AN ABSTRACT OF THE THESIS OF

Hadi Rahal-Arabi for the degree of Master of Science in Computer Science presented on June 17, 2021.

Title: The Symbioses of Oblivious Random Access Memory and Trusted Execution Environments

Abstract approved: ____

Yeongjin Jang

In recent years, Oblivious Random Access Memory (ORAM) controllers in Trusted Execution Environments (TEEs) have become a popular area of investigation, as coresident trusted systems allow for significantly more efficient oblivious execution. Further, in the case of Intel architectures, oblivious execution effectively eliminates the majority of confidentiality leakage holes in SGX. Unfortunately, the state of the art TEE-ORAM memory solutions for Intel SGX are still considered too slow for most applications, with memory block requests being handled at milliseconds latency. PRORAM, our novel oblivious memory controller, can deliver a block in the order of microseconds, approximately 10x–40x faster than prior work. This analysis will describe the design and implementation techniques that led to our significant performance gains.

The Symbioses of Oblivious Random Access Memory and Trusted Execution Environments

by

Hadi Rahal-Arabi

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 17, 2021
Commencement June 2022

Master of Science thesis of Hadi Rahal-Arabi presented on June 17, 2021.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

Hadi Rahal-Arabi, Author

ACKNOWLEDGEMENTS

I would like to thank my principal investigator and advisor Dr. Yeongjin Jang for his deep investment in my academic growth. Further, I would like to thank the collaborators in my lab, Andrew Quach and Cody Holliday, for their hard work on the PRORAM project.

Finally, to my loving family: thank you all, all of my work is born out of your support.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

Chapter 1: Introduction

For new organizations, the era of self-hosting is over. Cloud providers offer their tenants unprecedented levels of infrastructure scalability and flexibility, often accompanied by a reduction in management complexity. Unfortunately, several privacy sensitive entities cannot experience these benefits, because they cannot give up visibility into their operations and data o the infrastructure provider. Health care providers, journalists, and intelligence agencies all have motivation to avoid risking the use cloud services.

The Trusted Execution Environment (TEE) aims to solve this problem. A TEE is a hardware environment with greater security guarantees than those provided to general purpose programs. TEEs like Intel Software Guard eXtensions (SGX) provide excellent guarantees for memory confidentiality and program integrity, but choose not to protect against access pattern leakage. This has unfortunate consequences: in the right context, access pattern leakage can actually nullify the memory confidentiality guarantees [22].

The theoretical approach for defeating access pattern attacks is to leverage Oblivious Random Access Memory (ORAM). An Oblivious RAM scheme defines memory which forces access patterns to be probabilistically indistinguishable from random accesses, i.e. a witness with a transcript of all memory requests addresses cannot infer what data are being retrieved. Oblivious RAM schemes can be used for private access of cloud data, but they incur a heavy bandwidth overhead [29, 34, 28].

In the first ORAM proposal by Ostrovsky and Goldreich [11], ORAM is described as a theoretical approach to digital rights management. In their proposed system, an attacker with full memory access cannot determine the properties of programs, and thus would be unable to pirate it. The system model is a secure processor doing operations on an encrypted memory space: this is a remarkably familiar models for those familiar with contemporary TEEs. Their model predates the modern concept of trusted execution environments: it was invented to serve their analysis.

With time, this secure processor model was discarded in favor of a two-party model, in which a trusted client obliviously retrieves stored blocks from an untrusted server. This new model was developed alongside the proliferation of cloud computation; Researchers have turned to ORAM for additional privacy guarantees in outsourced storage.

Given the recent advent and popularity of Trusted Execution Environments, some ORAM designs are incorporating a system model very similar to that of the seminal ORAM paper. These investigations leverage modern two-party tree ORAM schemes, in which the trusted party will reside in in a TEE. By leveraging the security guarantees of TEEs, the ORAM bandwidth overhead can be incurred on a memory bus rather than a network, resulting in a relatively high performance ORAM. However, due to the bootstrapping problem of oblivious memory access in Intel SGX, e.g., doubly-oblivious construction, existing TEE+ORAM designs do not gain much performance over remote client constructions [26, 2, 13, 18].

To this end, this thesis shows that existing TEE+ORAM implementation are from optimal [26, 2, 13, 18]. Then, by constructing an implementation enhanced with several optimizations, we will demonstrate that the new ORAM can outperform existing ORAM constructions, and can support over 500 Mbps data transfer rate, which is 10x≈40x faster than existing solutions.

1.1 Contributions

Researchers have already the combination of ORAM and TEE to be symbiotic: Oblivious RAM defends any access pattern attack launched against a TEE, and TEEs provide the perfect high-performance environment to implement an ORAM. Typically, TEE+ORAM investigations evaluate the combination of Intel SGX and Path ORAM for a given use-case [29, 26, 2, 18] Their investigations are intended to demonstrate the practicality of ORAM when the bandwidth blowup is contained in the wide low-latency channel of the memory bus. We identify key performance issues in the state of the art publications, and make three contributions:

1. We provide an analysis for the best methods for selecting a theoretical ORAM for use in a TEE, regarding both the system level restrictions on enclaves and algorithmic complexity of ORAM protocol.

2. We devise and apply several optimizations to our SGX+ORAM implementation.

3. We evaluate our SGX+ORAM implementation with micro/macro-benchmark as well as application use case scenarios such as Google Key Transparency.

1.1.1 TEE+ORAM Performance Analysis

There is no defined methodology for selecting an ORAM scheme for use in a Trusted Execution Environment

In the case of ORAM in SGX, the factors that will impact request latency and throughput are not the same as those in the traditional two-party Tree ORAM setting. The ORAM components in SGX communicate via the memory bus, a high-speed high-throughput channel. This differs from the typical tree ORAM setting in which the ORAM client and server are networked components.

Path ORAM, the typical selection for TEE+ORAM implementations, was designed to minimize bandwidth overhead. This is not the best performance factor to optimize for when on a very high-bandwidth channel. In a trusted execution environment without fully trusted memory, ORAM clients must not perform any data-dependent branches, and they must access any client data obliviously. When there is a branch in a TEE+ORAM implementation, all execution paths must be taken so that the system can make no inference about the operations in the TEE. When there is client data access in an TEE+ORAM scheme, it must be replaced with an expensive oblivious access. Rather than optimizing for bandwidth in the TEE, an ORAM with minimal client storage and branching operations should be selected.

In Chapter 3, we provide the analysis that motivates the selection of Circuit ORAM [34] as the ORAM for implementations in SGX, and provide the tools for similar reasoning with arbitrary TEEs.

1.1.2 Optimization

We implement the Performant Recursive ORAM (PRORAM), to empirically verify our performance analysis. PRORAM improves the throughput and latency of SGX memory controllers by orders of magnitude for all reasonable use cases, and delivers blocks at the microsecond latency.

There are several challenges associated with creating a performant SGX+ORAM. Even in SGX's trusted memory, the enclave page cache, memory access patterns must be protected to fit the threat model of a malicious cloud provider. Protecting these access patterns performantly is a significant engineering effort.

Chapter 4 will summarize a critical subset of the optimizations, and chapter 5 will enumerate the PRORAM evaluations.

Chapter 2: Background

This section will introduce the concepts required for understanding ORAM, SGX, and their combination. Each important concept will receive some explanation, followed by a literature review of key publications.

2.1 Oblivious Random Access Memory (ORAM)

Encryption is not a holistic solution for maintaining privacy guarantees on collections of ciphertexts, as access patterns on encrypted data can leak valuable information about the respective plaintexts. In computer security, protected access patterns are called *oblivious*. We can trivially achieve obliviousness leakage by retrieving the complete set of ciphertexts for each access, but this linear technique becomes infeasible as the set of ciphertexts grows.

This problem has motivated the development of Oblivious Random Access Memory, a memory model that makes access patterns indistinguishable from random to any observer. A typical two-party ORAM scheme describes data structures and algorithms used by a trusted client to access encrypted storage on an untrusted server.

In their seminal paper on the subject [11], Goldreich and Ostrovsky showed that given an arbitrary database of N documents, accessing a single document obliviously would require $O(\log N)$ total accesses. Oblivious access schemes have applications in cloud privacy [29], secure multi-party-computation [34], and program obfuscation [1].

2.1.1 Literature Review

In this section, five theoretical Tree ORAM papers will be summarized in chronological order of their publication. With the exception of the original ORAM, we choose to summarize exclusively Tree ORAM implementations. Hierarchical and partition-based ORAM schemes have high worst-case latency, and our high performance ORAM is meant to be employed as a reliable block server. This is not a holistic review of ORAM, but provides the fundamental knowledge of the field that is required to understand the contributions of this thesis. The ORAM papers build on each other conceptually and historically, and each brings insights that were critical in the development of our systems ORAM solution.

First, Software Protection and Simulation on Oblivious RAMs [11], introduces the concept of oblivious random access memory. The first ORAMs were revolutionary, but had an issue with worst-case cost. On occasion, the ORAM blocks would need to be fully shuffled to maintain obliviousness. This made request times inconsistent: when a reshuffle is required, retrieving a single block is significantly more costly in time.

Recently, oblivious RAM with $O((\log N)^3)$ Worst-Case Cost [28] addresses this issue. The paper proposes the idea of organizing ORAM into a tree of blocks, and reshuffling a subset of the tree on each access. Path ORAM [29] soon followed, an ORAM derivative of Tree ORAM with better bandwidth bounds. Ring ORAM then optimized even further by reducing a constant time factor from Path ORAM [25].

Finally, we summarize Circuit ORAM [34]. In chapter 3 we contend that Circuit ORAM is the best selection for Intel SGX, but it was developed to have a minimal circuit size, for use in secure multi-party computation.

2.1.1.1 Software Protection and Simulation on Oblivious RAMs [11]

In an oblivious Turing machine, the tape's movements are agnostic to the input contents, but not the length. This means that two inputs of the same length always run for the same amount of time, and exhibit the movement behavior. Pippenger and Fischer showed that a two-tape oblivious Turing Machine can simiulate a one-tape Turing Machine with a logarithmic slowdown in time.

