Securing Cloud Computations with Oblivious Primitives from Intel SGX

by

Sajin Sasy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We are witnessing a confluence between applied cryptography and secure hardware systems in enabling secure cloud computing. On one hand, work in applied cryptography has enabled efficient, oblivious data-structures and memory primitives. On the other, secure hardware and the emergence of Intel SGX has enabled a low-overhead and mass market mechanism for isolated execution. By themselves these technologies have their disadvantages. Oblivious memory primitives carry high performance overheads, especially when run non-interactively. Intel SGX, while more efficient, suffers from numerous software-based side-channel attacks, high context switching costs, and bounded memory size.

In this work we build a new library of oblivious memory primitives, which we call ZeroTrace. ZeroTrace is designed to carefully combine state-of-art oblivious RAM techniques and SGX, while mitigating individual disadvantages of these technologies. To the best of our knowledge, ZeroTrace represents the first oblivious memory primitives running on a real secure hardware platform. ZeroTrace simultaneously enables a dramatic speedup over pure cryptography and protection from software-based side-channel attacks. The core of our design is an efficient and flexible block-level memory controller that provides oblivious execution against any active software adversary, and across asynchronous SGX enclave terminations. Performance-wise, the memory controller can service requests for 4 Byte blocks in 1.2 ms and 1 KB blocks in 3.4 ms (given a 10 GB dataset). On top of our memory controller, we evaluate Set/Dictionary/List interfaces which can all perform basic operations (e.g., get/put/insert) in 1-5 ms for a 4-8 Byte block size. ZeroTrace enables secure remote computations at substantially lower overheads than other comparable state-of-the-art techniques.

Acknowledgements

I wish to thank my supervisor, Dr. Sergey Gorbunov who patiently taught and guided me through graduate school and enabled me to be an effective security researcher. He has been an incredible mentor, who constantly motivates me to dream bigger and work towards it. I have to thank Dr. Ian Goldberg, for single handedly shifting my attention and interest towards Computer Security and Privacy, in his CS658 course in Fall 2015. I would also like to thank my thesis readers, Prof. Tamer Ozsu and Dr. Florian Kerschbaum, for their valuable feedbacks on my thesis.

I have to thank my lab group, both fellow and former "CrySPers" for all their help and guidance and also for all the little idiosyncrasies like "vortexing", that we enjoy together while more often than not learning something useful from it. Lastly, I'd like to thank my friends "the UWat Crew" for all the good times I've had and shared with them.

Dedication

To the ones I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Cloud computing is a paradigm, ever growing in popularity, that offers on-demand compute and storage resources for users. A myriad of applications today are migrating towards the cloud, few examples include machine learning, AI, data analytics, web and mobile services. Ever since its conception, cloud computing has been plagued with the tug-of-war between two contrasting pillars of functionality and security. In order to facilitate computations over data, maintaining data unencrypted and in data structures optimized for functionality on cloud servers has been the typical approach. This naturally leads to poor security guarantees. Whereas on the flip side, naively encrypting data before storing it on the cloud, strips it of any hope for efficient computation over the data. Thus ensuring both efficient functionality and strong security for applications have become a very elusive goal.

Up until recently, secure cloud computing could only be achieved through the "the holy grail" of cryptography, Fully Homomorphic Encryption – FHE [24]. FHE allows one to perform arbitrarily-complex, dynamically-chosen computations on encrypted data. As wonderful as this sounds, unfortunately FHE has severe performance and usability limitations as it introduces many orders of magnitude overheads.

An alternative path for achieving the promise of secure cloud-computing is through course-grained hardware isolation techniques (e.g., Intel TPM+TXT [33, 43, 71]). Intel TPM+TXT runs code on bare hardware, yet incurs very high switching cost in/out of TXT and low hardware utilization since the secure application owns the entire machine.

In wake of this interest in secure cloud computing, Intel recently released an instruction set extension called *Software Guard Extensions* (SGX) [15, 44, 45, 46] for their 6th

generation of processors (Skylake Lake) onwards which addresses some of the above challenges. In SGX, user-level sensitive portions of ring-3 applications can be run in one or more application containers called enclaves. To bootstrap security, attestation techniques provide the user with a proof that code and data were correctly loaded into a fresh enclave. While running, SGX uses a set of hardware mechanisms to preserve the privacy and integrity of enclave memory. Being an application container, enclaves run concurrently with other user applications and privileged code. At first glance, SGX may seem like the panacea to the secure cloud computation problem. However SGX has several of it's own shortcomings and challenges.

Challenges. An open challenge in using SGX is determining how best to map applications to enclave(s), that gives the best trade-off in trusted computing base (TCB) size, performance and code isolation. A common approach, natively supported by the Intel SGX, is to partition an application into trusted and untrusted code [63, 89]. A developer would manually define which parts of an application should run in one (or multiple) enclaves and define a communication method between them. In a good design, bugs in one component may be isolated from the rest of the system, limiting their affects. While theoretically this approach may lead to a very fine-grained application isolation, it raises the question of where to partition a complex application. Alternatively, a number of works study how to load unmodified applications into enclaves [3, 6, 32, 72]. To run full applications, these approaches load parts of an OS (e.g., libc, pthreads, container code) into the enclave alongside the application. This removes the need to decide how to re-architect the application, but introduces a large TCB: a bug in the library OS or the application can cause corruption anywhere else in the application.

Second, as it time-shares the CPU with other applications and privileged code, SGX can leak sensitive data over covert channels (e.g., cache/branch predictor sharing, page fault pattern). Researchers have shown how these indirect leakages can be devastating – from leaking key material [57] to the outlines of sensitive images [82]. Prior work proposed to transform the whole program to an *oblivious form* [51, 47]. The downside to obliviousness is performance overhead: conceptually, the program is converted to straight-line code. To exacerbate the overhead, prior work has only studied obliviousness as applied to the whole program – analogous to porting the entire application to a single enclave as described above.

Our Approach. We address these challenges by designing and implementing ZeroTrace – a library enabling applications to be built out of fine-grained, building-blocks at the

application's *data-structure* interface boundary. As part of this research, *we implement and evaluate the first oblivious memory controller running on a real secure hardware platform.*

Partitioning applications at the oblivious data-structure boundary hits a sweet spot for several reasons. First, the data-structure interface is narrow. This makes it easier to sanitize requests and responses from application to data-structure, improving intra-application security. Second, the data-structure interface is re-usable across many applications. A service provider can pre-package data-structure backends as pre-certified blocks with a common interface, enabling application developers to build complex applications from known-good pieces. Further, there is a rich literature in the security community on how to efficiently achieve various security properties when working with various data-structures [9, 26, 75, 87]. These works can be dropped into our system as different backend implementations, which gives clients the ability to hot-swap between implementations, depending on the application's security requirements.

Our system's core component is a fully-implemented SGX-based secure memory controller that exposes a block $\mathsf{read}(\mathsf{addr})$ and $\mathsf{write}(\mathsf{addr}, \mathsf{data})$ interface to applications. This memory controller runs in software, partly in an SGX enclave and partly in outside ring-3 support logic. The controller's primary design consideration is *flexibility*: we wish for the core controller to be usable across a variety of threat and usage models. At the highest level of security, the controller hides which operations are issued by the user and the arguments issued to those commands. That is, it runs *obliviously* [51, 49, 47, 26]. The module can be parameterized to defend against several types of adversaries, where the highest level of security provides obliviousness (privacy) and integrity (authenticity and freshness) guarantees against *arbitrary software-based* adversaries. The core can be parameterized to defend against a subset of these threats, depending on the context. To maximize usability, our controller exposes a low-level and generic 'frontend' secure channel-like interface to applications. The controller's backend interacts directly with untrusted, available DRAM and/or HDD/SSD storage, in a fashion transparent to the application.

Building an efficient memory controller in SGX is non-trivial, presenting security and performance challenges, due to the nature of SGX.

- Despite isolating enclave virtual memory from direct inspection, SGX can leak sensitive data over covert channels (e.g., cache/branch predictor sharing, page fault pattern). We employ additional mechanisms (e.g., [47, 49, 51]) to prevent these leakages.

- SGX enclaves do not support direct I/O to disk. To support disk backend storage, we partition the controller between trusted and un-trusted zones in a secure fashion.

- SGX does not support persistant integrity across boots, and risks memory controller data corruption on sudden/un-expected shutdowns. We develop a novel protocol to make the core memory controller fault tolerant: allowing the controller to quickly and securely recover from such a shutdown or failure (even in the event of partial data loss).

- On the performance front, the whole system design requires a careful balance of resources between enclave memory, untrusted DRAM and untrusted disk(s). We propose optimizations to efficiently make use of these different resources in a way that preserves the module's security guarantee.

Using our core memory controller as a building-block, we implement a library of data-structures that can interface directly with applications. We evaluate several common data-structures including arrays, sets, dictionary and lists. Building on top of enclaves that have flexible client-facing interfaces brings new advantages. Multiple clients can seamlessly share the same data-structure, with software-controlled access policies depending on the trust between those applications. Clients can also attach remotely to the data-structure, creating novel distributed systems that create interesting improvements to related research directions. For example, by extending the TCB to Intel SGX, we reduce the client-server bandwidth of a traditional oblivious file server *by over an order of magnitude.*

1.2 Related Works

Oblivious RAMs and Secure Hardware. Research in ORAM began with the seminal work by Goldreich and Ostrovsky [26], and has culminated in practical constructions with logarithmic bandwidth overhead [52, 68, 74] as we discuss in Section 3.3.

In the context of ORAM, our work moves the ORAM controller close to storage, exploiting the fact that ORAM bandwidth overhead occurs *between ORAM controller and storage* and not between client and ORAM controller. This idea has been explored previously by combining homomorphic encryption with ORAM by Onion ORAM [17], and by the ORAM-based systems Oblivistore [65] and ObliviAd [4] (which assume hypothetical secure hardware). The latter two works have a weaker threat model than our model, since our goal is to protect against all remote software attacks, whereas the latter two focus only on hiding ORAM protocol-level access patterns.

Another similar direction of research is secure hardware projects such as Phantom [41], Aegis [69] and Ascend [23]. Phantom is a secure processor that obfuscates it's memory access patterns by using PathORAM intrinsically for all its memory accesses. Aegis is aimed at incorporating privacy and integrity guarantees for physical attacks (in addition to software attacks) against the processor. (It makes use of PUF - Physically Unclonable Functions to create Physical Random Functions) Ascend is a secure coprocessor[1] that aims at achieving secure computations for a cloud server against semi-honest adversary. It is designed to perform oblivious computations to which end it obfuscates its instruction execution such that it appears to spend the same time/energy/effort for the execution of each instruction independent of the underlying instruction.

Phantom achieves similar security goals as that of ZeroTrace, however there are several differences between our project and such secure hardware projects. First, since these projects rely on custom hardware that are uncommon (typically unavailable) commercially, deployability of these projects are dubious at best. Intel SGX (and therefore ZeroTrace) is commercially available and already present on all Intel processors from Skylake series onwards. Secondly, these secure processors are innately tied to providing oblivious accesses to just the memory/DRAM, however ZeroTrace is extremely flexible with respect to the underlying storage support. Additionally, not every piece of data needs to have the same security properties. Exploiting this security flexibility, allows applications to trade their higher level of security for performance efficiency through ZeroTrace.

[1] An additional processor that sits alongside the main server, for performing secure computation.

Systems. A number of systems investigate the question of protecting applications running in enclaves. Raccoon [51] provides oblivious program execution via an integration with an ORAM and control-flow obfuscation techniques. In particular, they obfuscate programs by ensuring that all possible branches are executed, regardless of the input data. This approach is conceptually differs from ours since we provide oblivious building blocks for sensitive data with strict underlying security guarantees. Also, because of how the control-flow techniques are enforced in Racoon, it assumes a trusted operating system (Section 3, [51]). In our design, obliviousness is guaranteed even when an adversary compromises the entire software stack including the OS. Finally, while Raccoon is designed to run on an Intel SGX-enabled processor, the architectural limitations of SGX are not taken into consideration in their design. Their results are based on emulations of Intel SGX, and hence have results that our work disproves, since we implement an actual system on real SGX enabled hardware.

