresses are updated (a certain block’s address in the bucket will be set to \( \perp \)), and all the leaf labels and data are not changed. So we only need to update the Merkle tree for the block addresses, while all the MACs stay the same. On an RO access, we can either check the MAC for every bucket, or only the bucket that contains the requested block. These two strategies will have different security guarantees, which we further discuss in Section 7.5.3.

\textbf{Comparison to Path ORAM.} If we only check the bucket that contains the requested block, for every \( A \) RO accesses and one RW accesses, we need to check and rehash \( A + L + 1 \) buckets. This leads to a reduction in hashing bandwidth over the Merkle tree based scheme for Path ORAM in Section 7.4.1. In that scheme, each access must check and rehash all the \( L + 1 \) buckets on the path. If block size \( B \) outweighs other metadata, which is \( O(\log N) \) bits, then the reduction is about a factor of \( A \), similar to the reduction in encryption.

\section*{7.5.3 Security}

This scheme provides integrity. The block addresses are protected by the mirrored Merkle tree scheme in Section 7.4.1. Now we only focus on our change: the MAC for leaf labels and data in each bucket. An adversary cannot perform a replay attack because the monotonic write count \( g \) is included in the MAC, and can be correctly computed by the ORAM controller. Coming up with a forgery is as hard as breaking the underlying MAC. Note again that only the data \( d \) of a bucket come from the ORAM tree. The index \( b \) and the write count \( g \) of the bucket are computed inside the ORAM controller on each access. To come up with a forgery, the adversary has to find a pair \((h', d')\) such that \( h' = MAC_K(g || b || d') \).

If the adversary succeeds in doing so, it has broken the underlying MAC scheme.

On the privacy side, it depends on whether we check every bucket, or only the bucket that contains the requested block on RO accesses. If we do the former, the scheme detects any dishonest responses from the external RAM right away, and meets Definition 4. If we do the latter, however, then the ORAM controller may not detect all dishonest responses and may leak some information through the detection time.
7.6 For All Position-based ORAMs: PosMap MAC

We now describe a novel and simple integrity verification scheme that is general to any position-based ORAM. We call it PosMap MAC, or PMMAC for short, since it takes advantage of PosMap, the fundamental structure of position-based ORAMs. PMMAC achieves asymptotic improvements in hash bandwidth and removes a serialization bottleneck of the Merkle tree based schemes in previous sections. PMMAC has two downsides. First, it slightly increases server storage: PMMAC needs to store one MAC digest in server storage per block, instead of per bucket. Second, it allows a small amount of leakage through detection time. But due to its simplicity and huge reduction in hashing, we strongly prefer PMMAC in practice.

7.6.1 Construction

The key idea of PMMAC is that a pseudorandom PosMap can double as a table that tracks each block’s access count. Recall from the compressed PosMap (Section 5.2) that we use the concatenation of the group counter and the individual counter as a monotonic counter, and feed it to a PRF to get the pseudorandom leaf label. We can use this concatenation again as the monotonic counter to implement the replay-resistant MAC scheme in Section 7.5.1.

Consider some block $a$ which has data $d$ and access count $c = GC || IC$. We can produce the following MAC and store it alongside block $a$,

$$h = MAC_K(c || a || d)$$

When block $a$ is read, we check the authenticity/freshness by

assert $h \ast == MAC_K(c || a || d \ast)$

where $\ast$ denotes values that may have been tampered with. Crucially, the access count $c$ cannot be tampered with. The final PosMap stored in client storage forms the root of trust, then recursively, the verified PosMap blocks become the root of trust for the next level PosMap or data blocks.

PMMAC only needs to integrity verify (check and update) 1 block—namely the block of interest—per access, achieving an asymptotic reduction in hash bandwidth over Merkle tree schemes. This also eliminates the serialization bottleneck in Merkle tree schemes: each hash in the Merkle tree node depends on its child hashes, and hence the scheme is fundamentally sequential.

7.6.2 Adding Encryption: Subtle Attacks and Defences

In Section 3.2.1.3, we assumed the probabilistic encryption scheme is AES counter mode, and say that the counter can be either per bucket or global. Now we show that under active adversaries, per bucket counters no longer work because the adversary is able to replay the counter, and PMMAC does not fix the issue.

For a bucket currently encrypted with the pad $p = AES_K(BID || |Ctr||i)$, the adversary can replace the plaintext bucket counter with $Ctr - 1$. This modification may not trigger an integrity violation under PMMAC if this bucket does not contain the block of interest for the current access. If an integrity violation is not triggered, then this bucket will be encrypted using the same one-time pad $p$ again. This obviously leaks information: if the
same pad \( p \) is used to encrypt plaintext data \( d \) at some point and \( d' \) at another point, the adversary learns \( d \oplus d' \).

Simply using a global counter or a random seed for each bucket fixes the problem.