Oded Goldreich introduced the concept of Oblivious RAM in 1987, then formalized and published analysis alongside Rafail Ostrovsky in 1993. In their work, Goldreich and Ovstrovsky extend the concept of oblivious computation from Turing machines to the random access memory model. They propose the first ORAM scheme which can convert an arbitrary RAM program to an probabilistically oblivious RAM program, meaning an memory access pattern observer can make no inference about the data and execution state of a program.

Their first ORAM scheme has a relatively significant polylogarithmic complexity blowup in dummy accesses per real block access, but it serves as a foundation for

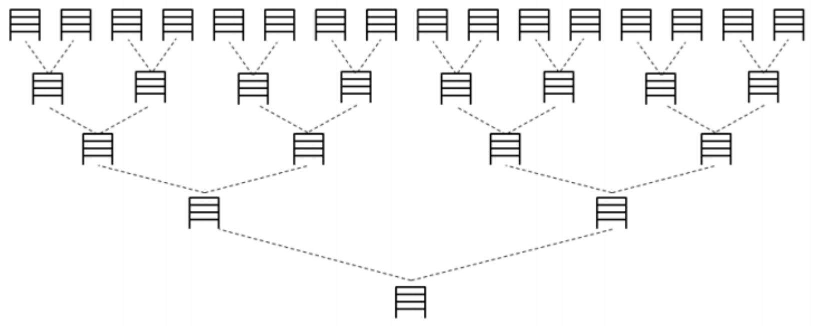


Figure 2.1: This figure from "Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost" [28] shows the binary tree structure her novel Tree ORAM paradigm

ORAM systems. The original ORAM scheme relied on occasional full reshuffles, in which all blocks in the ORAM would need to be moved into a new random location.

The seminal paper proved that there is a theoretical logarithmic lower bound for ORAM access overhead, a bound that has been reinforced in recent study.

2.1.1.2 Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost [28]

The first Tree ORAM paper ushered in a new age of ORAM. Shi et al. abandon the hierarchical ORAM paradigm proposed by Goldreich and Ovstrovsky, and propose a new organization for blocks of ORAM storage, now commonly referred to as Tree ORAM. Unlike the original ORAM, this scheme separates ORAM into two parties, a trusted client and an untrusted server. This paradigm is more suited to applications in trusted outsourced storage. Critically, Tree ORAM allows for sub-linear worst-case cost, whereas previous ORAM schemes employed an occasional full reshuffle on a request.

The scheme, visualized in Figure 2.1, works simply. New blocks that enter the ORAM do so at the root of the tree, then obliviously percolate their way towards the leaves via an ”eviction” algorithm. The key idea is to amortize the reshuffle operation by performing partial reshuffles (eviction) after every new read or write operation to the ORAM, rather than an occasional full reshuffle of the ORAM.

2.1.1.3 Path ORAM: An Extremely Simple Oblivious RAM Protocol [29]

Path ORAM is likely the most popular modern ORAM scheme, due to its simplicity and efficiency. Path ORAM has asymptotically better bandwidth cost than any previous ORAM with small client storage. The scheme stores all blocks in buckets in a binary tree, where a bucket is simply a collection of a constant number of blocks. Block positions in the tree are denoted by their path, and depth in the tree. A path begins at the root of the tree, and reaches one of the leaves. The easiest method of labeling a path is to simply denote the leaf that is on the path, as there can only be one unique path per leaf.

In Path ORAM, the client maintains two data structures, the stash and the position map. The position map contains the location of each block within the outsourced binary tree, and the stash contains a local cache of blocks.

The client algorithm does the following on any block access

1. Retrieve the current location of the requested block from the position map.

2. Randomly remap the location of the requested block in the position map.
3. Read the entire original path of the requested block, into the stash.
4. From leaf to root, evict every block from the stash that can return to the original path of the requested block.

The last step in the algorithm in the greedy eviction step. In deeper buckets on the tree, there are relatively fewer blocks that are eligible to be stored. Intuitively, since the root of the tree is on every path, any block can be stored in the root bucket. By choosing to write each block at its deepest possible location, the algorithm ensures that all blocks that can be evicted from the stash are placed on the server.

2.1.1.4 Constants Count: Practical Improvements to Oblivious RAM [25]

The Ring ORAM scheme aims to reduce the bandwidth requirements of Path ORAM by a constant factor. Ring ORAM makes the bandwidth independent of the bucket size: whereas Path ORAM will send an entire path of buckets on a request, Ring ORAM selects a single block from each bucket for access.

Ring ORAM achieves its goal by storing metadata in each bucket about the the ordering of the blocks. To perform an access operation, the client first reads all of the metadata of a single path. For the bucket containing the target block, the client will request the position in the metadata, and for all other buckets, a random dummy block will be read.

Ring ORAM borrows a trick from Burst ORAM [9] to reduce the bandwidth-per access to $O(1)$. The trick relies on the invariant that all blocks transmitted between the untrusted server and trusted client are dummy blocks, with the exception of the block of interest. Knowing this, the server can simply XOR all of the encrypted blocks requested by the client, and send the result. With the metadata requested in the first step, the client can then reconstruct the target block.

2.1.1.5 Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound [34]

Xiao Wang et al. develop Circuit ORAM, an ORAM scheme designed to have a minimal circuit size. The scheme is identical to the original Tree ORAM, with the exception of a modified eviction strategy.

Circuit ORAM is generally less efficient than Path ORAM, but for reasons that will be described in our design section, it is well suited for trusted execution environments. Wang also modernizes the proof for the ORAM logarithmic lower bound in his work.

2.2 Trusted Execution Environments (TEE)

In computer science theory, trusted outsourced computation typically relies on homomorphic encryption schemes. Given that homomorphic encryption is too costly for many applications, chipmakers have introduced Trusted Execution Environments

(TEEs) as a practical middle ground. A TEE is a processor component that has elevated guarantees of confidentiality and integrity. These guarantees typically rely on a combination of attestation techniques and memory encryption engines. Existing Trusted Execution Environments are vulnerable to different classes of side channel, denial of service, and control flow hijacking attacks.

2.2.1 Intel SGX

Intel Software Guard Extensions (SGX) is a trusted execution environment centered around trusted execution units called enclaves. The SGX threat model includes only the CPU and Intel's remote attestation mechanisms as the trusted entities. The enclave does not trust the operating system, chipset, cloud provider, or memory [8].

In this investigation, we leverage four of the security guarantees that can be provided by SGX enclaves. The security guarantees are listed below, and the vulnerabilities that threaten them will be listed in the SGX Attack Survey [20], the first paper in the SGX literature review.

1. **Local Confidentiality**: SGX provides local confidentiality with a proprietary Memory Encryption Engine (MEE), and an Enclave Page Cache (EPC). The EPC is an area of trusted memory on DRAM, and the MEE ensures that enclave memory never enters the EPC without first being encrypted.

2. **Local Integrity**: At any given time the EPC has verifiable integrity with a Merkle Tree construction that always resides in the EPC. If the MME discovers that the EPC and Merkle Tree mismatch, the CPU is halted.

3. **Remote Integrity**: Enclave binaries can expose an interface for remote attestation, allowing a third party to verify the trusted binary running on the enclave.

4. **Remote Confidentiality**: After checking the integrity of the enclave remotely, a third party may perform a key exchange to establish an encrypted channel with the enclave.

SGX can be applied in any scenario in which the above guarantees are beneficial, such as federated learning [17], trusted cloud computation [30], and digital rights management [3]. In the above list, the security property of obliviousness is noticeably absent: by design, Intel SGX chooses not protect the access patterns of enclaves.

2.2.2 Literature Review

Oblivious RAM defends a large subset of attacks against Intel SGX. Any attack that relies on access pattern inference can be defended with ORAM, and the majority of SGX confidentiality leakage attacks are driven by access pattern analysis.

This review will cover four papers that provide the relevant background for Intel vulnerabilities. The SGX Survey [20] summarizes the exploitable vulnerabilities in SGX. We then provide an overview of Spectre [14] and SGX-Step [5]. Spectre provides an example of secret leakage through analysis of data access patterns, and SGX-Step is the example of secret leakage via analysis of code access patterns. Together, these two papers motivate our threat model in Chapter 3. Finally, a summary of Raccoon [24] provides insight into oblivious primitives on x86 systems.

2.2.2.1 A Survey of Published Attacks on Intel SGX [20]

Figure 2.2, from the SGX Survey of attacks, lists 24 exploitable attacks against Intel SGX. The vulnerabilities are separated into the following impact categories:

1. Page access pattern These are attacks which are capable of leaking page access patterns to an attacker. Oblivious RAM has access patterns that are indistinguishable from random to an attacker, so SGX+ORAM constructions will not suffer any impact from these bugs.

2. Instruction trace Instruction trace bugs leak the execution patterns of enclave programs. In chapter 3, we discuss how these attacks can be defended by avoiding branching in the oblivious memory controller.

3. Memory access pattern Memory access pattern leakage is similar to page access pattern leakage, but at a finer granularity. They are similarly have no impact Oblivious RAM.

4. Memory Contents Attacks that leak memory contents are devastating to the SGX landscape. Fortunately, most of these attacks rely on cache side channels, which rely on access pattern leakage as a primitive. With the exception of RIDL [31], CrossTalk [23], CacheOut [32], and ZombieLoad [27], all of these bugs rely on either access pattern leakage or instruction pattern analysis, both of which are defended by Oblivious RAM.

5. Fault Injection Only a single attack, PlunderVolt, is listed in the Fault Injection category. The attack exploits the unpredictable behavior of Intel

CPUs when they are undervolted. This form of hardware attack cannot be defended by ORAM.