GhostRider [39] proposed a software-hardware hybrid approach to achieve program obliviousness. It is a set of compiler and hardware modifications that enables execution of an ORAM controller inside an FPGA card used for sensitive data accesses. Their work offers only a "conceptual" approach to the problem. In particular, they assume "unbounded resources, and no caching" and do not target any modern processor. In contrast, the focus of this work is to design a real-world system capable of running on a widely available Intel CPU architecture.

Opaque [89] is a secure Spark database system where components of the database server are run in SGX enclaves. Opaque is complementary to ZeroTrace: their focus is to support oblivious queries for a database system; our focus is to support arbitrary oblivious read/write operations. Each system is superior in supporting its chosen task.

Attacks and Defenses. The primary attack vectors against SGX in the literature stem from the fact that enclaves share physical resources with other applications and interact with the OS to perform syscalls and paging. Using a shared resource, typically cache usage [29, 50, 70, 84, 88, 34, 40, 73] can be exploited by an adversary to reveal fine-grain details about program execution. Similarly, a malicious OS can monitor application page fault behavior to learn program memory access patterns [82, 62]. Recently, we have even seen attacks that infer fine-grained control flow from code executing in an SGX enclave via branch shadowing [38].

Most cache attacks against Intel SGX follows a generic blue print. The setup typically constitutes a malicious attacker process and a victim enclave process that are executed on the same CPU. The attacker process sets the cache into a "clean" state by filling it up

with it's contents, it then allows the victim enclave to execute causing it to replace cache contents or "dirty" cache locations. For the attack, one periodically swaps out the victim enclave for the attacker process, infers the cache usage pattern of the victim enclave and refreshes the cache state.

T-SGX is a system that helps to protect against controlled-channel attacks within enclaves [61]. However, T-SGX is based on identifying if an enclave is being swapped/interrupted more frequently than expected and forcefully shutting itself down in such an event to prevent leaking sensitive information. However, Brasser et al.[11], demonstrated how one can perform these cache attacks with minimal enclave exits/swaps, bypassing defense mechanisms such as T-SGX. Hence in SGX-based systems, there is currently an arms race underway between defenses (e.g., T-SGX[61], SGXbounds[37], Deja Vu[14]) and new attacks (e.g., Brasser et al.[11], Lee et al.[38]) related to shared resource usage.

ZeroTrace protects against all such shared resource and page fault-related attacks by converting the program to an oblivious representation.

1.3 Our Contributions

This thesis makes the following contributions

1. We design and build an oblivious memory controller from Intel SGX. To the best of our knowledge, the core memory controller (the bulk of our system) is the first oblivious memory controller implemented on a real secure hardware platform.

2. We design and implement ZeroTrace, an application library for serving data-structures obliviously in an SGX environment that runs on top of our memory controller.

3. We evaluate system performance in two settings: as a remote file server and as a plug-and-play for data-structures. It can also make oblivious read and write calls to 4 B/1 KB memory locations on a 10 GB dataset in 1.2/3.4 ms. In the plug-and-play setting, ZeroTrace can make oblivious read and write calls at 8 B granularity on an 80 MB array in 1.2 ms.

4. We model how we envision secure cloud computation using ZeroTrace and compare and contrast it with existential models.

1.4 Overview of the Thesis

- In Chapter 2, we describe our usage model and security model.
- Chapter 3 covers all the background knowledge required for explaining ZeroTrace. Here, we explain Intel SGX, focusing on the aspects most relevant to ZeroTrace. We also give a brief introduction to Oblivious RAM(ORAM) schemas, emphasizing on PathORAM and CircuitORAM.
- In Chapter 4, we give details on our architecture for ZeroTrace; including the instantiation process, client and server components, optimizations, persistant integrity technique and security analysis.
- Chapter 5 describes our prototype implementation and evaluation.
- Finally, Chapter 6 concludes by discussing existential secure computation models and how ZeroTrace compares with them for secure computation.

Chapter 2

Computation Model

2.1 Usage Model

We consider a setting where a computationally weak client wishes to outsource storage or computation to an untrusted remote server that supports Intel's Software Guard Extension (SGX). As secure hardware extensions such as SGX reach the market, we anticipate this setting will become a common way to implement many real world applications such as image/movie/document storage and computation outsourcing. The cloud can be any standard public cloud such as Amazon AWS, Microsoft Azure or Google cloud, and the client can be any mobile or local device. In fact, Microsoft Azure recently announced support for "confidential computing" by making SGX-enabled servers available via their cloud service [1].

As introduced in Chapter 1, our proposal consists of stand-alone enclaves that implement secure memory services. We envision future applications being constructed from these (and similar) plug-and-play services. We now describe this general scenario in more detail. Afterwards, we show how a special case of this scenario improves performance in a related branch of research.

Plug-and-play memory protection for outsourced computation. We envision an emerging scenario where client applications (e.g., a database server), which run in an SGX enclave(s), connect to other enclaves to implement secure memory and data-structure services. In an example deployment, calling a memory service enclave is hidden behind a function call, which is dynamically linked (connected to another enclave via a secure

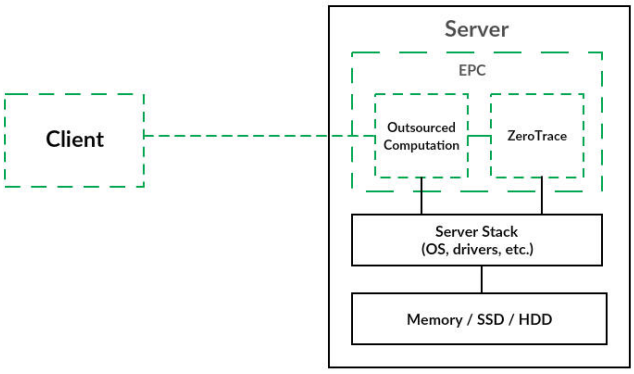

Figure 2.1: Plug-and-play memory controller model

channel) at runtime. What "backend" memory service our system supports can be changed depending on the application's needs. For example, our core memory controller currently supports an ORAM backend. Without changing the application-side interface, this backend can be transparently changed to support a different ORAM, different security level for memory protection (e.g., plain encryption) or different security primitive entirely (e.g., a proof of retrievability [9]). A similar argument goes for memory services exposing a data-structure interface. For example, Wang et al. [75] proposed a linked-list optimized for use as an iterator, while another implementation can be optimized for insertion.

A reasonable question is: why break these services into separate enclaves, as opposed to statically linking them into the main application? Our design has several advantages. First, breaking an application into modules eases verification. SGX provides enclave memory isolation. Thus, verifying correct operation reduces to sanitizing the module interface (a similar philosophy is used in Google's NaCl [85]). Data structures and memory controllers naturally have narrow interfaces (compared to more general interfaces, such as POSIX [63]), easing this verification. Second, breaking applications into modules eases patching. Upgraded memory services can be re-certified and re-attached piecemeal, without requiring the vendor to re-compile and the client to re-attest the entire application. Third, inter-communicating between enclaves gives flexibility in deployment, as shown in the next paragraph.

Remote block data storage. Suppose a client device wishes to store blocks of data (e.g., files) on the remote server (e.g., Amazon S3). To achieve obliviousness, the standard

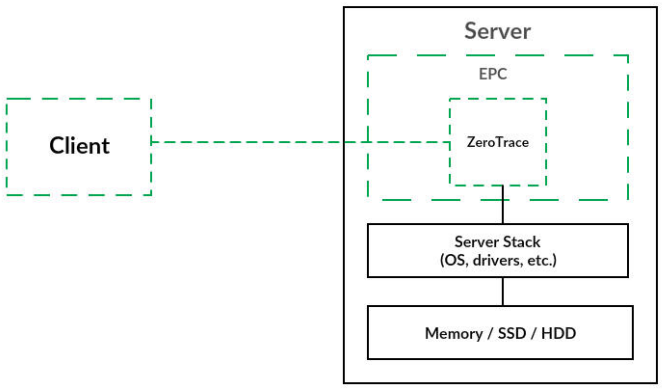

Figure 2.2: Remote Data Storage model

approach is for the client to use an Oblivious RAM protocol where the client runs the ORAM controller locally [66, 77]. The ORAM controller interacts over the network with the server, which acts as a disk. While benefitting from not trusting the server, these solutions immediately incur an at-least logarithmic bandwidth blowup over the network (e.g., WAN) due to the protocol between ORAM controller and server. As a special case of the first setting (above), the core memory controller can serve as the ORAM controller, from the oblivious remote file server setting, now hosted on the server side. As our architecture can protect side-channel leakages introduced from the SGX architecture, the only change to security is we now trust the SGX mechanism. The advantage is bandwidth savings: this deployment improves client communication over the network **by over an order of magnitude**. Our scheme still incurs logarithmic bandwidth blowup between the enclave code and server disks, but this is dwarfed by the cost to send data over the network.

2.2 Threat Model

In our setting, memory controller logic (e.g., the ORAM controller) and higher-level interfaces are implemented in software run on the server. The server hosts SGX and a regular software stack outside of SGX. The client and SGX mechanism are trusted; memory controller logic is assumed to be implemented correctly. We do not trust any component on the server beyond SGX (e.g., the software stack, disks, the connection between client

and server, other hardware components besides the processor hosting SGX). Per the usual SGX threat model, we assume the OS is compromised and may run concurrently on the same hardware as the memory controller. By trusting the SGX mechanism, we trust the processor manufacturer (e.g., Intel).

Security goals. Our highest supported level of security, and our focus in this thesis, is for the SGX enclave running the memory controller to operate *obliviously* in the presence of any active (malicious), software-based adversary. In this case, the memory controller must run an ORAM protocol over untrusted storage. We default to this level of security because a known limitation of SGX is its software-based side-channel leakages (Section 3.2.1), which are dealt with via oblivious execution. (Related work calls these *digital side-channels* [51].) Obliviousness means the adversary only learns the number of requests made between client and memory controller; i.e., not any information contained in those requests. We are interested in preserving privacy and integrity of requests. The server may deviate from the protocol, in an attempt to learn about the client's requests or to tamper with the result. Our system's threat surface is broken into several parts:

Security of memory. First, the memory accesses made by the SGX enclave to external memory. These are completely exposed to the server and must preserve privacy and integrity of the underlying data. These accesses inherit the security of the underlying memory protection (e.g., ORAM), which we detail in Section 3.3.1.

Security of enclave execution. Second, the SGX enclave's execution as it is orchestrating accesses to external memory. At a high level, SGX only provides privacy/integrity guarantees for enclave virtual memory. Running ORAM controller code in an enclave does not, by itself, ensure obliviousness. External server software (which shares the hardware with the enclave) can still monitor any interactions the enclave makes with the outside world (e.g., syscalls, etc.), how the enclave uses shared processor resources such as cache [11, 57] and how/when the enclave suffers page faults [82]. Our system has mechanisms to preserve privacy and integrity despite the above vulnerabilities. We formalize this security guarantee in Section 3.1 and map SGX to these definitions in Section 3.2.

Security across enclave termination. Third, recovery and security given enclave termination. An important caveat of SGX is that the OS can terminate enclave execution at any time. This has been shown to create avenues for replay attacks [42], and

risks irreverable data-loss. We develop novel protocols in Section 4.4 to make the ORAM+enclave system fault tolerant and secure against arbitrary enclave terminations.

Security non-goals. We do not defend against hardware attacks (e.g., power analysis[35] or EM emissions [58]), compromised manufacturing (e.g., hardware trojans [83]) or denial of service attacks.

Chapter 3

Preliminaries

In this chapter, we first formalize this notion of oblivious enclave execution that we desire, we then establish some basic understanding of Intel SGX, the security properties it offers and its limitations .

3.1 Oblivious Enclave Execution

We now formalize oblivious execution for enclaves that we set out to achieve in our system. We first give a general definition for enclave-based trusted/oblivious execution, that defines the client API, security guarantees, and where privacy leakages can occur. In the next section, we describe exactly what privacy and integrity threats are present in Intel SGX in particular, and the challenges in protecting them.