### 7.6.3 Security

#### 7.6.3.1 Integrity

We show that breaking PMMAC as hard as breaking the underlying MAC, and thus PMMAC attains ORAM integrity. First, we have the following observation:

**Observation 3.** 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. Assuming no tampering has happened, every block \( a \) (data or PosMap) is associated with a dedicated counter that increments on each access. Thus, each address and counter pair will be different from previous ones. We now use Observation 3 to prove the following theorem.

**Theorem 3.** 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 \(h_{1} = MAC_{K}(c_{1}||a_{1}||d_{1})\). \((a_{1}, c_{1})\) 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 \(h'_{1} = MAC_{K}(c_{1}||a_{1}||d'_{1})\) is as hard as breaking the underlying MAC. Suppose no integrity violation has happened and Theorem 3 holds up to access \(n-1\). This means the Frontend sees fresh and authentic \((a_{i}, c_{i})\)'s for \(1 \leq i \leq n-1\). By Observation 3, \((a_{n}, c_{n})\) 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 (§ 7.5.1) because all \((a_{i}, c_{i})\)'s are distinct from each other and are tamper-proof. Being able to produce a forgery \((h'_{i}, d'_{i})\) where \(d'_{i} \neq d_{i}\) and \(h'_{i} = MAC_{K}(c_{i}||a_{i}||d'_{i})\) means the adversary can break the underlying MAC.

### 7.6.3.2 Privacy

The privacy guarantee under active adversaries requires certain assumptions with PMMAC, since PMMAC only checks the block of interest. Take (RAW) Path ORAM as an example, some tampered data may be written to the stash and later be written back to the ORAM tree. The adversary can also tamper with the block addresses, so the ORAM controller won't be able find the block of interest (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 require a correct implementation of the position-based ORAM Backend to have the following property. If the Backend makes an ORAM access, it only reveals to the adversary (a) the (leaf) label sent by the Frontend for that access and (b) a fixed amount of encrypted data to be written to server storage.

If the property holds, it is straightforward to see that any memory request address trace generated by the Backend is computationally indistinguishable from other traces of the same length. That is, the Frontend receives tamper-proof responses (by Theorem 3) and therefore
produces independent and random leaves. The global seed scheme in Section 7.6.2 trivially guarantees that the data written back to memory gets a fresh pad. Admittedly even if the property holds, the detection time of some tampering still depends on many factors and is unclear, so PMMAC does not meet Definition 4. But a PMMAC verified ORAM only leaks the detection time, and maintains access pattern privacy up to that point.
Chapter 8

Evaluation

8.1 Methodology and Parameters

We evaluate our proposals using the Graphite simulator \cite{24} with the parameters in Table 8.1. We use a representative subset of SPEC06-int benchmarks \cite{17} with reference inputs only. All workloads are warmed up over 1 billion instructions and then run for 3 billion instructions.

We evaluate Unified ORAM Frontend with Path ORAM Backend. We derive AES/Hash latency, Frontend and Backend latency directly from a hardware prototype we build. Frontend latency is the time to evict and refill a block from the PLB (Section 4.3.3) and occurs at most once per Backend call. Backend latency (approximately) accounts for the cycles lost due to hardware effects such as buffer latency. Frontend latency and Backend latency are both added on top the time it takes to read/write an ORAM tree path in DRAM. ORAM latency of reading/writing paths is sensitive to DRAM model, and more importantly, how we map buckets to physical addresses. We explore this issue in Section 8.2.

8.2 Path ORAM on DRAM

Most works on ORAM only focused on ORAM’s ideal memory bandwidth overhead, i.e., the amount of extra data moved per access or other similar metrics. We now study how to efficiently implement ORAM on commodity DRAM, in order for it to be used in processors. If not properly done, Path ORAM can incur significantly larger overhead than the ideal results when actually built on DRAM. For example, DRAM latency is much higher on a row buffer miss than on a row buffer hit. 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. We propose an efficient memory placement strategy to improve Path ORAM’s performance on DRAM. The technique applies to RAW Path ORAM as well.

We pack each subtree with $k$ levels together, and treat them as the nodes of a new tree, a $2^k$-ary tree with $\left\lceil \frac{L+1}{k} \right\rceil$ levels. Figure 8-1 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, which together with the original bucket size determines $k$. In other words, we make each subtree fit in a DRAM row. With this memory layout, while we are traversing a path, within each subtree we are guaranteed to hit in the row buffers. Row buffer misses can (and will) occur
only when we transition to the next subtree.