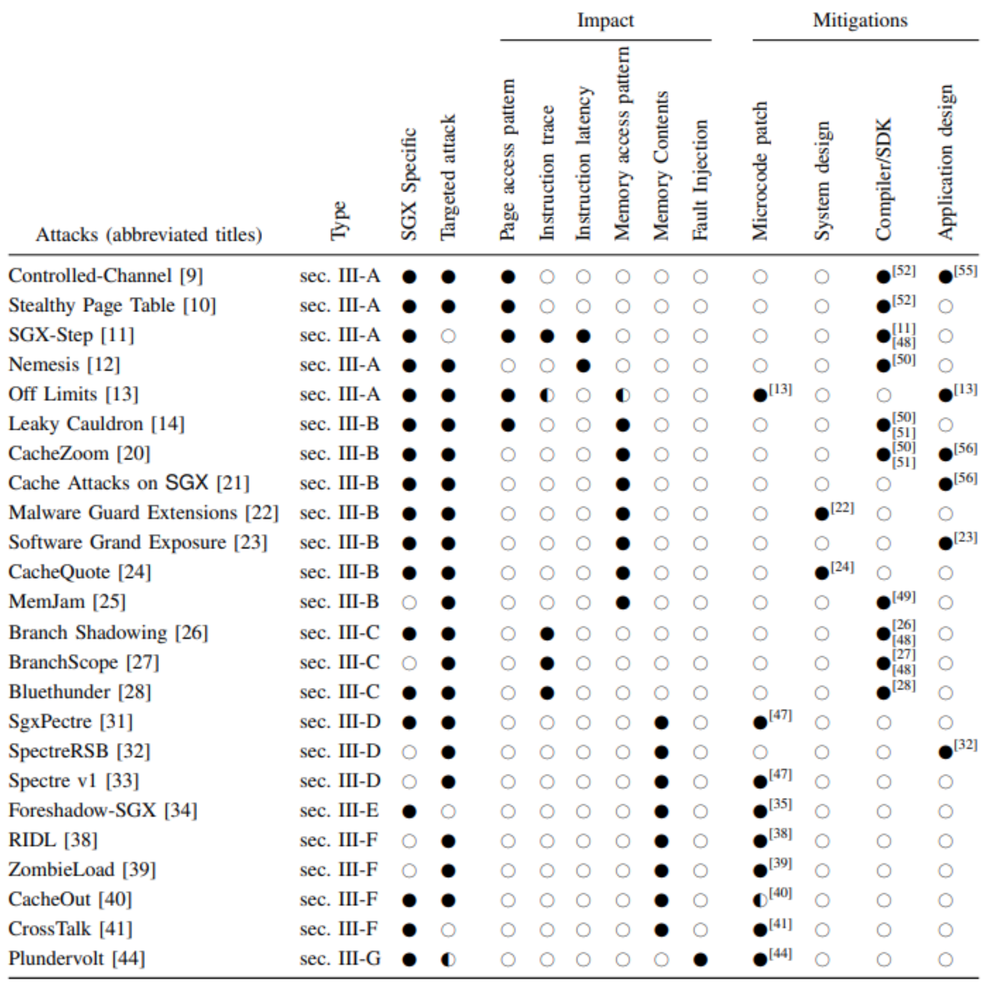
2.2.2.2 Spectre Attacks: Exploiting Speculative Execution [14]

Spectre is an attack paper that exploits CPUs that leverage speculative execution, a population that represents billions of microprocessors. Speculative execution is a processor optimization that mitigates the impact of waiting on memory requests.

As an example, imagine a branching condition that is contingent on boolean in memory. The processor can idle as it waits for the memory to be delivered, or it can cache it's current state and guess the branch that will be executed. If the guess is correct, the program will be in the correct state, and further ahead in the execution than it would be if it had not executed speculatively. If the guess is incorrect, the only performance consequence is that the processor must spend extra time rolling back to it's previous state.

Speculative and non-speculative execution share the cache, and the cache is not flushed when the branch predictor fails: this is the key insight that drives a spectre attack. This co-residency allows side-channel attacks to be launched to leak speculative state. Leaking speculative state is consequential: as speculative execution is not bound by memory protection mechanisms.

To control the leak, an attacker can "train" the speculative executor to take a certain branch, then pass parameters to the branch that will read privileged memory. The privileged memory will be loaded into the cache, and an attacker can



Attacks (abbreviated titles)	Type	SGX Specific	Targeted attack	Impact: Page access pattern	Impact: Instruction trace	Impact: Instruction latency	Impact: Memory access pattern	Impact: Memory Contents	Impact: Fault Injection	Mitigations: Microcode patch	Mitigations: System design	Mitigations: Compiler/SDK	Mitigations: Application design
Controlled-Channel [9]	sec. III-A	●	●	●	○	○	○	○	○	○	○	●[52]	●[55]
Stealthy Page Table [10]	sec. III-A	●	●	●	○	○	○	○	○	○	○	●[52]	○
SGX-Step [11]	sec. III-A	●	○	●	●	●	○	○	○	○	○	●[11] [48]	○
Nemesis [12]	sec. III-A	●	●	○	○	●	○	○	○	○	○	●[50]	○
Off Limits [13]	sec. III-A	●	●	●	◐	○	◐	○	○	●[13]	○	○	●[13]
Leaky Cauldron [14]	sec. III-B	●	●	●	○	○	●	○	○	○	○	●[50] [51]	○
CacheZoom [20]	sec. III-B	●	●	○	○	○	●	○	○	○	○	●[50] [51]	●[56]
Cache Attacks on SGX [21]	sec. III-B	●	●	○	○	○	●	○	○	○	○	○	●[56]
Malware Guard Extensions [22]	sec. III-B	●	●	○	○	○	●	○	○	○	●[22]	○	○
Software Grand Exposure [23]	sec. III-B	●	●	○	○	○	●	○	○	○	○	○	●[23]
CacheQuote [24]	sec. III-B	●	●	○	○	○	●	○	○	○	●[24]	○	○
MemJam [25]	sec. III-B	○	●	○	○	○	●	○	○	○	○	●[49]	○
Branch Shadowing [26]	sec. III-C	●	●	○	●	○	○	○	○	○	○	●[26] [48]	○
BranchScope [27]	sec. III-C	○	●	○	●	○	○	○	○	○	○	●[27] [48]	○
Bluethunder [28]	sec. III-C	●	●	○	●	○	○	○	○	○	○	●[28]	○
SgxPectre [31]	sec. III-D	●	●	○	○	○	○	●	○	●[47]	○	○	○
SpectreRSB [32]	sec. III-D	○	●	○	○	○	○	●	○	○	○	○	●[32]
Spectre v1 [33]	sec. III-D	○	●	○	○	○	○	●	○	●[47]	○	○	○
Foreshadow-SGX [34]	sec. III-E	●	○	○	○	○	○	●	○	●[35]	○	○	○
RIDL [38]	sec. III-F	○	●	○	○	○	○	●	○	●[38]	○	○	○
ZombieLoad [39]	sec. III-F	○	●	○	○	○	○	●	○	●[39]	○	○	○
CacheOut [40]	sec. III-F	●	●	○	○	○	○	●	○	◐[40]	○	○	○
CrossTalk [41]	sec. III-F	●	○	○	○	○	○	●	○	●[41]	○	○	○
Plundervolt [44]	sec. III-G	●	◐	○	○	○	○	○	●	●[44]	○	○	○

Figure 2.2: This figure from the SGX attack survey [20] lists the current viable attacks against Intel SGX.

launch a cache side-channel attack to infer the contents.

Spectre breaks many privilege isolations, including that between an operating system and SGX enclave.

2.2.2.3 SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control [5]

SGX-Step is a paper and framework describing methods of malicious control of enclaves from user-space.

Of particular interest is the capability for a host system to instrument checkpoints at every instruction in an enclave. This fine-grained control allows for inference instruction counting side channels. The consequence is that an untrusted host has the ability to infer which branches are being taken by an enclave program by observing instruction counts and memory transactions.

2.2.2.4 Raccoon: Closing Digital Side-Channels through Obfuscated Execution [24]

Raccoon is the first proposal of oblivious primitives as a defense against side channel attacks. For this thesis, the relevant contributions of the raccoon can be distilled into a code snippet, shown in Figure 2.3.

The code leverages the cmov instruction present on x86 architecture processors. cmov, or conditional move, takes in a condition, a source location, and a destination location. If the condition is fulfilled, the source operand is written to the destination operand. Regardless of whether the condition is fulfilled, the source operand is read.

Since the memory source operand will always be read, conditional move is an excellent primitive for oblivious loads and stores. The primitive can be applied to ”dummy” data or real data, with the condition determining whether or not the read is real. In Raccoon, this primitive is leveraged in the execution of decoy paths: code paths that are execute entirely useless code, and discard the result. Raccoon combines this primitive with a system model that includes encrypted memory and intermixed decoy/real execution paths. At the time of it’s publication, Raccoon was the best systems approach to execution obfuscation.

```
cmov(uint8_t pred, uint32_t a, uint32_t b){
    uint32_t result;
    __asm__volatile(
        "mov %2,%0;"
        "test %1,%1;"
        "cmovz %3,%0;"
        "test %2,%2;"
        : "=r"(result)
        : "r"(pred), "r"(a), "r"(b)
        : "cc"
    );
    return result;
}
```

Figure 2.3: This code snippet, first seen in Raccoon [24] represents the current state of the art in oblivious memory moves. With no branching or data dependent memory access, we can obliviously select a word based on our predicate.

2.3 SGX + ORAM

Researchers have identified the combination of ORAM and TEE to be symbiotic, often by evaluating the combination of Intel SGX and Path ORAM for a given use-case [26, 2, 18]. These investigations are intended to demonstrate the practicality of ORAM when the bandwidth blowup is contained in the wide low-latency channel of the memory bus.

2.3.1 Literature Review

This review covers four SGX+ORAM papers. SGX is the most popular ORAM for implementation in TEE, and this selection of papers covers the highlights of a relatively novel field.

First, ZeroTrace [26], is the first known SGX+ORAM investigation, and the PRORAM project attempts to push their state of the art implementation to new limits. Second, Oblix [18], implements an alternative form of position map to avoid the overhead of position map access in SGX+ORAM schemes. Although Oblix is called a search index in its title, it does not support many-to-many mappings between indicies and data objects, so it cannot function as a search index. POSUP [13] fixes this issue by designing and implementing a searchable ORAM in Intel SGX. Finally, MOSE [12] extends SGX+ORAM to be allow for multiple-users with defined access control rules to connect to the same ORAM server.

2.3.1.1 ZeroTrace: Oblivious Memory Primitives from Intel SGX [26]

ZeroTrace combines SGX and ORAM to create a secure oblivious memory controller. Their investigation is the current state of the art in SGX oblivious memory controllers.

ZeroTrace provides a baseline implementation to compare with our own. We compare their results to ours in chapter 5. The ZeroTrace implementation includes a comparison between Path and Circuit ORAM memory controllers, as well as a benchmark for recursive ORAM in SGX. ZeroTrace contends a higher overhead for Circuit ORAM when compared to Path ORAM, claiming that the overhead of entering the enclave repeatedly outweighs benefits of Circuit ORAM. In chapter 5, we demonstrate that this should not be the case with correct implementation.