To help us formalize the definition, we define a pair of algorithms Load and $\mathsf{Execute}$, that are required by a client to load a program into an enclave, and execute it with a given input.

$\mathsf{Load}(\mathsf{P}) \rightarrow (\mathsf{E}_\mathsf{P}, \phi)$. The load function takes a program P, and produces an enclave E_P, loaded with P along with a proof ϕ, which the client can use to verify that the enclave did load the program P.

$\mathsf{Execute}(\mathsf{E}_\mathsf{P}, \mathsf{in}) \rightarrow (\mathsf{out}, \psi)$. The execute function, given an enclave loaded with a program P, feeds the enclave with an input in, to produce a tuple constituting of the output out,

and ψ which the client can use to verify that the output out was produced by the enclave $\mathsf{E_P}$ executing with input in.

Execution also produces $\mathsf{trace}_{(\mathsf{E_P},\mathsf{in})}$, which captures the execution trace induced by running the enclave $\mathsf{E_P}$ with the input in which is visible to the server. This $\mathsf{trace}_{(\mathsf{E_P},\mathsf{in})}$ contains all the powerful side channel artifacts that the adversarial server can view, such as cache usage, etc. These are discussed in detail in the case of Intel SGX in Section 3.2.1, below.

Security. When a program P is loaded in an enclave, and a set of inputs $\overrightarrow{\mathsf{y}} := (\mathsf{in_M}, ..., \mathsf{in_1})$ are executed by this enclave, it results in an adversarial view $\mathsf{V}(\overrightarrow{\mathsf{y}}) := (\mathsf{trace}_{(\mathsf{E_P},\mathsf{in_M})}, ..., \mathsf{trace}_{(\mathsf{E_P},\mathsf{in_1})})$. We say that an enclave execution is oblivious, if given two sets of inputs $\overrightarrow{\mathsf{y}}$ and $\overrightarrow{\mathsf{z}}$, their adversarial views $\mathsf{V}(\overrightarrow{\mathsf{y}})$ and $\mathsf{V}(\overrightarrow{\mathsf{z}})$ are computationally indistinguishable to anyone but the client.

3.2 Intel SGX

In this section we give a brief introduction to Intel Software Guard Extensions (SGX) and highlight aspects relevant to $\mathsf{ZeroTrace}$. (See [2, 15] for more details on SGX.) Intel SGX is a set of new x86 instructions that enable code isolation within virtual containers called enclaves. In the SGX architecture, developers are responsible for partitioning the application into enclave code and untrusted code, and to define an appropriate I/O communications interface between them.

In SGX, security is bootstrapped from an underlying trusted processor, not trust in a remote software stack. To this end, the processor is fused with cryptographic keys in its manufacturing phase. These keys can only be accessed by the processor, and is used to encrypt the contents of a subsets of DRAM, referred as Processor Reserved Memory (PRM) which gets set aside securely at boot time. The PRM, is commonly referred to as Enclave Page Cache (EPC) in SGX literature. All pages in the EPC are encrypted with the processor owned keys. A subset of these EPC pages called as Version Array(VA) pages are set aside to maintain freshness and integrity guarantees of these EPC pages. These VA pages stores a merkle tree of hashes for the EPC pages. Hence every EPC access undergoes a decryption as well as an integrity check against the merkle tree, to ensure that EPC pages are confidential and not tampered with.

We now describe how Intel SGX implements the $\mathsf{Load}(\mathsf{P})$ and $\mathsf{Execute}(\mathsf{E_P}, \mathsf{in})$ functions from the previous section.

$\mathsf{Load}(\mathsf{P}) \rightarrow (\mathsf{E}_\mathsf{P}, \phi)$**.** A client receives a proof ϕ that its intended program P (and initial data) has been loaded into an enclave via an attestation procedure. Code loaded into enclaves is measured by SGX during initialization (using SHA-256) and signed with respect to public parameters. The client can verify the measurement/signature pair to attest that the intended program was loaded via the Intel Attestation Service.

$\mathsf{Execute}(\mathsf{E}_\mathsf{P}, \mathsf{in}) \rightarrow (\mathsf{out}, \psi)$**.** SGX protects enclave program execution by isolating enclave code and data in the EPC. Cache lines read into the processor cache from the EPC are isolated from non-enclave read/writes via hardware paging mechanisms, and encrypted/integrity-checked at the processor boundary as mentioned above. Thus, data in the EPC is protected (privacy and integrity-wise) against certain physical attacks (e.g., bus snooping), the operating system (direct inspection of pages, DMA), and the hypervisor.

Paging. In Intel SGX, the EPC has limited capacity. To support applications with large working sets, the OS performs paging to move pages in and out of the EPC on demand. Hardware mechanisms in SGX ensure that all pages swapped in/out of the EPC are integrity checked and encrypted before being handed to the OS. Thus, the OS learns only that a page with a public address needed to be swapped, not the data in the page. Special pages controlled by SGX (called VA pages) implement an integrity tree over swapped pages. In the event the system is shutdown, the VA pages and (consequently) enclave data pages are lost.

Enclave I/O. It is the developer's responsibility to partition applications into trusted and untrusted parts and to define a communication interface between them. The literature has made several proposals for a standard interface, e.g., a POSIX interface [63].

3.2.1 Security Challenges in Intel SGX

We now detail aspects of Intel SGX that present security challenges for and motivate the design of ZeroTrace.

Software side channels. Although SGX prevents an adversary from directly inspecting/tampering with the contents of the EPC, it does not protect against multiple software-based side channels. In particular, SGX enclaves share hardware resources with untrusted applications and delegate EPC paging to the OS. Correspondingly, the literature has

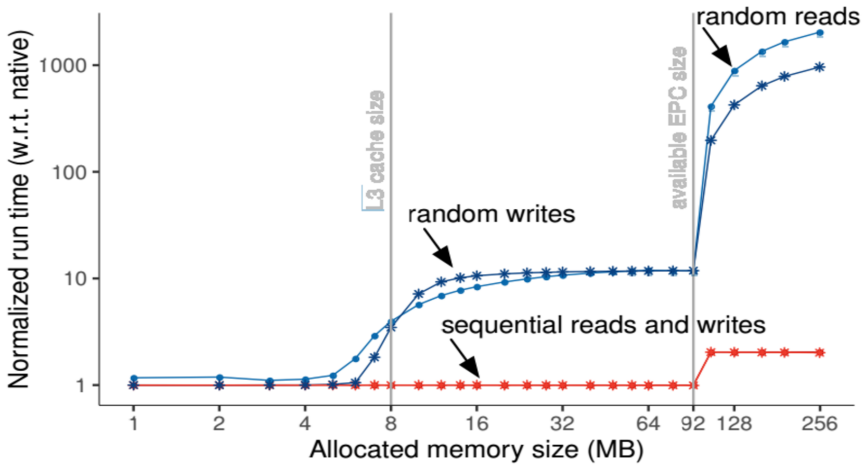

Figure 3.1: Normalized overhead of memory accesses with SGX enclaves. (Figure 3 from Scone by Arnautov et al.[3].)

demonstrated attacks that extract sensitive data through hardware resource pressure (e.g., cache [11, 57] and branch predictor [38]) and the application's page-level access pattern [82].

EPC scope. Since the integrity verification tree for EPC pages is located in the EPC itself (in VA pages), SGX does not support integrity (with freshness) guarantees in the event of a system shutdown [42]. More generally, SGX provides no privacy/integrity guarantees for any memory beyond the EPC (e.g., non-volitile disk). Ensuring persistent integrity for data and privacy/integrity for non-volitile data is delegated to the user/application level.

No direct IO/syscalls. Code executing within an enclave operates in ring-3 user space and is not allowed to perform direct IO (e.g., disk, network) and system calls. If an enclave has to make use of either, then it must delegate it to untrusted code running outside of the enclave.

3.2.2 Additional Challenges In Enclave Design

We now summarize additional properties of Intel SGX (1.0) that make designing prevention methods against the above issues challenging.

EPC limit. Currently, the size of EPC is physically upper bounded by 128 MB by the processor. Around 30 MB of EPC is used for bookkeeping, leaving around 95 MB of usable memory. As mentioned above, EPC paging alleviates this problem but reveals page-level access patterns. However EPC paging is expensive and can cost between 3x and 1000x depending on the underlying page access pattern as shown in Fig 3.1.

Context switching. At any time, the OS controls when enclave code starts and stops running. Each switch incurs a large performance overhead – the processor must save the state needed to resume execution and clear registers to prevent information leakages. Further, it is difficult to achieve persistant system integrity if the enclave can be terminated/swapped at any point in its execution.

3.3 Background for ORAM

ORAM or Oblivious RAM (Random Access Memory) was introduced by Goldreich and Ostrovsky[26] in 1996 in their theoretical treatment of software protection. Their work considers an adversary that can make non-trivial inferences from the *memory access patterns* of an encrypted program. To this end, they emphasize that memory access patterns should be independent of the executed program. In their seminal work, Goldreich and Ostrovsky showed that every model of computation can be transformed into an equivalent oblivious computation at the cost of a slowdown in the running time of the oblivious machine and they provided a lower bound for this transformation.

This problem of memory access patterns of a program are analogous to that of file access patterns in a file storage system. Hence ORAM's extend it's utility in oblivious file storage systems as well [8, 66, 76]. Obliviousness is more important than ever today, since over the last few years, there are several privacy and security breaches that stem from access pattern leakages [13, 34, 40, 48, 73, 82, 84, 88].

Informally the goals of an ORAM schema are to not leak any information about

- Which data is being accessed
- How old is the data accessed (last access)
- Whether the same data is being accessed (linkability)
- Whether the access is a read or a write

Over the past two decades, we have seen several innovative constructions for ORAM's. In this chapter, we first state the security model for ORAMs from literature, followed by a brief literature of ORAM schemas highlighting their differences, and then proceed to explain the construction for Path RAM [68] and Circuit ORAM [74] which are the ORAM schemas underlying ZeroTrace.

3.3.1 Security model

Correctness. We say that an ORAM construction is correct if it returns, on input $\overrightarrow{y}$, data that is consistent with $\overrightarrow{y}$ with probability $\geq$ 1 - negl($|\overrightarrow{y}|$), i.e. the ORAM may fail with probability negl($|\overrightarrow{y}|$).

Security. Let

$$\overrightarrow{\mathsf{y}} := ((\mathsf{op_M}, \mathsf{a_M}, \mathsf{data_M}), ..., (\mathsf{op_1}, \mathsf{a_1}, \mathsf{data_1}))$$

denote a data request sequence of length M where each $\mathsf{op_i}$ denotes a $\mathsf{read}(\mathsf{a_i})$ or a $\mathsf{write}(\mathsf{a_i})$ operation. Specifically, $\mathsf{a_i}$ denotes the identifier of the block being read or written, and $\mathsf{data_i}$ represents the data being written. In this notation, index $\mathsf{1}$ corresponds to the most recent load/store and index M corresponds to the oldest load/store operation. Let $\mathsf{ORAM}(\overrightarrow{\mathsf{y}})$ denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests $\overrightarrow{\mathsf{y}}$. An ORAM construction is said to be secure if for any two data request sequences $\overrightarrow{\mathsf{y}}$ and $\overrightarrow{\mathsf{z}}$ of the same length, their access patterns $\mathsf{ORAM}(\overrightarrow{\mathsf{y}})$ and $\mathsf{ORAM}(\overrightarrow{\mathsf{z}})$ are computationally indistinguishable to anyone but the client.

3.3.2 Choices of ORAM

The efficiency of Oblivious RAM is measured by three main parameters, the amount of local (client) storage, the amount of server (remote) storage and the overhead of reading/writing an element. Over the last two decades there have been several refinements and innovative constructions for ORAMs. The constructions can be broadly classified into two types, namely *hierarchical* and *tree-based*.