We use DRAMSim2 [30] to simulate ORAM performance on commodity DRAM. We simulate the DDR3_micron model with 8 banks, 16384 rows and 1024 columns per row at 667 MHz. Figure 8-2 presents the time (in nano seconds) to read a path from a Path ORAM with the configurations in Table 8.1. We evaluate the naïve memory placement and our subtree strategy, and compare with the ideal value. The access time is averaged over multiple ORAM accesses. The naïve scheme is more than 2\times worse than the ideal result. The subtree memory placement strategy is only 6% worse than the ideal value with one memory channel, and 19% worse with two. The performance loss comes from the few row buffer misses and DRAM refresh. ORAM latency decreases with channel count as expected but the effect becomes increasingly sub-linear for larger channel counts due to DRAM channel

<table>
<thead>
<tr>
<th>Table 8.1: Experimental setup.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core, on-chip cache and DRAM</td>
</tr>
<tr>
<td>core model</td>
</tr>
<tr>
<td>add/sub/mul/div</td>
</tr>
<tr>
<td>fadd/fsub/fmul/fdiv</td>
</tr>
<tr>
<td>L1 I/D cache</td>
</tr>
<tr>
<td>L1 data + tag access time</td>
</tr>
<tr>
<td>L2 Cache</td>
</tr>
<tr>
<td>L2 data + tag access time</td>
</tr>
<tr>
<td>cache line size</td>
</tr>
<tr>
<td>Path ORAM/ORAM controller</td>
</tr>
<tr>
<td>data block size</td>
</tr>
<tr>
<td>data ORAM capacity</td>
</tr>
<tr>
<td>Z</td>
</tr>
<tr>
<td>AES/SHA-3 latency</td>
</tr>
<tr>
<td>Frontend/Backend latency</td>
</tr>
<tr>
<td>DRAM</td>
</tr>
<tr>
<td>DRAM bandwidth</td>
</tr>
<tr>
<td>DRAM latency</td>
</tr>
</tbody>
</table>

![Figure 8-1: Illustration of subtree locality.](image-url)
8.3 PLB Design Space

Figure 8-3 shows how direct-mapped PLB capacity impacts performance. For a majority of benchmarks, larger PLBs add small benefits (≤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 (not shown for space) and found that, with a fixed PLB capacity, a fully associative PLB improves performance by ≤10% when compared to direct-mapped. To keep the architecture simple, we therefore assume direct-mapped PLB from now on. Going from a 64 KB to 128 KB direct-mapped PLB, average performance only increases by only 2.7%, so we assume a 64 KB direct-mapped PLB for the rest of the evaluation.

8.4 Final Results

We now present the performance of PLB, compressed PosMap and PMMAC. To name our schemes in the discussion, we use the letters P, I and C to indicate the PLB, Integrity verification (PMMC) and Compressed PosMap, respectively. For example, PC_X32 denotes PLB+Compressed PosMap with χ = 32. For PC_X32 and PIC_X32, we apply recursion three times to get a 4 KB final PosMap. R_X8 is a recursive ORAM with χ = 8 (32-Byte PosMap ORAM blocks, the best found in [51]) 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.

To give more insight, Figure 8-5 shows the average data movement per ORAM access. 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 rapidly 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 χ. Compressed PosMap (PIC_X32) solves this problem.

Lastly, we comment that the empirical results in Table 1.1 of Section 1.1 come from this experiment with 4 GB capacities. Note that an ORAM access returns a 64-Byte block (cache line) to the processor. Recursive Path ORAM R_X8 moves 14.74 KB data to fulfill this access, so we say its empirical memory bandwidth overhead is 14.74 KB / 64 B = 235.8×. Unified Path ORAM with PLB and compressed PosMap are calculated in the same way. For RAW Path ORAM, we assume the Z3.42.4 setting, which has a data-independent 17% improvement of memory bandwidth over Path ORAM with Z = 4, giving the 121× result. This improvement is smaller than what’s reported in Figure 6-4 and Section 6.5 because we consider metadata here.

Figure 8-4: Performance of PLB, Compressed PosMap and PMMAC. Slowdown is relative to a conventional system with DRAM.

Figure 8-5: Scalability to large ORAM capacities. White shaded regions indicate data movement from PosMap ORAMs. Slowdown is relative to a conventional system with DRAM.
Chapter 9

Conclusion

We have defined the position-based ORAM interface and divided position-based ORAMs into frontend and backend so that we can optimize them separately. We have presented the following three improvements to position-based ORAM frontend, which work with any position-based ORAM: (1) Unified ORAM with the PosMap Lookaside Buffer (PLB) to securely exploit program locality and decrease the performance overhead of recursion, (2) compressed PosMap to improve both asymptotic and practical memory bandwidth, and (3) PosMap MAC to get integrity verification with an asymptotic reduction in hashing. We have also proposed RAW Path ORAM, a new position-based ORAM backend. It reduces both memory bandwidth overhead and encryption overhead by a constant factor, and dramatically simplifies the proof for correctness.

With the asymptotic improvement in this thesis, Unified (RAW) Path ORAM has the best memory bandwidth among ORAM constructions with constant or polylogarithmic client storage under any block size. For the practical improvements, simulation results show that our PLB and PosMap compression techniques combined, reduce PosMap (overall) ORAM bandwidth overhead by 82% (38%), which leads to a 1.43× speedup for SPEC workloads. RAW Path ORAM can potentially contribute about another 30% reduction in memory bandwidth overhead.
Bibliography


[59] Christopher W. 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 High Performance Computer