2.3.1.2 Oblix: An Efficient Oblivious Search Index [18]

Oblix takes a different approach than ZeroTrace; because recursive ORAM is heavy with Path-ORAM in doubly-oblivious setting, which Intel SGX enclave falls in, the work discusses reducing the overhead of recursive ORAM. Recursive ORAM is a technique for storing an ORAM position map on the untrusted server, in its own ORAM. To access an block in the top-level ORAM (e.g. the one containing non-metadata blocks), first the position ORAM will be queried to retrieve a position map entry, then the entry is used to query the top-level ORAM This construction can be nested at arbitrary depths, i.e. we can also have an ORAM describing the position map for the position map ORAM.

Oblix devises a method of retrieving positions *alongside* the data, rather than before it. This scheme produced arguably more performant results than ZeroTrace, but is built for different use cases. Each block identifier (key) is treated as a search index, in a one-to-many relationship with several blocks: this is atypical, as most ORAM implementations have unique mappings between ids and blocks. When a query is made on a key, the top results for that key are returned. To prevent leakage, a fixed number of ”top” blocks is always retrieved. The scheme can be less efficient than ZeroTrace in the case that single blocks need to be accessed.

2.3.1.3 Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset [13]

POSUP implements an oblivious system for many-to-many mappings between search terms and data blocks. This enables performant searchable ORAM. The key elements of this paper can be distilled into the architecture and oblivious search algorithm.

POSUP contains two top-level ORAM trees, the index tree and data tree. In the paper, they are referred to as the keyword hash table and database respectively. The index tree is an inverted index: each block in the index tree maps a keyword to several block IDs in the database. After getting the list of database entries mapped to a keyword, all entries are retrieved and returned to the user. POSUP accepts a set of keywords as an input, and returns a set of blocks that map to the keyword on output. The number of blocks that map to individual keywords is not leaked, but the total number of blocks associated with the set of keywords is leaked.

2.3.1.4 MOSE: Practical Multi-User Oblivious Storage via Secure Enclaves [12]

Figure 2.4, from the MOSE paper, shows the high-level architecture of the scheme.

Each ORAM block has an associated block in an ORAM Tree of metadata blocks. The metadata blocks are of fixed size, and define the users that have access to a given block. Accessing a block in MOSE is a simple task: a user sends their credentials, as well as the block they would like to retrieve. The credentials are checked against the metadata tree, and if they are accepted, the block is retrieved from the ORAM tree.

MOSE is implemented by the authors for empirical performance testing, and we compare their results with ours in chapter 5. It is noteworthy that MOSE selects Circuit ORAM as the scheme for its high-performance implementation. We agree with the authors on this selection, and formalize their reasoning in our investigation.

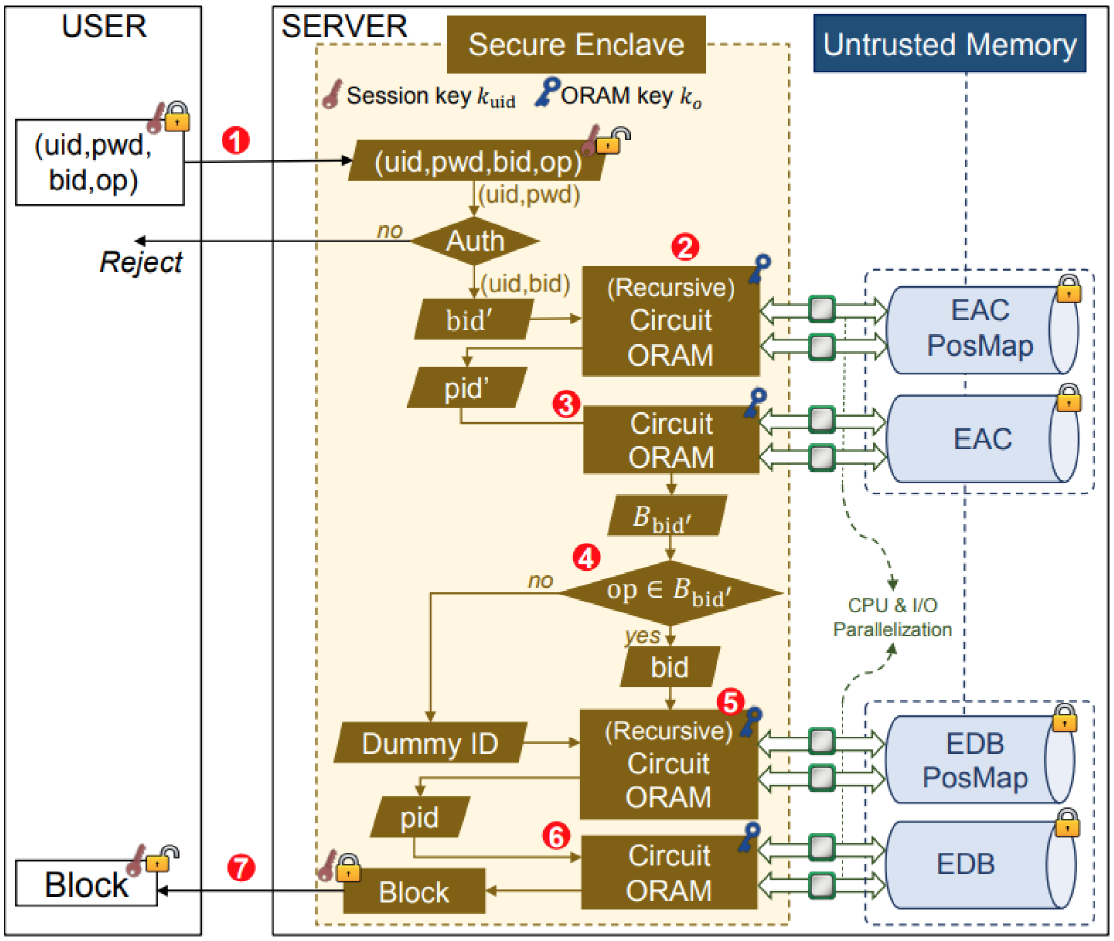


Figure 2.4: The MOSE architecture

Chapter 3: Analysis

This chapter covers three areas of critical analysis. The first is a security analysis and threat model, which imposes some performance limitations on the ORAM implementation. The second is an analysis and description of the high level components that are necessary and present in any TEE+ORAM implementation. The final section describes the first contribution of the PRORAM project: an analysis of the performance factors in a TEE ORAM, informed by our threat model and architecture.

3.1 Threat Model

This threat model is specific to SGX, but several of the assertions are generalizable to threat models in alternative TEEs. The trusted computing base includes the CPU and the enclave binary. We assume that the ORAM controller is implemented correctly, as bugs in enclave code can be exploited to gain arbitrary control of SGX [16]. We can consider the chipset, hypervisor, memory, network, operating system, and cloud provider untrusted, and show below what is required to thwart attackers on these systems.

We consider two adversaries in our threat model. The first adversary witnesses exchanges between the cloud storage user and the ORAM server. This adversary is trivially defeated by performing a key exchange with the enclave and using randomized encryption to communicate with the server.

Another common adversary model in TEE ORAM considers a software-based attacker. In the cloud setting, this attacker is commonly imagined to be a malicious co-tenant on rented infrastructure, snooping on the neighbors execution. Alternatively, the attacker is represented as malware with kernel privilege. This threat model is does not demonstrate the full power of the ORAM TEE symbiosis: with remote attestation guarantees, *even the cloud provider can be considered malicious*. In our investigation, we consider our second attacker to be the cloud provider. As long as the processor is in the TCB, we can thwart attackers anywhere on the system, including the memory bus.

Thwarting the attacker on the cloud provider requires enforcing that the ORAM controller never branches, as the attacker can leverage instruction-tracing attacks to infer which branches are taken [20, 5]. Alternatively, an attacker on the memory bus can track the access pattern across code pages. SGX ORAM controllers can avoid branching by leveraging the oblivious primitives introduced in Raccoon [24], which we improve upon in this project.

3.2 Architectural Components

Figure 3.1 shows the high level process of retrieving a block from a TEE ORAM. A short description of the components will be provided here.

Client The simplest component in the PRORAM design, the client performs

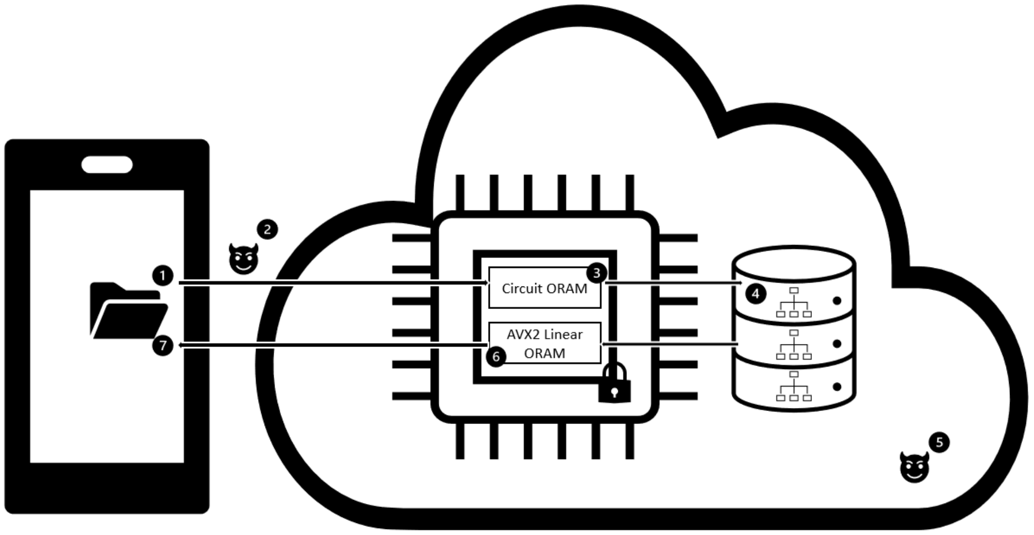


Figure 3.1: A high level overview of TEE ORAM Components. The two considered adversaries denoted on the figure are those snooping on the network transactions, and those with root access to the cloud host

remote attestation with the trusted environment, ensuring integrity of the remote program. It then establishes an encrypted channel for communication with the enclave. After these steps are completed, the client may simply treat the enclave as an addressed block server.