Hierarchial Constructions. Goldreich and Ostrovsky's work [26] (popularly known as GO-RAM) provided the first hierarchical construction. These constructions involve a hierarchy of buffers, each buffer contains a number of buckets that are geometrically increasing as you go down the hierarchy. A bucket contains slots for $\log N$ blocks, where N is the total input blocks. The construction treats each of these buffers as a hash table, and allots a hash function to it to determine which bucket to place an incoming block to. In order to place a block in a buffer, the hash function for that buffer is computed for the block in question, and it is inserted into the corresponding bucket. At the beginning, all blocks are inserted into the deepest level buffer.

In order to access a block, one searches for the block in each of the buffers by scanning[1] the bucket dictated by the hash function for the requested block. Once the block is found in a buffer, the algorithm proceeds to make dummy accesses for the block in the rest of the buffers. At the end of a search the requested block is inserted back into the first buffer.

Eventually, to avoid overflow of blocks in the first buffer, all blocks from it are moved to the next buffer. Thus, after number of accesses corresponding to the buffer size of the

[1]By scanning, we mean accessing all the actual blocks in the bucket

level i , the contents of the buffer i gets moved to the next buffer $\mathsf{i}+1$. This move involves "obliviously sorting" [2] the contents of the two buffers and inserting them into the buffer $\mathsf{i}+1$ with a new hash function. The original GO-RAM construction featured a sequence of buffers which grew in an unbounded fashion with access requests, i.e. once the deepest level buffer was full a new buffer is allocated as the deepest level buffer. We make a quick note here that all blocks are always encrypted and authenticated using a randomized encryption and authentication schema, and that this is an orthogonal problem for any ORAM scheme.

Williams and Sion [76] improved upon GO-RAM by introducing a method to perform oblivious sorting by using $\mathsf{O}\sqrt{N}$ local memory to reduce the access overhead to $\mathsf{O}(\log^2(N))$. Williams et al. [78] followed up their work in 2008, with a construction that reduced the access overhead to $\mathsf{O}(\log N \log\log N)$ with use of Bloom filters. However Kushilevitz et al. [36] showed that this construction allowed an adversary to distinguish access patterns based on hash overflows. In their work, Kushilevitz et al. [36] produced the state of the art hierarchical construction which extended the construction by Goodrich and Mitzendmacher [27] with a shared stash to handle the hash overflows. Their construction splits the server storage into two, the upper half of buffers are filled using bucket hashing identical to that of GO-RAM [26], and the lower half of buffers are filled with cuckoo hashing. Their work is optimized for bandwidth, while still maintaining minimal client storage.

Tree based constructions. Damgard et al. [16] in 2011 introduced an interesting ORAM construction that removed the need for random oracles from the GO-RAM construction by using binary trees. Following that Shi et al. [59] introduced the first tree based ORAM construction. It was followed up with several other notable constructions [22, 52, 68, 74] that made use of the tree framework. These schemas typically have the server-side ORAM storage organized into a binary tree of nodes that are called buckets. Most notable among them being Path ORAM [68], which we elaborate upon in Section 3.3.3. In these tree based constructions, the client is associated with a small storage which holds two local data structures, a position map which maps every block to a leaf label on the storage tree and a stash for overflown blocks that are to be held at the client side. Fetching a block corresponds to selecting a path from the root of this binary tree to the leaf node that it is mapped to in the position map.

These tree based constructions typically differ only in their eviction strategies. In Path ORAM the eviction strategy is to refill the read path that was accessed, with blocks from

[2] GO-RAM construction uses the Batcher Sorting network [5] with a complexity of $\mathsf{O}(N \log^2 N)$

the stash, pushing blocks as closer to the leaf as possible. However in Ring ORAM [52] and Circuit ORAM [74] the eviction paths are independent of the read path.

In Ring ORAM the focus is to reduce the online amortized bandwidth. To this end, instead of reading all blocks in a path, for an access Ring ORAM reads a block per bucket on the path to the leaf label. In order to facilitate this single block read per bucket, it maintains metadata within a bucket for the permutation of blocks in it. This also implies that Ring ORAM can use the XOR trick[3] from Burst ORAM [64], further reducing the bandwidth.

Bucket ORAM [22] tries to merge the best of both worlds (tree and hierarchical). Although it features level-rebuild shuffling like hierarchical ORAM's, it avoids the oblivious sort of all blocks within a level, instead the client works on a subset of the buckets in the level. By maintaining the level-rebuild shuffling, Bucket ORAM can take advantage of an additively homomorphic encryption technique from Onion ORAM [17] to achieve constant bandwidth blowup.

CircuitORAM [74] was designed with the intent of optimizing circuit complexity for efficient instantiations of Multi-Party Computation (MPC) protocols. It does so by simplifying the eviction circuit. Circuit ORAM reads two eviction paths for each access, and makes two metadata scans of the eviction paths. It uses this foresight from the metadata scans to push blocks deeper into the eviction paths in a single scan.

Table 3.3.2, gives breakdown of efficiencies of relevant ORAM constructions from literature. Among the plethora of ORAM constructions, we note that hierarchical constructions are inefficient for our purpose. There are several reasons for this claim.

- All hierarchical schemas involve periodical reshuffling of large buffers (for large values of N). In our setting this would involve using up a large number of EPC pages, and the additional overheads that stem from that.
- For every access, hierarchical schemas typically involve accessing a buffer from a bucket based on the evaluation of a PRF. Within the SGX setting, this evaluation of PRF can lead to side channel attacks, and hence it become crucial to use PRF's with an oblivious implementation.
- Tree based schemas offer better security than hierarchical since the security guarantee arises from statistical security (distribution of data over paths to randomly sampled

[3]The XOR trick - instead of sending $\mathsf{O}(\log N)$ blocks on the block, one can simply XOR the dummy blocks on the path with the real block. The client can then generate the dummy blocks locally and XOR them with the obtained block to retrieve the underlying real block.

ORAM Scheme	Client Storage (# of blocks)	Server Storage (# of blocks)	Read/Write Bandwidth (# of blocks of size B)
GO-RAM [26]	$O(1)$	$O(N \log N)$	$O(\log^3 N)$
Kushilevitz et al. [36]	$O(1)$	$O(N)$	$O(\log^2 N / \log \log N)$
SSS ORAM [67]	$O(\sqrt{N})$	$O(N)$	$O(\log N)$
Path ORAM [68]	$O(\log N)$	$O(N)$	$O(\log^2 N)$
Bucket ORAM [22]	$O(\log N)$	$O(N)$	$O(\log^2 N)$
Ring ORAM [52]	$O(\log N)$	$O(N)$	$O(\log^2 N)$
Circuit ORAM [74]	$O(1)$	$O(N)$	$O(\log^2 N)$

Table 3.1: Asymptotic performance of different ORAM Constructions, for simplicity of representation the table uses a small block size $B = \Omega \log^2 N$. The complexities of Path ORAM, Ring ORAM, Bucket ORAM and Circuit ORAM are discussed for their recursive versions.

leafs) as opposed to the computational security guarantee of PRF evaluations in hierarchical constructions [4]

- Hierarchical schemas also have a worst case complexity that is significantly worse than their amortised response time. This stems from the fact that they have to periodically reshuffle buffers, which is often a drawback of using hierarchical schemas in application settings.

We are still faced with a selection of tree based ORAM schemas. One thing to note is that ORAM bandwidth to untrusted storage and ORAM controller trusted storage are inversely proportional [67, 68, 74].

Among the aforementioned constructions, BucketORAM and RingORAM face the same problem of evaluating PRF's within the enclave (as mentioned earlier) for permuting and fetching the blocks within a bucket. In our setting, the SGX and obliviousness requirements take a performance penalty when using larger controller storage (due to EPC evictions [42] and the cost of running oblivious programs; see Section 3.2.2). Hence, recursive variants of these constructions are a better match for our setting. Additionally, passing data in and

[4] This is without accounting for the computational security of encryption and authentication schemas.

out of an enclave introduces delays due to the underlying context switch (Section 3.2.2), hence Circuit ORAM would introduce more delays as it is more I/O intensive since eviction paths are different from the read path.

Path ORAM provides a middle ground here, better bandwidth/larger storage than Circuit ORAM[74]. Although worse bandwidth/smaller storage than SSS ORAM[67]. Hence, initially we used PathORAM as the underlying ORAM schema for ZeroTrace, but we also added CircuitORAM support for ZeroTrace and describe more in detail about the performance of these ORAM schemas for ZeroTrace in Chapter 5.

3.3.3 PathORAM

We now give a summary of Path ORAM [68], the ORAM used in our current implementation. Path ORAM is a tree based ORAM schema, arguably one of the simplest and efficient ORAM schemas so far.

N	Number of blocks
B	Block size in bits
$L = \lceil \log(N) \rceil$	Height of tree
Z	Capacity of each bucket (in blocks)

Table 3.2: PathORAM Notations

Server Storage. Path ORAM stores N data blocks, where B is the block size in bits, and treats untrusted storage as a binary tree of height L (with 2^L leaves). Each node in the tree is a bucket that contains $\leq Z$ blocks. In the case of a bucket having $< Z$ blocks, remaining slots are padded with dummy blocks.

Controller Storage. The Path ORAM controller storage consists of a stash and position map. The stash is a set of blocks that Path ORAM can hold onto at any given time (see below). To keep the stash small (negligible probability of overflow), experiments show $Z \geq 4$ is required for the stash size to be bound to $\omega(\log N)$ [68]. The position map is a dictionary that maps each block in Path ORAM to a leaf in the server's binary tree. Thus, the position map size is $O(LN)$ bits.

Main Invariant. At any given point of time, a block in the system resides either on the path to the leaf label that it is mapped to in the position map, or in the local stash of the PathORAM controller.

Memory Access. As stated above, each block in Path ORAM is mapped to a leaf bucket in the server's binary tree via the position map. For a block a mapped to leaf l, Path ORAM guarantees that block a is currently stored in (i) some bucket on the path from the tree's root to leaf l, or (ii) the stash. Then, in order to perform a read/write request to block a (mapped to leaf l), we perform the following steps:

- Read the leaf label l for the block a from the position map.
- Re-assign this block to a freshly sampled leaf label l', chosen uniformly at random.
- Fetch the entire path from the root to leaf bucket in server storage.
- Retrieve the block from the combination of the fetched path and the local stash.
- Write back the path to the server storage. In this step the client must push blocks in the stash as far down the path as possible, while keeping with the main invariant. This strategy minimizes the number of blocks in the stash after each access and is needed to achieve a small (logarithmic) stash size.

Security intuition. The adversary's view during each access is limited to the path read/written (summarized by the leaf in the position map) during each access. This leaf is re-assigned to a uniform random new leaf on each access to the block of interest. Thus, the adversary sees a sequence of uniform random-sampled leaves that are independent of the actual access pattern.

Extension: Recursion. The Path ORAM position map is $O(LN)$ bits, which is too large to fit in trusted storage for large N. To reduce the client side storage to $O(1)$, Path ORAM can borrow the standard recursion trick from the ORAM constructions of Stefenov et al. [67] and Shi et al. [60].

The idea is to store the position map itself as a smaller ORAM on the server side and then recurse on it. Each smaller "position map" ORAM must be accessed in turn, to retrieve the leaf label for the original ORAM. Hence on the server side we store $ORAM_0, ORAM_1, ..., ORAM_x$, where $ORAM_0$ is the actual data ORAM that we set out

to store. Then only the position map corresponding to $ORAM_{\mathsf{x}}$ is stored at the client side. Making a recursive ORAM access corresponds to making an $ORAM$ request for each of there recursive ORAM levels, where at each level the data fetched corresponds to the postion map entry for the next level.

Extension: Integrity. Path ORAM assumes a passive adversary by default. To provide an integrity guarantee with freshness, one can construct a Merkle tree mirrored [68] onto the Path ORAM tree, which adds a constant factor to the bandwidth cost. We remark that when ORAM recursion is used, an integrity mechanism is also required to guarantee ORAM *privacy* [54].

Both integrity verification and ORAM recursion will be needed in our final design to achieve a performant system against active attacks.