Position Map The position map is an oblivious data structure which maps block identifiers to locations. In the case of Tree ORAMs, the location is stored in the form of a leaf node, where we maintain an invariant the the block will remain in a bucket on the path from the root node to the specified leaf. Since this client data must be accessed obliviously, we borrow the concept of Recursive ORAM: the position map will either be a small map that can be linearly scanned, or an ORAM tree of positions.

Core ORAM The core ORAM must be built to a provably secure specification. When implementing a classical two-party ORAM with a trusted client and untrusted server, this core represents the client logic. To be resilient in our threat model, the client must be modified to be oblivious: there must be no data-dependent access-patterns or branching.

Encryption Pipeline Moving data in and out of the trusted memory requires re-encryption. To maximize performance when possible, the encryption should be hardware accelerated, and the encryption pipeline should be saturated.

Linear Scanner During recursive ORAM access, each position map block contains an array of positions. For example, in a 256-byte block configuration, where each position is represented as a 32-bit integer, a block of the position map will contain 64 positions. This small array must be accessed obliviously, but it would be inefficient to store each position in another layer of the ORAM: When dealing with small constants, the polylogarthmic overhead of ORAM overshadows the linear overhead of a full scan.

3.3 Performance Factors

In this section, we describe the strategy employed to select an ORAM for use in a trusted execution environment. In the traditional Tree ORAM setting, the controller has a remote client, and often the most significant cost in terms of time is the bandwidth blowup between the client and server [29]. Figure 3.2 compares

Scheme	**Bandwidth Cost (Blocks)**	**Rounds**
Circuit	$4Z \log N$	$3 \log N$
Path	$2Z \log N$	2
Ring	$3.5 \log N$	$2 + \frac{1}{v}$

Figure 3.2: Listed bandwidth cost values ignore the cost of metadata transfer, which is dominated by block transfer with sufficient block size. Ring ORAM employs a parameterized eviction rate v.

the bandwidth costs of three of the theoretical ORAMs considered as candidates for the PRORAM implementation.

It is critical to recognize that the factors that influence performance in theoretical ORAM schemes can be overshadowed by requirements of the systems used in their implementation. As our first contribution, we define and analyze the following performance factors, which are the most critical in SGX+ORAM implementations. We note that the computational overhead of non-homomorphic ORAM schemes tends to be quite low [34, 29, 11, 28]. When in a very high-bandwidth environment, the natural place to check for bottlenecks is in the copy and encryption operations performed by the ORAM controller. The following list denotes the performance factors of an ORAM scheme in an SGX+ORAM scheme:

Encryption Overhead The number of blocks that must be re-encrypted by the ORAM protocol for the access of a single block.

When writing an ORAM for a TEE, this is the number of blocks that must enter trusted execution environment during a request.

Even with hardware-accelerated instruction sets, encryption is computationally intensive, so it is critical to select an ORAM that requires minimal transfer between the trusted client and untrusted server.

Access Overhead The number of block reads and writes that must be performed by the protocol for the access of a single block.

Adapting ORAM clients to be oblivious often causes a blowup in this factor.

B, Bucket Size The number of blocks in a bucket, encryption and access usually often scale with this term. In a Tree ORAM, copy and encryption transactions are often in path granularity, and the size of a path scales with the bucket size.

S, Stash Size The number of blocks in the client stash, access overhead often scales with this term. Access overhead scales with this term because the stash is a piece of client data that is intended to be implemented in trusted memory. The enclave does not protect access patterns, so we must access the stash obliviously, incurring a significant overhead.

Figure 3.3 describes our performance factors analysis for three candidate Tree ORAMS. The access overhead is a sum of the dummy accesses that must be made for the ORAM in the typical setting, and the number of dummy accesses that must be added to avoid branching in the controller. The encryption overhead is simply determined by counting the number of blocks that are passed between the client and server in the networked ORAM setting, as these are the blocks that will be re-encrypted in an ORAM access.

The reader may note that all candidate ORAM schemes below are Tree ORAMS:

Scheme	**Encryption**	**Memcpy**	**Bucket Size (Z)**	**Stash Size (S)**
Circuit	$3Z \log N$	$S + Z \log N$	2-5	10-20
Path	$2Z \log N$	$SZ \log N$	4-6	53-120
Ring	$\log N + \frac{\log N \cdot Z}{A}$	$SZ \log N$	4-6	32-595

Figure 3.3: When implementing ORAM in a Trusted Execution Environment, bandwidth cost is only relevant if it will saturate the memory bus. The more relevant performance factors are those that require the system to perform more encryption operations and memory moves.

we do not analyze hierarchical ORAMs, as hierarchical ORAMs are only valuable in use-cases where worst-case latency can be very large.

We contend that it is most critical to select an ORAM with minimal block transfer between the server and client, and to limit transfer between trusted and untrusted memory as much as possible. The most efficient ORAM for our use-case is Circuit ORAM, this judgement will be detailed in the following optimization chapter.

Chapter 4: Optimizations

We developed an ORAM implementation, PRORAM, to verify our design strategy. We select Intel SGX as our trusted execution environment, and circuit ORAM as our oblivious memory protocol. This section will enumerate the implementation decisions made to maximize oblivious memory controller performance.

It is critical to read the optimizations section with the context of the systemic and algorithmic constraints of an SGX+ORAM implementation.

1. Intel SGX has no fully trusted storage.

2. Intel SGX's semi-trusted storage, the EPC, only holds 128MBs. The EPC is considered semi-trusted because its access patterns are leaked, but it otherwise has functional guarantees for confidentiality and integrity.

3. The position map for ORAM is often larger than the EPC.

4. The ORAM implementation can perform no data-dependent branching.

5. Blocks that are moved from the EPC to memory must be encrypted. Encryption comes at significant computational cost.

4.1 Circuit ORAM

While the encryption overhead of circuit ORAM is higher than the other candidates, the Memcpy overhead is significantly lower.

The table in Figure 3.3 makes the choice simple. Consider that the access overhead for a single block is roughly $(3Z \log N)E + (S + Z \log N)M$, where E is the amortized time it takes to encrypt a block, and M is the amortized time it takes to move a block. The corresponding overhead for Path ORAM is $(2Z \log N)E + (SZ \log N)M$.

From these formulas, we would like to produce a single metric that roughly compares the overhead of the two schemes. First we substitute typical configuration parameters from the respective publications.[1] Then fix M to be 1, and define E to be the number of memory copies that can be performed in the time it takes to encrypt a fixed number of bytes. For example, in our target system, copying a block is roughly 2 times faster than encrypting a block, so we fix E to be 2. This produces an output that can be defined in copy overhead, e.g. the output of the formula is defined as the number of memory copies that could be performed in the time of a single ORAM access.

Figure 4.1 shows the resulting overhead relationship: Path ORAM only exceeds the performance of Circuit ORAM at unreasonably large capacities.

[1]For Path ORAM, S=80, Z=10. For Circuit ORAM, S=10, Z=2.

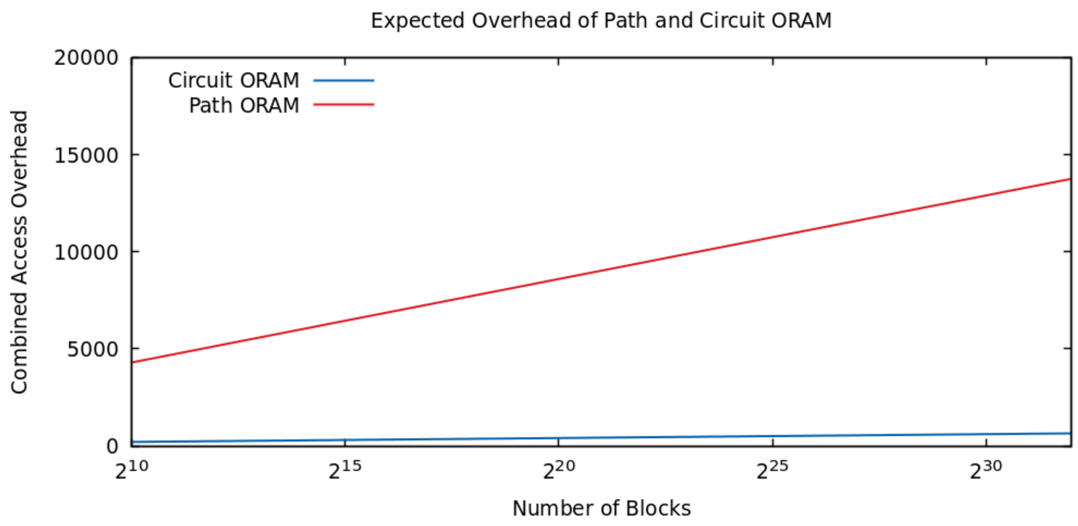


Figure 4.1: On our target system, and for any reasonable number of blocks, Path ORAM cannot compete with Circuit ORAM. Combined Access Overhead is a value representing the combined overhead of encrypting and copying blocks during an ORAM access.

4.2 Recursive Position Map

PRORAM implements a recursive position map (RPM) to avoid leaking access patterns on the position map. A significant portion of the time spent accessing an ORAM is spent accessing the rpm, so it is critical to optimize. We implement three optimizations:

1. EPC Tree Caching
2. Simultaneous Access and Update
3. Empirical Recursive ORAM parameterization

Position map ORAMs are notably smaller than their parent ORAMs: where b is the number of bytes required to represent a position, and B is the size of a block, each recursive ORAM will be smaller than their parent by a factor of $\frac{b}{B}$. Given this fact, many recursive ORAM trees are small enough to be cached entirely within the EPC, eliminating the encryption overhead of the map. By caching the tree in the EPC, we avoid the encryption overhead associated with moving the blocks between the enclave and untrusted memory.

On each ORAM recursive access, it is critical to retrieve and update the position simultaneously, otherwise position map accesses will scale exponentially with recursive depth, rather than linearly. For n position maps, if the position maps of the inner ORAMs are read and updated separately, the total time to access a block will be $\sum_{i=0}^{n} 2^i t_i$, where t_0 is the time it takes to retrieve a single block from the data ORAM, t_1 is the time required to retrieve a block from the first position map ORAM, etc. If we update the position as we access it the latency is simply $\sum_{i=0}^{n} t_i$.

It is not required for the RPM to share the same block size as the data ORAM. For each configuration (block size, capacity) we test, we run the controller with several different recursive ORAM block sizes, and select the best-performing.

4.3 Coalesced Blocks

It is often performant to configure an ORAM block to be larger than the intended unit of storage, then combine multiple units into a block. As an example, when storing 4KB images in oblivious cloud storage, it could be useful to configure the ORAM to have 8KB blocks, and obliviously select the desired image linearly after block retrieval. We refer to this optimization as *block coalescing*.

4.4 AVX2 Primitives

Vector registers are ultra-wide processor registers used for processing large amounts of data. In several cases, their proliferation has significantly improved the speed of memory copy and set operations. We can similarly leverage vector registers, such as AVX2 in the case of x86 microprocessors, to improve the speed of oblivious memory sets and copies.

Figure 4.2 shows an example of an AVX2 memory primitive used in PRORAM. The code will obliviously select 8 bytes from a memory region of 256 bytes.

The function takes in the memory region as input, as well as an index representing the offset of the 8-byte aligned target. First, the input 256 bytes are read across 8 vector registers, ymm0 to ymm7. We then perform the following operations on registers ymm1-ymm7

1. Check if the input index is references memory that is in the current current register. If so, obliviously set the our predicate -1 with cmov primitives, otherwise is obliviously set to 0. Since the predicate is an unsigned integer, setting its value to -1 will populate 8-byte integer with a bitvector of all 1s.

2. Use the ”vpbroadcastq” instruction to broadcast our predicate into a scratch register, ymm9. The scratch register now contains all 1s if the earlier in the current register is the target, or all 0s if is not.

3. Use the ”vpblendvps” to blend the current vector register with ymm0, using the contents of the scratch register as a mask. If the scratch register contains all 0s, then ymm0 will remain unchanged, otherwise it will be populated with the contents of the current register.

After performing those operations on registers ymm1-ymm7, ymm0 will contain our target bytes. However, since our target is 8 bytes, we now simply perform an oblivious linear scan on the array to retrieve the target 8 bytes.

Several analagous primitives exist for setting memory regions, as well as general purpose oblivious memcpy primitives. They will be published in the near future with the open-sourcing of PRORAM.

```
uint64_t
get_8_bytes_from_256_bytes(void *src_256, uint64_t idx) {
    uint64_t r_idx = idx / 4;   // row index: ymm
    uint64_t c_idx = idx % 4;   // column index
    asm volatile(""
            "vmovups 0x0(%[mem]), %%ymm0\n\t"
            "vmovups 0x20(%[mem]), %%ymm1\n\t"
            "vmovups 0x40(%[mem]), %%ymm2\n\t"
            "vmovups 0x60(%[mem]), %%ymm3\n\t"
            "vmovups 0x80(%[mem]), %%ymm4\n\t"
            "vmovups 0xa0(%[mem]), %%ymm5\n\t"
            "vmovups 0xc0(%[mem]), %%ymm6\n\t"
            "vmovups 0xe0(%[mem]), %%ymm7\n\t"
            :
            : [mem] "r" (src_256)
            : "memory", "ymm0", "ymm1", "ymm2", "ymm3", "ymm4", "ymm5", "ymm6", "ymm7");


    // select target row and store that to ymm8 (use vblendvps)
    uint64_t row_pred;
    row_pred = -check_equal(r_idx, 1);
    asm volatile(""
            "vpbroadcastq (%[pred]), %%ymm9\n\t"
            "vblendvps %%ymm9, %%ymm1, %%ymm0, %%ymm0\n\t"
            :
            : [pred] "r" (&row_pred)
            : "memory", "ymm9", "ymm1", "ymm0");

    ...

    // now ymm0 contains the target row
    uint64_t values[4];

    asm volatile(""
            "vmovups %%ymm0, (%[mem])\n\t"
            :
            : [mem] "r" (values)
            : "memory", "ymm0"
            );

    // select 1 data from ymm via blend
    uint64_t preds[4];
    for(int i = 0; i < 4; i++){
        preds[i] = check_equal(c_idx, i);
    uint64_t ret = -1;

    for(int i = 0; i < 4; i++){
        ret = select_value(preds[i], values[i], ret);
    }

    return ret;
}
```

Figure 4.2: This code shows how PRORAM leverages AVX2 instructions to vastly speedup oblivious retrieval The "..." represents a repetitious portion of code described in §4.4: predicate-setting and inline asm are repeated for ymm2-ymm7

Chapter 5: Evaluation

5.1 Experimental Setup

5.1.1 Hardware

Evaluation is done on a machine with a 10-core Intel i9-10900K processor, with the CPU running on a fixed 4.7 Ghz frequency. The machine is equipped with 128GB of RAM running at 3200 MT/s.

When comparing the evaluations in this dissertation, note that we evaluate on state of the art hardware. For example, Zerotrace runs evaluations on a 4-core Skylake with 64-GB of RAM. It is reasonable to assume that ZeroTrace would run significantly faster on our experimental setup than their own, it is easy to imagine a 3-fold improvement in their latency and throughput. However, it will be shown that our implementation runs over an order of magnitude faster than any SGX-ORAM controller, the significance of our improvements cannot be attributed to the hardware.

5.1.2 Default Configuration

A general-purpose ORAM controller must be configurable to be valuable. This presents a challenge for evaluation: each parameter option results in an exponential

Option	Default	Description
RPM	ENABLE	Whether the controller should use an RPM
N	262144	The number of memory blocks stored in ORAM
B	8192	The size of an memory block, in bytes
Z	2	The number of blocks in a bucket
b	256	The size of a block in the recursive position map
z	2	The size of a bucket in the recursive ORAM
C	1	The coalesce factor, described in §4.3
THREADS	1	The number of AESNI threads

Figure 5.1: The default PRORAM evaluation configuration

blowup of configurations to be evaluated. The PRORAM evaluation strategy is to define a default configuration, then isolate the parameter under test for changes.

For each evaluation, parameters are isolated for test from the configuration in Figure 5.1. If the value of a parameter is not clearly defined in a below evaluation, it has the default value.

5.2 Circuit vs Path ORAM

All memory copy operations are logged during an access in our implementation. Figure 5.2 shows that the copy overhead of Circuit and Path ORAM align with the analysis in Chapter 3.

Figure 5.3 demonstrates the effect of the reduced copy overhead on the latency of a single block request: Circuit ORAM results in significant performance gains.

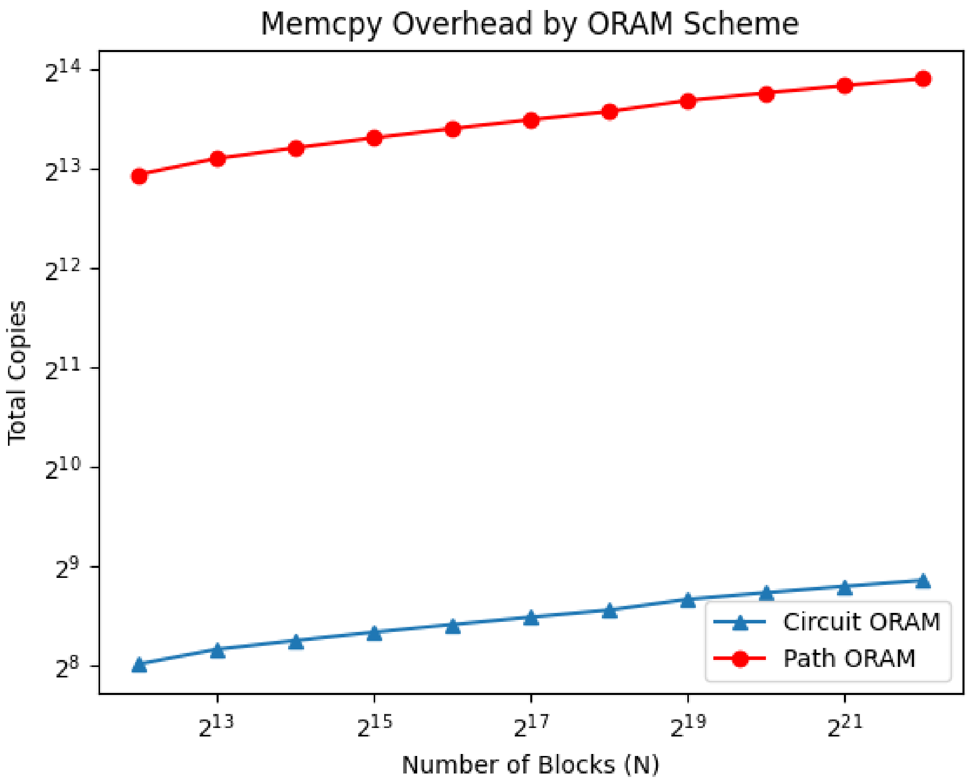


Figure 5.2: The number of memory copies made in a single Path/Circuit ORAM access.