3.3.4 Circuit ORAM

We now briefly highlight the differences between Circuit ORAM [74] and Path ORAM. In the interest of space, we describe our work using PathORAM as the memory controller since it is the conceptually simpler ORAM schema. Circuit ORAM was designed with the intent of having the smallest circuit complexity.[5] Both of these construction operate identically upto the fetch path step. The difference lies in their eviction strategy.

Circuit ORAM uses two additional eviction paths unlike Path ORAM which evicts blocks from the local stash onto the fetched path itself. The strategy is to perform eviction on a path in a single pass over (the stash and) the path, by picking up blocks that can be pushed deeper down the path and dropping it into vacant slots that are deeper in the path. This however requires some amount of "foresight" for which blocks can be moved to a deeper location in the path and if there are vacant slots that could accommodate them. To achieve this foresight, Circuit ORAM makes two meta data scans over each eviction path, to construct helper arrays that assist in performing eviction in a single (stash +) path scan.

There are two differences between these eviction strategies in the context of ZeroTrace

- Circuit ORAM introduces more I/O bandwidth than Path ORAM, since it has to fetch and evict two additional paths per access.
- The stash required by Circuit ORAM is much lesser than that of PathORAM.

[5] In the interest of optimizing ORAMs for use in the multi-party computation (MPC) context

Chapter 4

ZeroTrace

We now describe the design of our core memory controller which is implemented on the server. We focus on the details of supporting our strongest level of security: obliviousness against an active adversary (Section 2.2). The entire system is shown in Fig. 4.1. The design's main component is a secure Intel SGX enclave which we henceforth call the ORAM Controller Enclave. This ORAM Controller Enclave acts as the intermediary between client and the server. The client and controller enclave engage in logical data block requests and responses. Behind the scenes, the ORAM Controller Enclave interacts with the server to handle the backend storage for each of these requests.

4.1 Design Summary

Security challenges and solutions. Since ZeroTrace's ORAM controller runs inside an enclave, and is therefore vulnerable to software-level side channel attacks (Section 3.2.1), we design the ORAM controller to run as an *oblivious program*. (A similar approach is used to guard against software side channels by Ohrimenko et al. [49] and Rane et al. [51].) For instance, if the ORAM controller were to access an index in the position map directly, it would fetch a processor cache line whose address depended on the program access pattern. To prevent revealing this address, our oblivious program scans through the position map and uses oblivious select operations to extract the index as it is streamed through.

A second security challenge is how to map the controller logic itself to SGX enclaves. In a naive design, the entire ORAM controller and memory can be stored in the EPC. The enclave makes accesses to its own virtual address space to perform ORAM accesses

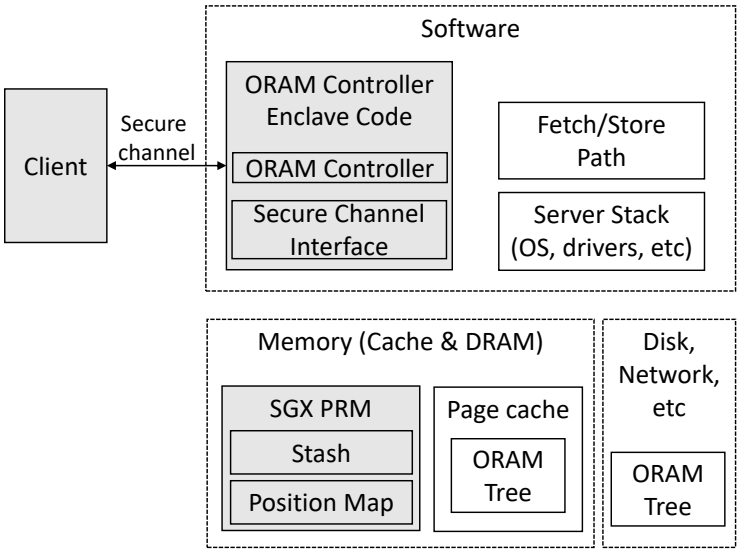

Figure 4.1: System components on the server. Trusted components (software and regions of memory) are shaded. Depending on the setting, the client may be connecting from a remote device (not on the server) or from another enclave on the same machine.

and run controller logic, and the OS uses EPC paging as needed. This design seems reasonable because it re-uses existing integrity/privacy mechanisms for protecting the EPC. Unfortunately, it makes supporting persistant storage difficult because the EPC is volitile (Section 3.2), moreover it incurs large EPC paging overheads (Section 3.2.2) and bloats the TCB (the entire controller runs in the enclave). To address this challenge, we make an observation that *once Path ORAM (and other tree-based ORAMs [74, 19, 53]) reveals the leaf it is accessing, the actual fetch logic can performed by an untrusted party.* Correspondingly, we split the ORAM controller into trusted (runs inside enclave) and untrusted (runs in Ring-3 outside of enclave) parts, which communicate between each other at the path fetch/store boundary. This approach has un-expected TCB benefits: we propose optimizations in Section 4.5 which bloat the path fetch/store code. By delegating these parts to untrusted code, they can be implemented with no change to the TCB.

Performance challenges and solutions. Running an oblivious ORAM controller inside of SGX *efficiently* requires a careful partitioning of the work/data-structures between the enclave (which controls the EPC pages $\sim$ 95 MB), untrusted in-memory code (which has access to DRAM) and untrusted code managing disk. For instance, the cost to access

ORAM data structures obliviously increases as their size increases. Further, as mentioned above, when the enclave memory footprint exceeds the EPC page limit, software paging introduces an additional overhead between 3× and 1000× – depending on the access pattern [3]. To improve performance, we will carefully set parameters to match the hardware and use techniques such as ORAM recursion to further reduce client storage.

Additionally, the ORAM storage itself should be split between DRAM and disk to maximize performance. For instance, we design the protocol to keep the top-portion of the ORAM tree in non-EPC DRAM when possible. In some cases, disk accesses can be avoided entirely. When the ORAM spills to disk, we layout the ORAM tree in disk to take advantage of parallel networks of disks (e.g., RAID0).

4.2 Client/Server Interface

4.2.1 Client Interface

The ORAM Controller Enclave exposes two API calls to the user, namely read(addr) and write(addr, data). Under the hood, both the API functions perform an ORAM access (Section 3.3.3).

4.2.2 Server Processes

The server acts as an intermediary between the trusted enclave and the data (either memory or disk). It performs the following two functions on behalf of the trusted enclave (e.g., in a Ring-3 application that runs alongside the enclave):

- FetchPath(leaf): Given a leaf label, the server transfers all the buckets on that path in the tree to the enclave.
- StorePath(tpath, leaf): Given a tpath, the server overwrites that existing path to the addresses deduced from the leaf label, leaf.

Passing data in/out of enclave. The standard mechanism of data passing between enclave and untrusted application is through a sequence of input/output routines defined for that specific enclave. The Intel SGX SDK comes with the Intel Edger8r tool that generates edge routines as a part of enclave build process. Edger8r produces a pair of

edge routines for each function that crosses the enclave boundary, one routine sits in the untrusted domain, and the other within the trusted enclave domain. Data is transferred across these boundaries by physically copying it across each routine, while checking that the original address range does not cross the enclave boundary.

TCB implications. Fetch/store path are traditionally the performance bottleneck in ORAM design. Given the above interface, these functions make no assumptions on the untrusted storage or how the server manages it to support ORAM. Thus, the server is free to perform performance optimizations on Fetch/Store path (e.g., split the ORAM between fast DRAM and slow disk, parallelize accesses to disk; see Section 4.5). Since Fetch/Store path are not in the TCB, these optimizations do not effect security.

4.3 Memory Controller Enclave Program

In this section we outline the core memory controller's enclave program which we refer to from now on as P.

4.3.1 Initialization

For initialization, the server performs the function $\mathsf{Load}(\mathsf{P}) \rightarrow (\mathsf{E}_\mathsf{P}, \phi)$, where P is the ZeroTrace Controller Enclave. The client can then verify the proof ϕ produced by this function to ensure that ZeroTrace has been honestly initialized by the server. We note that the proof also embeds within it a public key K_e from an asymmetric key pair $(\mathsf{K}_\mathsf{e}, \mathsf{K}_\mathsf{d})$ sampled within the enclave. The client encrypts a secret key K under this public key K_e for the enclave. The user and enclave henceforth communicate using this K for an authenticated encrypted channel.

4.3.2 System Calls

Our enclave logic does not make any syscalls. All enclave memory is statically allocated in the EPC based on initialization parameters. Server processes (i.e., Fetch/Store path) may perform arbitrary syscalls without impacting the TCB.

4.3.3 Building Block: Oblivious Functions

To remain data oblivious, we built the ORAM controller out of a library of *assembly-level functions* that perform oblivious comparisons, arithmetic and other basic functions. The only code executed in the enclave is specified precisely by the assembly instructions in our library (all compiler optimizations on our library are disabled).

Our library is composed of several assembly level instructions, most notably the **CMOV** x86 instruction [49, 51]. CMOV is a conditional move instruction that takes a source and destination register as input and moves the source to destination if a condition (calculated via the CMP instruction) is true. CMOV has several variants that can be used in conjunction with different comparison operators, we specifically use the CMOVZ instruction for equality comparisons. The decision to use CMOV was not fundamental: we could have also used bitwise instructions (e.g., AND, OR) to implement multiplexers in software to achieve the obliviousness guarantee.

CMOV safely implements oblivious stores because it does the same work regardless of the input. Regardless of the input, all operands involved are brought into registers inside the processor, the conditional move is performed on those registers, and the result is written back.

Throughout the rest of the section, we will describe the ORAM controller operations in terms of a wrapper function around cmov called `oupdate`, which has the following signature:

```
oupdate<srcT, dstT>(bool cond, srcT src, dstT dst, sizeT sz)
```

`oupdate` uses CMOV to obliviously and conditionally copy `sz` bytes from `src` to `dst`, depending on the value of a bit `cond` which is calculated outside the function. `src` and `dst` can refer to either registers or memory locations based on the types `srcT` and `dstT`. We use template parameters `srcT` and `dstT` to simplify the writing, but note that CMOV doesn't support setting `src` to a memory location by default. Additional instructions (not shown) are needed to move the result of a register `dst` CMOV to memory.

4.3.4 Building Block: Encryption & Cryptographic Hashing

Our implementation relies on encryption and integrity checking via cryptographic hashing in two places.

- First, when the client sends an ORAM request to the ORAM Controller Enclave, that request must be decrypted and integrity checked (if integrity checking is enabled).

- Second, during each ORAM access, the path returned and received by Fetch/Store Path (Section 4.2.2) need to be decrypted/re-encrypted and integrity verified.

These routines must also be oblivious, hence for encryption, we use the Intel instruction set extensions AES-NI, which were designed by Intel to be side channel resistant (i.e., the AES SBOX is built directly into hardware). Unless otherwise stated, all encryption is AES-CTR mode, which can easily be achieved by wrapping AES-NI instructions in oblivious instructions which manage the counter. For hashing we use SHA-256, which is available through the Intel tcrypto library.

To avoid confusion, we note that SGX has separate encryption/hashing mechanisms to ensure privacy/integrity of pages evicted from the EPC [15]. But since our design accesses ORAM through a Fetch/Store Path interface, we cannot use these SGX built-in mechanisms for ORAM privacy/integrity.

4.3.5 ORAM Controller

The ORAM Controller handles client queries of the form $(\mathsf{op}, \mathsf{id}, \mathsf{data}*)$, where op is the mode of operation, i.e. read or write, id corresponds to an identifier of the data element and $\mathsf{data}*$ is a dummy block in case of read and the actual data contents to be written in case it is a write operation. These queries are encrypted under K, the secret key established in the Initialization (Section 4.3.1) phase. The incoming client queries are first decrypted within the enclave program. From this point, the ORAM controller enclave runs the ORAM protocol. Given that the adversary may monitor any pressure the enclave places on shared hardware resources, the entire ORAM protocol is re-written in an oblivious form. The Raccoon system performed a similar exercise to convert ORAM to oblivious form, in a different setting [51].