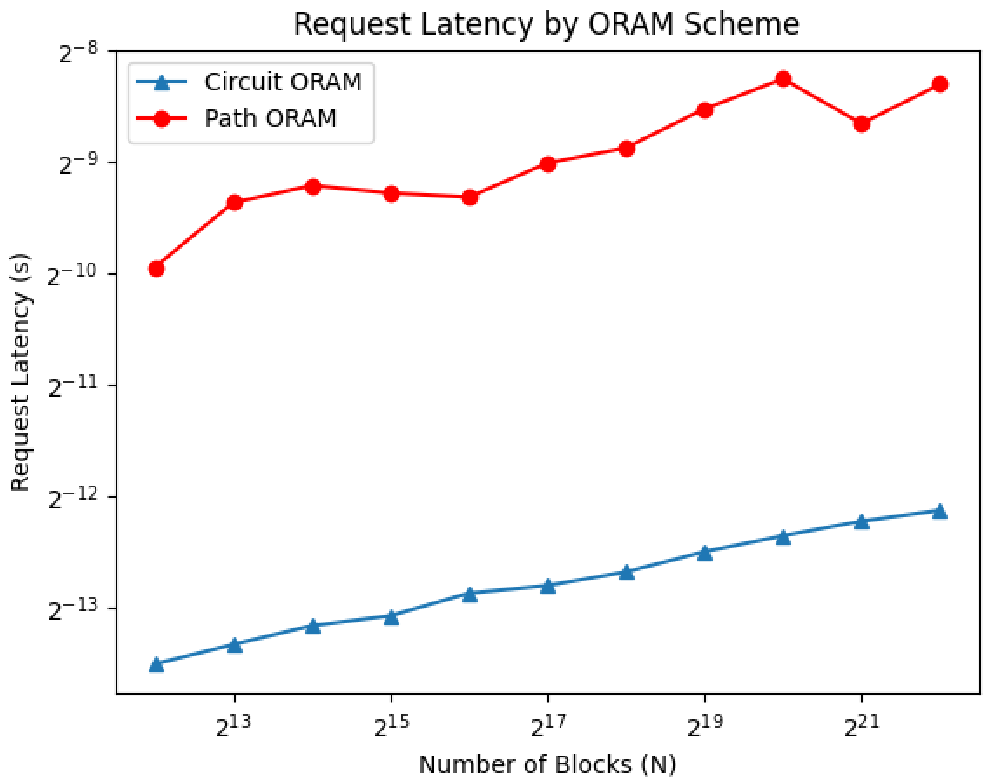


Figure 5.3: A comparison of request latency between Path and Circuit ORAM.

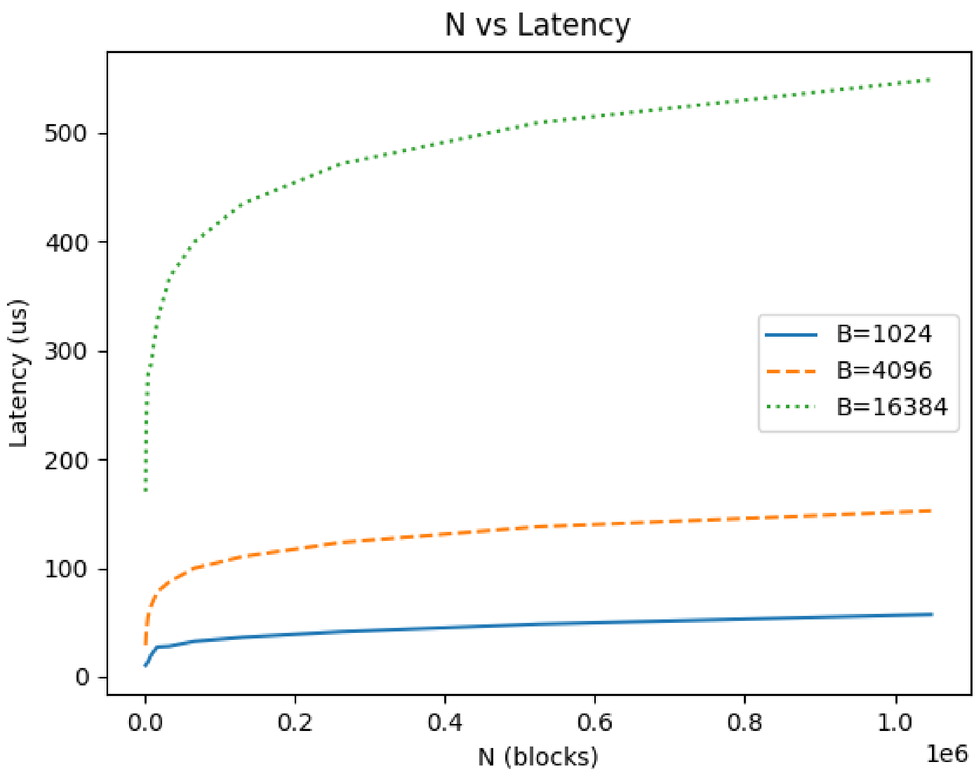


Figure 5.4: As expected from a Tree ORAM implementation PRORAM latency scales logarithmically with the number of blocks, and lineraly with the size of a block.

Block Size (bytes)	1	8	16	32	128	256	448
Bandwidth (MB/s)	2042	8542	8610	9160	10561	10486	10797
Memcpy Ratio	13.18	55.13	55.57	59.12	68.17	67.68	69.69

Figure 5.5: This table compares the bandwidth of traditional memory copies and our oblivious memory copy implementation.

5.3 Key-Value Storage

Figure 5.4 shows the scaling of our ORAMs block request latency against the capacity of the ORAM.

Note that in it's best configuration, with 100 1KB blocks, ZeroTrace achieves latency of 1ms. Figure 5.4 shows that with the same blocksize, and 10000 times the number of blocks, we still achieve a 20x increase in speed and throughput.

5.4 AVX2

Figure 5.5 shows a direct comparison between the speed of a regular memory copy, and our AVX oblivious memory copy.

Figure 5.6 shows the effect of AVX optimizations on our ORAM implementation. At small blocksizes, there is no reason to use vector registers, as a 64-bit general purpose register can transfer a significant portion of the block. As the block size grows, we see a consistent and significant improvement in throughput.

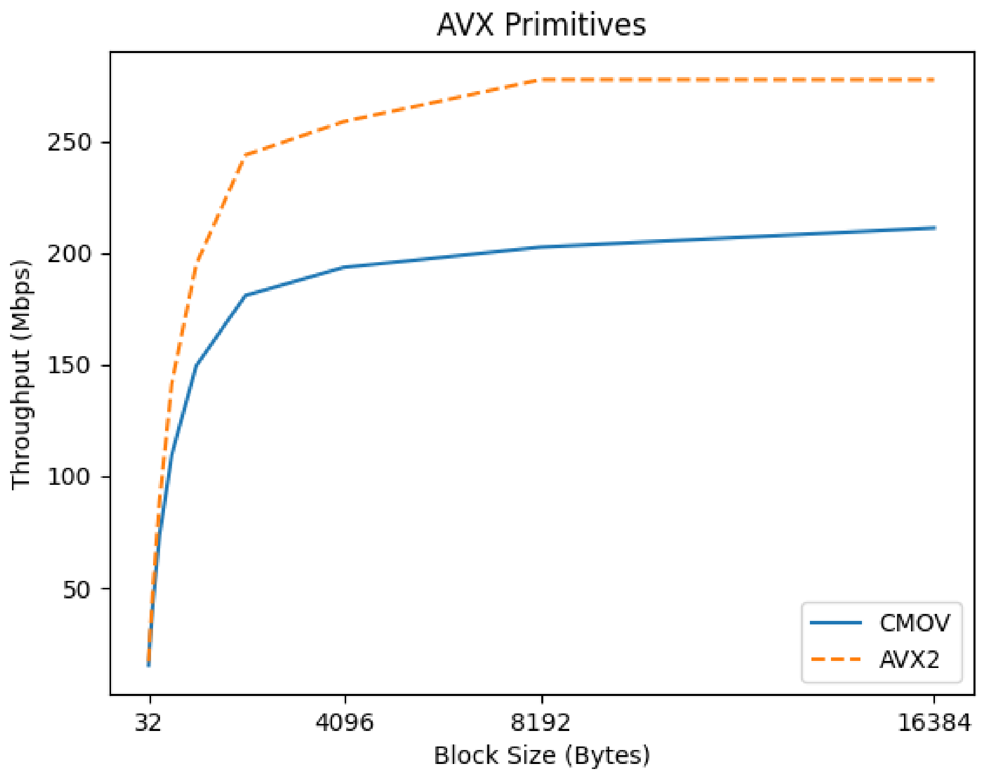


Figure 5.6: The above graph shows PRORAM running with CMOV and AVX2 primitives. As expected, vector registers allow for a significant increase in throughput.

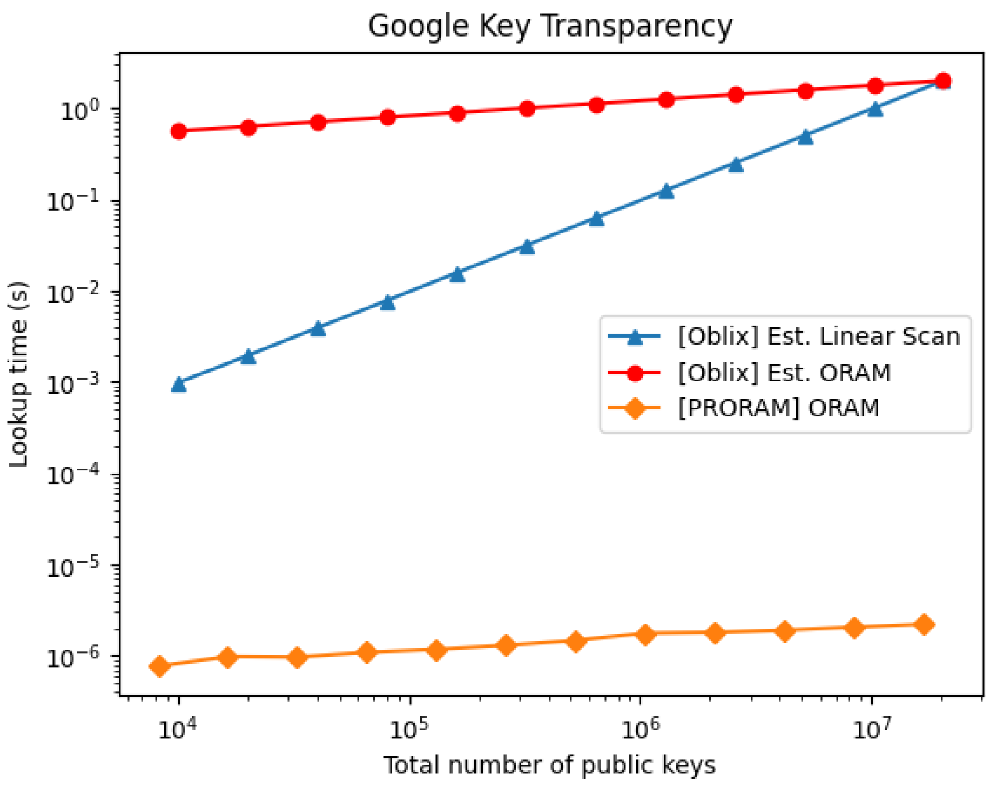


Figure 5.7: PRORAM achieves anonymous key transparency at practical rates.