We discuss in detail this process for Path ORAM scheme, we similarly designed and implemented an oblivious variant of Circuit ORAM as well. For simplicity, we focus on explaining this process of converting an ORAM schema to an oblivious form using our Path ORAM scheme. Path ORAM can be broken into two main data-structures (position map and stash) and three main parts. We now explain how these parts are made oblivious.

Oblivious Leaf-label Retrieval. When the enclave receives a request $(\mathsf{op}, \mathsf{id}, \mathsf{data}*)$, it must read and update a location in the position map (Section 3.3.3) using $\mathsf{oupdate}$ calls, as shown in the pseudocode below.

```
newleaf = random(N)
for i in range(0, N):
  cond = (i == x)
  oupdate(cond, position_map[i], leaf, size)
  oupdate(cond, newleaf, position_map[i], size)

newleaf = random(N)
leaf = position_map[x]
position_map[x] = newleaf
```

We note that P samples a new leaf label through a call to AES-CTR with a fresh counter. Due to a requirement in Section 4.4, where execution must be deterministic, we will assume leaf generation is seeded by the client when the ORAM is initialized (and not by a TRNG such as Intel's RDRAND instruction). The entire position map must be scanned to achieve obliviousness, as will be the case for the other parts of the algorithm, regardless of when `cond` is true. At the end of this step, the enclave has read the leaf label, leaf, for this access.

Oblivious Block Retrieval. P must now fetch the path for leaf (Section 3.3.3) using a Fetch Path call (Section 4.2.2). When the server returns the path, now loaded into enclave memory, P does the following:

```
path = FetchPath(leaf)
for p in path:
  for s in stash:
    cond = (p != Dummy) && (s != occupied)
    oupdate(cond, s, p, BlockSize)
result = new Block
for s in stash:
  cond = (s.id == id)
  oupdate(cond, s, result, BlockSize)
```

The output of this step is `result`, which is encrypted and returned to the client application.

In the above steps, iterating over the stash must take a data-independent amount of time. First, regardless of when oupdate succeeds in moving a block, the inner loop runs to completion. When the update succeeds, a bit is obliviously set to prevent the CMOV from

succeeding again (to avoid duplicates). Second, the stash size (the inner loop bound) must be data-independent. This will not be the case with Path ORAM: the stash occupancy depends on the access pattern [68]. To cope, we use a stash with a static size at all times, and process empty slots in the same way as full slots. Prior work [41, 68] showed that a stash size of 89 to 147 is sufficient to achieve failure probability of $2^{-\lambda}$ with the security parameter values from $\lambda = 80$ to $\lambda = 128$. In our implementation, we use a static stash size of 90.

Oblivious Path Rebuilding. Finally, P must rebuild and write back the path for leaf (Section 3.3.3) using internal logic and a Store Path call (Section 4.2.2). P rebuilds this path by making a pass over the stash for each bucket in the path as shown here:

```
for bu in new_path:
  for b in bu:
    for s in stash:
      cond = FitInPath(s.id,leaf)
      oupdate(cond, b, s, BlockSize)
StorePath(leaf,new_path)
```

For each bucket location *bu* on path to leaf in reverse order (i.e. from leaf to root), iterates over the block locations b (in the available Z locations) and perform oupdate calls to obliviously move compatible blocks from the stash to that bucket (using an oblivious subroutine called `FitInPath`). This greedy approach of filling buckets in a bottom to top fashion is equivalent to the eviction routine in Section 3.3.3. At the end, P then calls Store Path on the rebuilt path, causing the server to overwrite the existing path in server storage.

Encryption and Integrity. As data is processed in the block retrieval and path re-building steps, it is decrypted/re-encrypted using the primitives in Section 4.3.4. At the same time, an oblivious implementation of the Merkle tree checks and re-build is done to verify and maintain integrity with freshness.

4.4 Persistant Integrity

An important attribute in storage systems is to be persistant and recoverable across protocol disruptions. This is particularly important for ORAM, and similar memory

controller backends, where corrupting any state (in the ORAM Controller Enclave itself or in the ORAM trees) can lead to partial or complete loss of data. SGX exacerbates this issue, as enclave state is wiped on disruptions such as reboots and power failures.

We now discuss an extension to ZeroTrace that allows untrusted storage and the ORAM Controller Enclave to recover from data corruptions and achieve persistant integrity. First, we state a sufficient condition to achieve fault tolerance. We model an enclave program as a function P which performs $S_{t+1} \leftarrow \mathsf{P}(I_t, S_t)$, where I_t is the t-th request made by the client and S_t is the enclave state after requests $0, \ldots, t-1$ are made. When we say *enclave protocol*, we refer to the multi-interactive protocol between the client and P from system initialization onwards.

Definition 4.4.1 (Fault tolerance) *Suppose an enclave protocol has completed t' requests. If the enclave protocol is designed such that the server can efficiently re-compute $S_{t+1} \leftarrow \mathsf{P}(I_t, S_t)$ for any $t < t'$, then the enclave protocol is fault tolerant.*

This provides fault tolerance as follows: if the current state $S_{t'}$ is corrupted, $S_{t'}$ can be iteratively re-constructed by replaying past (not corrupted) states and inputs to P. We remark that the above definition is similar to RDD fault tolerance in Apache Spark [86, 89]. Finally, the above definition isn't specific to ORAM controllers, however we will assume an ORAM controller for concreteness.

Functionality. In our setting, S includes the ORAM Controller Enclave state (the stash, position map, ORAM key, merkle root hash) and the ORAM tree. In practice, the server can snapshot S at some time t (or at some periodic schedule), and save future client requests $I_t, \ldots, I_{t'}$ to recover $S_{t'}$. Thus, we must add a server-controllable operation to the ORAM Controller Enclave that writes out the enclave state to untrusted storage on-command.

Security. To maintain the same security level as described in Section 2.2, the above scheme needs to defeat all mix-and-match and replay attacks.

A mix-and-match attack succeeds if the server is able to compute $\mathsf{P}(I_a, S_b)$ for $a \neq b$, which creates a state inconsistent with the client's requests. These attacks can be prevented by encrypting state in S and each client request I with an authenticated encryption scheme, that *uses the current request count t* as a nonce. The client generates each request I and thus controls the nonce on I. For S: the enclave controls the nonce on its private state and integrity verifies external storage with a merkle tree (whose root hash is protected as

a part of the private state). On re-execution, P can integrity-verify I_a and S_b under the constraint that $a = b$.

A replay attack succeeds if the server is able to learn something about the client's access pattern by re-computing on consistant data – e.g., $\mathsf{P}(I_t, S_t)$. Replay attacks are prevented if replaying $\mathsf{P}(I_t, S_t)$ always results in a statistically indistinguishable trace trace (Section 3.1). In our setting, we must analyze two places in the protocol. First, the path written back to untrusted storage after each request (Section 4.3.5) is always re-encrypted using a randomized encryption scheme that is independent of underlying data. Second, the *leaf label* output as an argument to Fetch/Store Path (Section 4.2.2) must be *deterministic* with respect to previous requests. This property is achieved by re-assigning leaf labels using a pseudo-random number generator.

4.5 Optimizing Fetch/Store Path

We now discuss several performance optimizations/extensions for the Fetch/Store Path subroutines, to take advantage of the server's storage hierarchy (which consists of DRAM and disk). Since these operations run in untrusted code, they do not impact the TCB.

Scaling bandwidth with multiple disks. Ideally, if the server supports multiple disks which can be accessed in parallel (e.g., in a RAID0), the time it takes to perform Fetch/Store Path calls should drop proportionally. We now present a scheme to perfectly load-balance a Tree ORAM in a RAID0-like configuration.

RAID0 combines W disks (e.g., SSDs, HDDs, etc) into a larger logical disk. A RAID0 'logical disk' is accessed at *stripe* granularity (S bytes). S is configurable and $S = 4$ KB is reasonable. When disk stripe address i is accessed, the request is sent to disk $i\%W$ under the hood.

The problem with RAID0 (and similar organizations) combined with Tree ORAM is that when the tree is laid out flat in memory, the buckets touched on a random path will not hit each of the W disks the same number of times (if $S{*}W > B{*}Z$ for ORAM parameters B and Z). In that case, potential disk parallelism is lost. We desire a block address mapping from (ORAM tree address, at stripe granularity) to (RAID0 stripe address) that equalizes the number of accesses to each of the W disks, while ensuring that each disk stores an equal (ORAM tree size) / W Byte share. Call this mapping $\mathsf{Map}(\mathsf{tree\ addr}) \rightarrow$ RAID addr, which may be implemented as a pre-disk lookup table in untrusted Fetch/Store Path code.

We now describe how to implement Map. First, define a new parameter subtree height H. A subtree is a bucket j, and all of the descendant buckets of j in the tree, that are $< H$ levels from bucket j. For ORAM tree height L, choose $H < L$ (ideally, H divides L). Break the ORAM tree into disjoint subtrees. Second, consider the list of all the subtrees ALoST. We will map each stripe-sized data chunk in each subtree to a disk in the RAID0. The notation `Disk[k] += [stripeA, stripeB]` means we use an indirection table to map `stripeA` and `stripeB` to disk `k`. We generate `Disk` as:

```
for subtree_index in length(ALoST):
  for level in subtree: // levels run from 0...H-1
      // break data in subtree level into stripe-sized chunks
    stripes_in_level = ALoST[subtree_index][level]
      Disk[(subtree_index + level) % W] += stripes_in_level
```

When $W = H$, mapping each subtree level to a single disk means any path in the ORAM tree will access each disk $O(L/H)$ times. Changing the subtree level $\rightarrow$ disk map in a round-robin fashion via `subtree_index` ensures that each disk will hold the same number of stripes, counting all the subtrees. Finally, from `Disk`, it is trivial to derive Map.

Caching the ORAM tree. A popular Tree ORAM optimization is to cache the top portion of the ORAM tree in a fast memory [53, 41]. This works because each access goes from root to leaf: caching the top l' levels is guaranteed to improve access time for those top l' levels. Because the shape is a tree, the top levels occupy relatively small storage (e.g., caching the top half requires $O(\sqrt{N})$ blocks of storage).

This optimization is very effective in our system (as seen in Figure 5.2) because the server (who controls Fetch/Store Path) can use any spare DRAM (e.g., GigaBytes) to store the top portion of the tree. In that case, Fetch/Store Path allocates regular process memory to store the top portion, and explicitly stores the lower portion behind disk I/O calls.

4.6 Security Analysis

We now give a security analysis for the core memory controller running ORAM. Since we support ORAM, we wish to show the following theorem:

Theorem 4.6.1 *Assuming the security of the Path ORAM protocol, and the isolated execution and attestation properties of Intel SGX, the core memory controller is secure according to the security definition in Section 3.1.*

In this section, we'll prove the above theorem informally, by tracing the execution of a query in ZeroTrace, step by step as shown in Figure 4.2.

Claim 4.6.1.1 *Initialization is secure*

For initialization, the enclave first samples a public key pair, then includes this public key in the clear with the enclave measurement, in the attestation (Section 3.2) that it produces. No malicious adversary can tamper with this step, as it would have to produce a signature that is verifiable by the Intel Attestation Service.

Claim 4.6.1.2 *Decrypting and encrypting requests leak no information*

We use AES-NI, the side-channel resilient hardware instruction by Intel for performing encryption and decryption.

Claim 4.6.1.3 *Oblivious Leaf-Label Retrieval leaks no information*

Retrieving a leaf label from the EPC-based position map performs a data-independent traversal of the entire position map via oupdate (Section 4.3.3) operations. oupdate performs work independent of its arguments within the register space of the processor chip, which is hidden from adversarial view. Thus, the adversary learns no information from observing leaf-label retrieval.

Claim 4.6.1.4 FetchPath *leaks no information*

FetchPath retrieves the path to a given leaf label. The randomness of this reduces to the security of the underlying Path ORAM protocol (Section 3.3.3).