5.5 Key Transparency

Google Key Transparency is a lookup service that is commonly used for distributing public keys. The service maintains a Merkle prefix tree of all user keys, and distributes the root hash amongst all of the users. When a key is requested, the service returns a proof of integrity for the key, containing the siblings of all of the nodes in the path of the public key. Key transparency is not anonymous: the central server knows which keys are being requested at any given time.

In Oblix[18], the authors show that they are able to anonymize Key Transparency by storing the Merkle prefix tree in an ORAM, but their results are still impractical, with lookup times in the order of seconds. In Figure 5.7, we show that PRORAM can achieve very practical speeds for anonymous key transparency.

Chapter 6: Discussions

6.1 Intel CPU Bugs

The majority of confidentiality bugs in Intel SGX involve cache side channel attacks [20, 14, 33, 7, 35] Cache side channels rely on the ability to make inference from access patterns. Since ORAM obfuscates access patterns, it resolves these bugs.

However, ORAM is not a complete solution to SGX security bugs. Cache side channel attacks are not the only issues facing SGX. PlunderVolt [19], a recent attack which undervolts the CPU to cause unintended behavior, can leak SGX secrets without relying on access patterns. Other examples include microarchitectural data sampling attacks [31, 6], attacks which leak secrets by sampling microarchitectural buffers that are intended to be invisible to users.

6.2 Long Term Storage

6.2.1 Snapshotting

For reliability and usability, it should be possible to snapshot and recover the ORAM state. Zerotrace defines a methodology for doing so [26]. First, keep a recording of the initial state of the ORAM. Log each transaction as it occurs (e.g.

each requested block ID), and dump it to the disk in an encrypted form. Recovering a snapshot is as simple as restoring the initial state, and replaying transactions to reach the desired state.

For this scheme to be secure, it is critical that the rng used for remapping block positions is deterministic. Otherwise, the snapshotting will be susceptible to mix and match reply attacks, in which an attacker recovers several permutations of the snapshot and records the access patterns of the position map and stash.

6.2.2 Hard Disk Storage

The practical speeds achieved in our ORAM implementation are only possible when all blocks can be stored in memory: this is feasible in the case of some web servers. In the case where the cost of memory outweighs the benefit of high-performance, it can be valuable to implement an ORAM on disk.

The transference of data from memory to disk should follow the same principles as the transference from the enclave to the untrusted memory. This means that access patterns from memory to disk should be static: this can be achieved by caching entire layers of the tree on disk, or caching the entirety of some trees on disk. If the patterns are not static, then there will be timing leakage.

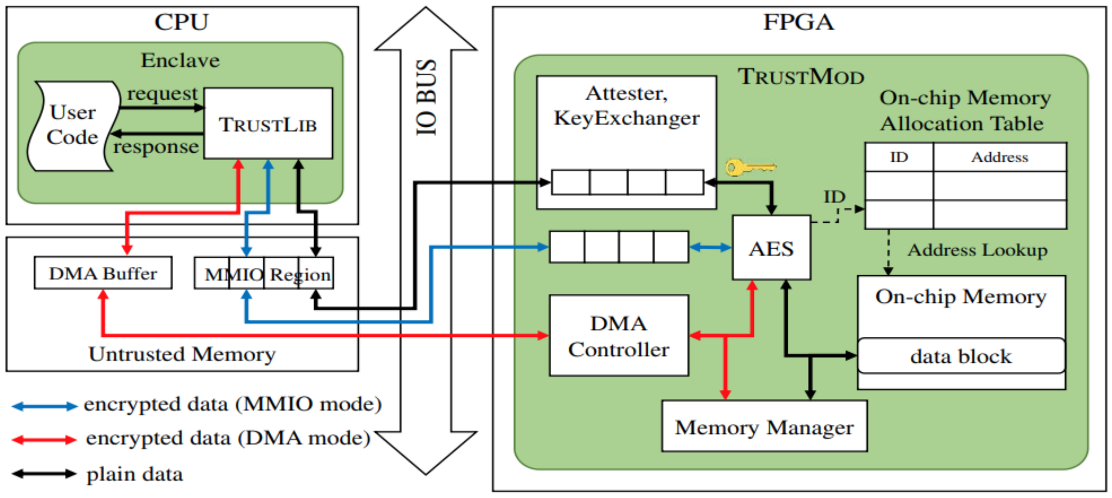


Figure 6.1: This figure from Trustore [21] shows how SGX can be combined with special purpose hardware to create performant ORAM

6.3 Future Work

6.3.1 Hardware Implementations

Figure 6.1 shows the architecture of Trustore [21]. Trustore shows a promising approach towards hardware accelerated-ORAM. The performance benefit largely hinges on having a fully secure co-processor that can operate alongside ORAM+SGX. The challenges listed in the the optimization chapter of this thesis are repeated below:

1. Intel SGX has no fully trusted storage.
2. Intel SGX's semi-trusted storage, the EPC, only holds 128MBs. The EPC is considered semi-trusted because its access patterns are leaked, but it otherwise has functional guarantees for confidentiality and integrity.

3. The position map for ORAM is often larger than the EPC.

4. The ORAM implementation can perform no data-dependent branching.

5. Blocks that are moved from the EPC to memory must be encrypted. Encryption comes at significant computational cost.

Intuitively, having a trusted secure co-processor that does not leak access patterns eliminates four out of the five listed performance challenges. However, removing SGX entirely and having a fully trusted secure co-processor removes all such limitations, and ORAM implementations such as Tiny ORAM [10] do just that.

All such implementations still require delegation from a trusted CPU if they are to be used on commodity off-the-shelf equipment. To solve this problem, future investigations could be performed on modifying Keystone [15], the RISC-V TEE with an oblivious memory controller.

6.3.2 ORAM Redesign

Rather than selecting a good ORAM scheme to implement in Intel SGX, a talented ORAM theorist could design an ORAM specifically for Intel SGX. Such an ORAM could reflect the system-level constraints of SGX. To the best of our review, there has never been an ORAM that is specifically optimized for minimal data-dependent branches and minimal trusted storage.

Chapter 7: Conclusion

Trusted Execution Environments have a promising future, but in recent years implementations of TEEs have been shown to be insecure [4, 20, 7, 31, 27, 6, 32, 23]. ORAM has been a growing interest alongside the growth of cloud computation, but it suffers heavy bandwidth overhead. The combination of TEEs and ORAM is symbiotic: ORAM patches the holes in TEEs, and TEEs are the perfect systems for running high-performance ORAM.

This thesis has been a summary of the key concepts and methods of my PRORAM project with the System's Security and Hacking Lab at Oregon State University. PRORAM represents a marked improvement in cloud privacy guarantees for users that require high assurances, and it effectively mitigates many of the bugs in Intel SGX. PRORAM also exhibits the performance necessary for practical private outsourced storage.

This investigation carries two key contributions to the scientific community:

1. We develop the state of the art practice for selecting a theoretical ORAM for use in a TEE.

2. We evaluate our analysis empirically with an optimized SGX+ORAM implementation.

Bibliography

[1] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. OBFUSCURO: A commodity obfuscation engine on intel SGX. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[3] Erick Bauman and Zhiqiang Lin. A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. Insecure until proven updated: Analyzing AMD sev's remote attestation. *CoRR*, abs/1908.11680, 2019.

[5] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, pages 4:1–4:6. ACM, 2017.

[6] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 769–784. ACM, 2019.

[7] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution. *IEEE Secur. Priv.*, 18(3):28–37, 2020.

[8] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[9] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 749–764, USA, 2014. USENIX Association.

[10] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, Dimitrios N. Serpanos, and Srinivas Devadas. A low-latency, low-area hardware oblivious RAM controller. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*, pages 215–222. IEEE Computer Society, 2015.

[11] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[12] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A. Yavuz. MOSE: practical multi-user oblivious storage via secure enclaves. In Vassil Roussev, Bhavani M. Thuraisingham, Barbara Carminati, and Murat Kantarcioglu, editors, *CODASPY '20: Tenth ACM Conference on Data and Application Security and Privacy, New Orleans, LA, USA, March 16-18, 2020*, pages 17–28. ACM, 2020.

[13] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A. Yavuz. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *Proc. Priv. Enhancing Technol.*, 2019(1):172–191, 2019.

[14] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *meltdownattack.com*, 2018.

[15] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[16] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang.

Hacking in darkness: Return-oriented programming against secure enclaves. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 523–539. USENIX Association, 2017.

[17] Sheng Lin, Chenghong Wang, Hongjia Li, Jieren Deng, Yanzhi Wang, and Caiwen Ding. ESMFL: efficient and secure models for federated learning. *CoRR*, abs/2009.01867, 2020.

[18] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[19] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[20] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel SGX. *CoRR*, abs/2006.13598, 2020.

[21] Hyunyoung Oh, Adil Ahmad, Seonghyun Park, Byoungyoung Lee, and Yunheung Paek. TRUSTORE: side-channel resistant storage for SGX using intel hybrid CPU-FPGA. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1903–1918. ACM, 2020.

[22] David Pouliot and Charles V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1341–1352, New York, NY, USA, 2016. Association for Computing Machinery.

[23] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, May 2021. Intel Bounty Reward.

[24] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[25] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring ORAM: closing the gap between small and large client storage oblivious RAM. *IACR Cryptol. ePrint Arch.*, 2014:997, 2014.

[26] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[27] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[28] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with o((logn)3) worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2011.

[29] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, October 2013.

[30] Dave (Jing) Tian, Joseph I. Choi, Grant Hernandez, Patrick Traynor, and Kevin R. B. Butler. A practical intel SGX setting for linux containers in the cloud. In Gail-Joon Ahn, Bhavani M. Thuraisingham, Murat Kantarcioglu, and Ram Krishnan, editors, *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX, USA, March 25-27, 2019*, pages 255–266. ACM, 2019.

[31] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue

in-flight data load. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 88–105. IEEE, 2019.

[32] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. *CoRR*, abs/2006.13353, 2020.

[33] Daimeng Wang, Zhiyun Qian, Nael B. Abu-Ghazaleh, and Srikanth V. Krishnamurthy. PAPP: prefetcher-aware prime and probe side-channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 62. ACM, 2019.

[34] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[35] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 719–732. USENIX Association, 2014.