Claim 4.6.1.5 *Verifying fetched path leaks no information*

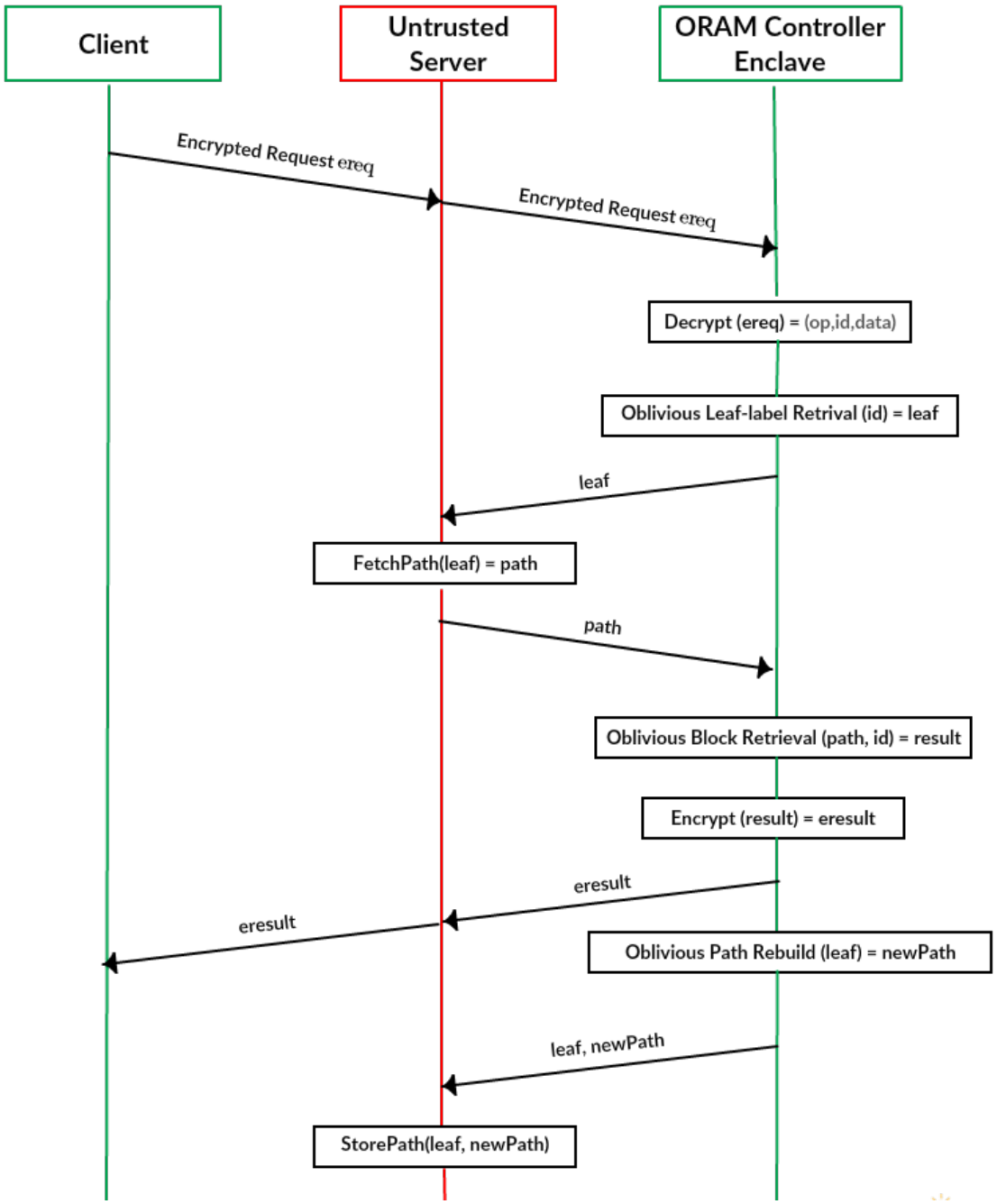

Figure 4.2: Execution of an access request

To verify the integrity of a fetched path, the enclave re-computes the Merkle root using SHA-256 over the path it fetched and subling hashes [68]. We note that our current implementation uses SHA-256 from the Intel tcrypto library, which is not innately side-channel resistant. Despite this, our scheme still achieves side-channel resistance because all SHA-256 operations are over *encrypted* buckets. The same argument applies when rebuilding the path on the way out to storage.

Claim 4.6.1.6 *Oblivious Block Retrieval leaks no information*

Once FetchPath completes, the only code that processes the path, to load that path into the stash and return the requested block to the user, is decryption logic plus the oblivious subroutine given in Section 4.3.5. Since the length of `path` and `stash` are data-dependent, obliviousness reduces to the security of oupdate (see Claim 4.6.1.3).

Claim 4.6.1.7 *Oblivious Rebuild leaks no information*

Same argument as Claim 4.6.1.6, since `new_path`, `bu` and `stash` have data independent size.

Claim 4.6.1.8 StorePath *leaks no information*

StorePath returns the new path to a leaf label that was fetched by an ORAM controller enclave. From the adversary's perspective, the stored path itself is an encrypted payload of a known size, independent of underlying data.

Chapter 5

Implementation and Evaluation

5.1 Experiment Setup

We implemented and evaluated the performance of ZeroTrace on a Dell Optiflex 7040, with a 4 core Intel i5 6500 Skylake processor with SGX enabled and 64 GB of DRAM.

Beyond DRAM, our system utilizes a Western Digital WD5001AALS 500 GB 7200 RPM HDD as backing untrusted storage. ZeroTrace is implemented purely in C/C++ for both performance and easier compatibility with Intel SGX as enclave code is limited to purely C/C++ code. Our implementation consists of 4000 lines of code in total, with 1800 lines of code within the enclave, which counts towards the TCB. We measure the time it takes our memory service enclaves to complete user requests. In all experiments, our core memory controller and data-structure APIs are implemented as application libraries in stand-alone enclaves – to best model their performance as plug-and-play memory protection primitives (Section 2.1). Thus, request time includes the time to send/receive the request to/from the enclave, as well as the time to process the request (e.g., do an ORAM access).

5.2 Evaluation of Core Memory Controller

We first evaluate performance of ZeroTrace for the core memory controller component, configured to resist software-based side channel attacks from an active adversary (Section 2.2). Figure 5.1 shows the time taken by a single access request in contrast with the number of data blocks N in the system, for DRAM and HDD untrusted storage systems. For the

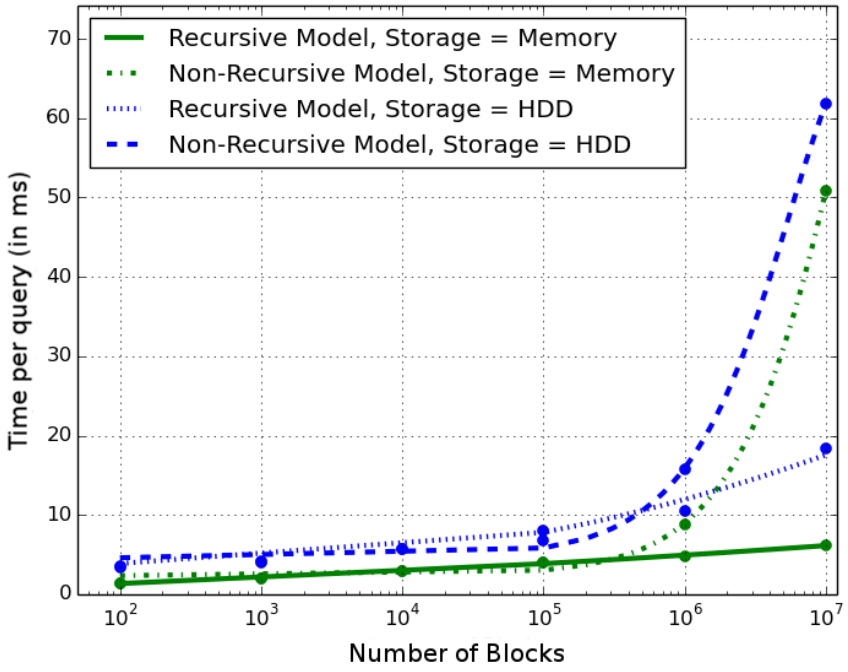

Figure 5.1: Representative result. Shows the number of data blocks vs. time per request, with data blocks of size 1 KB. We use ZeroTrace with a PathORAM schema and the storage is HDD. We use a tree top caching mechanism which caches all the recursion and integrity trees and all but the last level of the data tree.

points using the ORAM recursion technique, we use a position map of size 1 KB within the EPC and always set the recursion ORAM block size to 64 Bytes (a cacheline). When recursion is not used, the position map is streamed through the EPC, paging as necessary.

In Figure 5.1, we see recursive ORAM pays off for large datasets, we note that in the non-recursive settings the overhead comes from scanning the large position map to maintain obliviousness. This matches the theory [68] and our system uses whichever configuration achieves the best performance, depending on public parameters.

Performance breakdown. We further analyze the time taken to run oblivious enclave code in the memory controller, vs. the time spent servicing untrusted memory requests, in Figure 5.2. The main results are that the oblivious controller is the bottleneck given fast

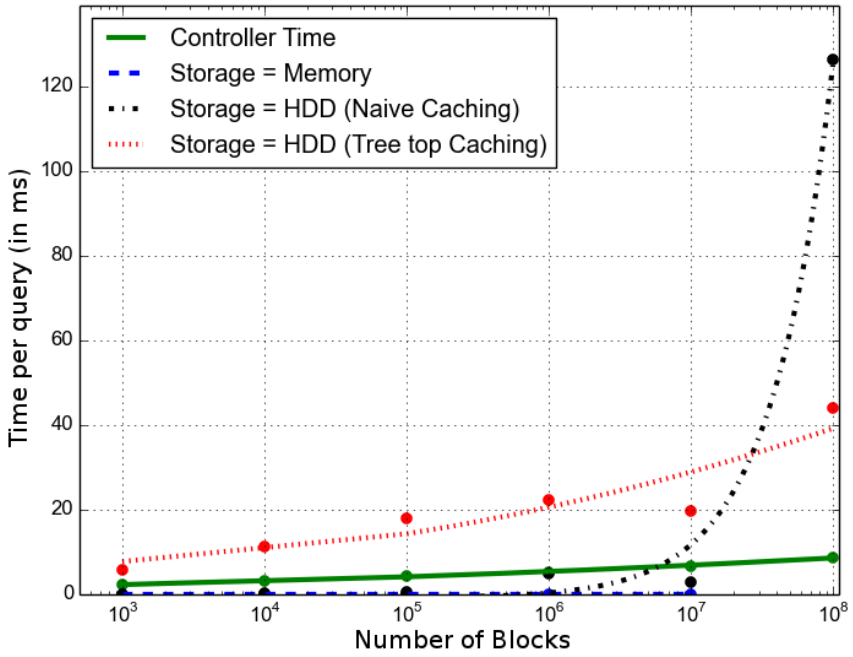

Figure 5.2: Detailed performance breakdown. Shows the number of data blocks vs. time spent in different parts of the request, with different storage backends, with a block size of 1 KB. Total time per request is the sum of controller and storage (DRAM or HDD) times. Naive caching delegates caching completely to the OS, while tree top caching caches all the recursion and integrity trees and all but one level of the data tree until 10^7 blocks. Beyond that point it caches as many levels of the data tree as it can within the DRAM limit of 64GB. DRAM is shown until 10^7 blocks, which is our DRAM capacity.

untrusted storage devices (e.g., DRAM). Our system caches the top portion of the ORAM tree in DRAM until half of the DRAM (32 GB) is used, after which the system incurs a large latency for disk seeks. This issue isn't fundamental; our system can use an SSD to improve disk latency.

For completeness, Figure 5.3 shows the controller request time varying the data ORAM block size. For data ORAM block sizes, the curve is flat since the cost of recursion dominates.

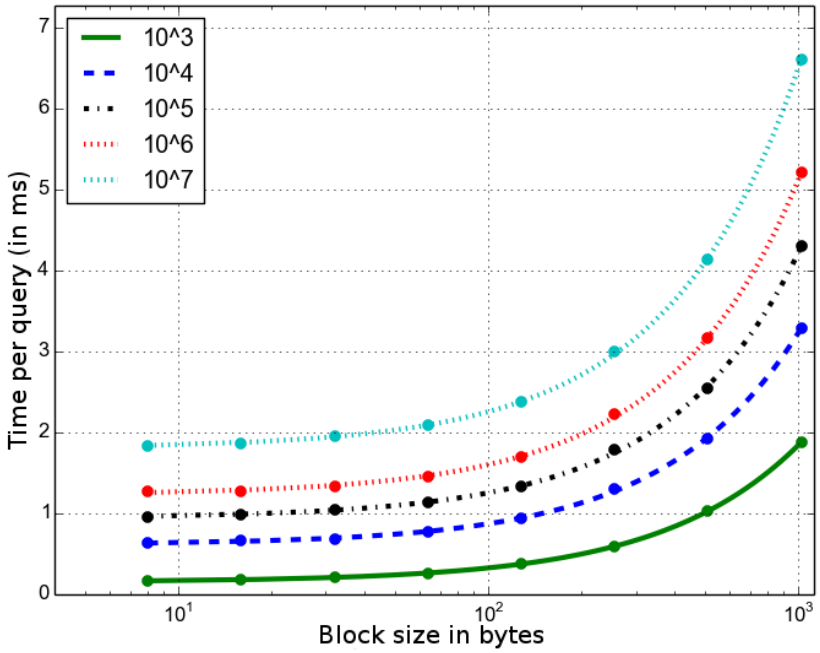

Figure 5.3: Performance as a function of data ORAM block size for datasets with varying number of blocks N.

5.3 Evaluation of Controller Flexibility

We envision that applications could choose to trade stronger levels of security in exchange for query efficiency depending on its requirements. ZeroTrace was hence designed to be extremely flexible in switching between these levels. In figure 5.4, we contrast the time taken per query in contrast with the number of blocks N, for four different levels of security. At the strongest security setting, we present an oblivious memory controller resistant to active adversaries. In this setting we protect against all the side channels introduced by Intel SGX along with integrity verification through a Merkle tree (Section 3.3.3). If the application setting envisions a passive (honest but curious) adversary then one can forego the integrity verification module to obtain slightly better performance.

Alternatively one can chose to forego the controller obliviousness, while this mode provides obliviousness to accesses at data level from the underlying PathORAM schema,

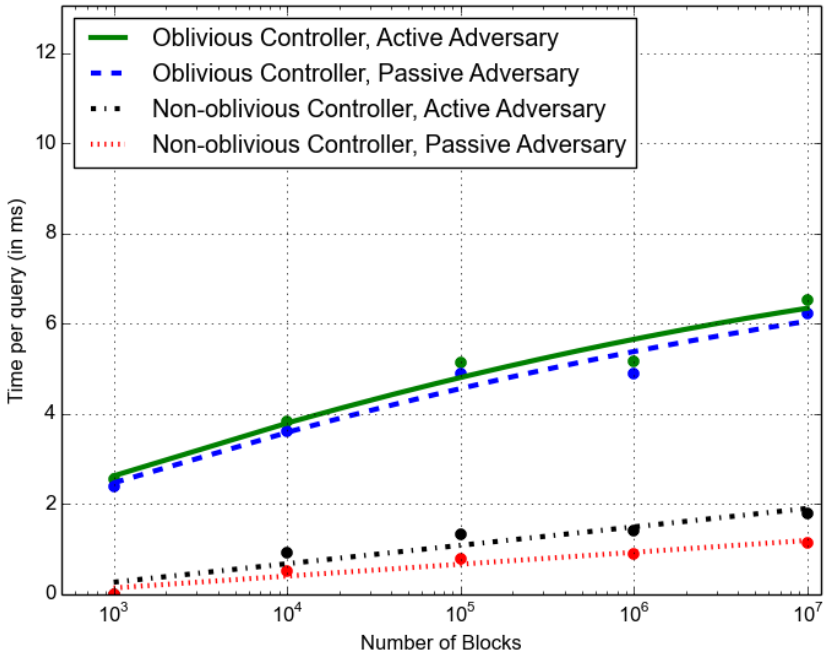

Figure 5.4: Evaluation of our oblivious memory controller library for different security levels with PathORAM as the underlying ORAM scheme.

it becomes susceptible to side channel leakages of Intel SGX (Section 3.2).

5.4 Improving the Controller Time

ZeroTrace is aimed at providing secure memory abstractions when using SGX for secure computations. Keeping this in mind, looking at Figure 5.4, we note that when we use memory as the backend, ZeroTrace is throttled by it's controller time. In order to improve upon this, we added support for an oblivious CircuitORAM schema as the underlying ORAM for ZeroTrace. In Figures 5.5 and 5.6, we compare the performances of ZeroTrace with the different underlying schemas.

In Figure 5.5, we see that CircuitORAM does not improve the performance of ZeroTrace when the underlying data block size is small (8 bytes in this figure). Although the circuit

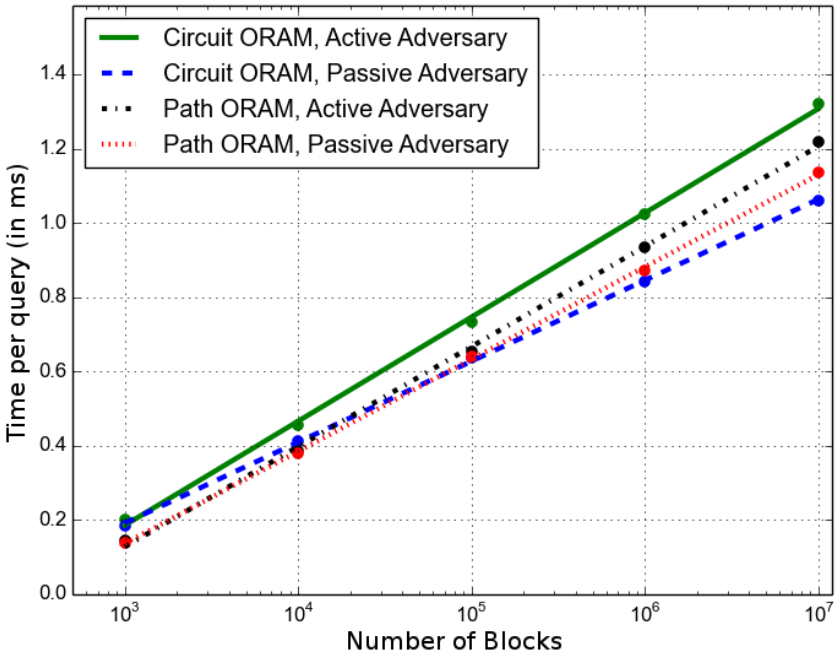

Figure 5.5: Evaluation of ZeroTrace comparing PathORAM and CircuitORAM as the underlying ORAM schema for data block sizes of 8 bytes.

complexity of Circuit ORAM is better than that of PathORAM, the discrepancy arises due to the additional I/O required in CircuitORAM. As we mention in Section 3.3.4, CircuitORAM uses 2 eviction paths chosen independently of the read path. As seen in Figure 5.5 itself, if we were to switch to a passive adversarial model, CircuitORAM becomes slightly more efficient than PathORAM just from the time saved in not performing the multiple path integrity verification routines. Figure 5.6 we compare the performances of the two underlying ORAM schemas of ZeroTrace, under varying data block sizes. It is clear from the figure that CircuitORAM outperforms PathORAM in ZeroTrace under larger block sizes. This reflects expected behaviour, as CircuitORAM requires only a much smaller stash size than PathORAM and also has a one pass eviction circuit as mentioned in Section 3.3.4, thus reducing the cost for oblivious evictions with increase in block size from the time saved in scanning stash and eviction path.

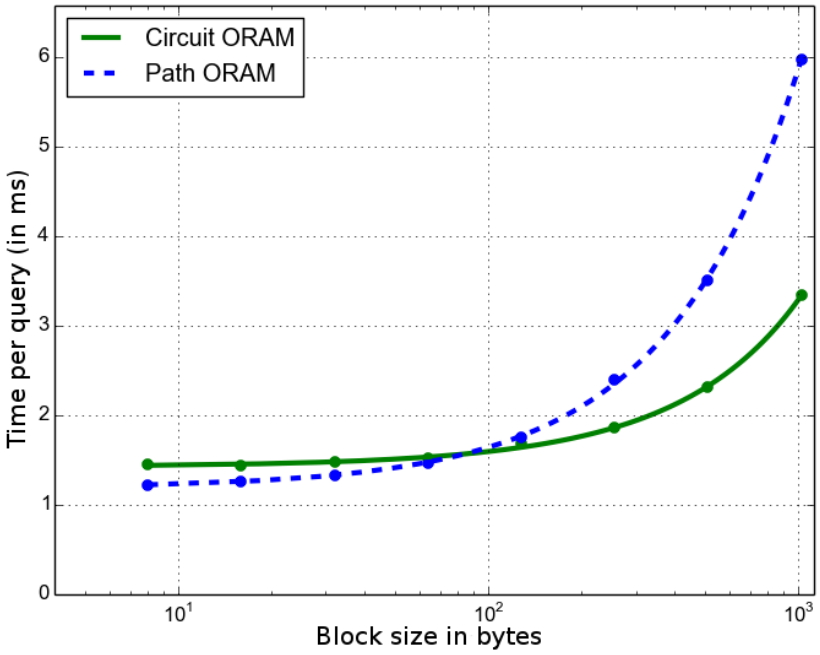

Figure 5.6: Evaluation of ZeroTrace comparing PathORAM and CircuitORAM as the underlying ORAM schema for varying data block sizes with $N = 10^7$

5.5 Evaluation of Data-Structure Modules

We now evaluate a library of oblivious data-structures, which use our core memory controller as a primitive. Data-structures expose two function calls to client applications:

Initialize(N, size) Informs the ZeroTrace memory controller enclave to provision storage for N size-Byte blocks.

Access(op, req) Performs the operation op, given arguments as a tuple req, whose format changes based on the data-structure. Enclaves are required to sanitize this input to ensure proper formatting.

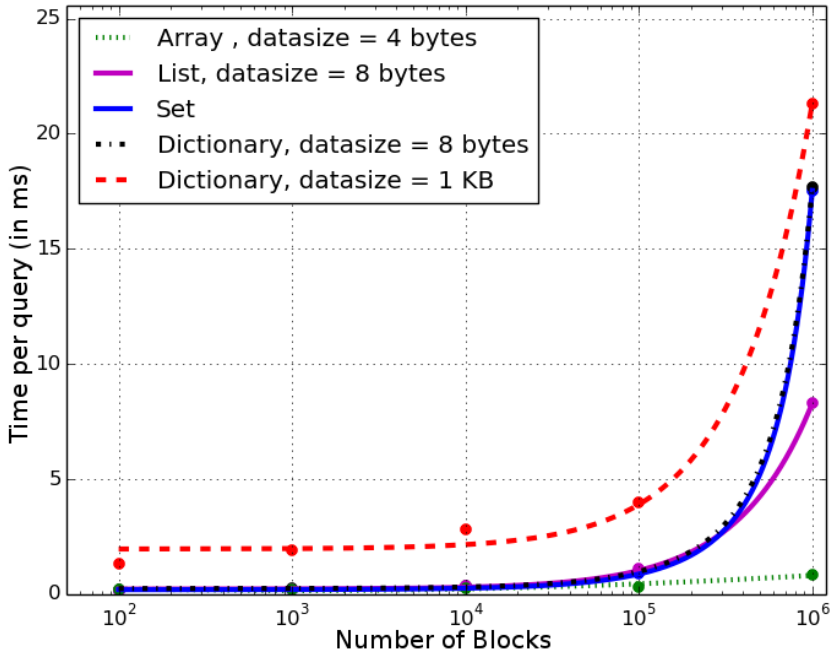

Figure 5.7: Evaluation of our oblivious memory controller library for Set/Dictionary/List/Array. Array is a direct call to our core memory controller, which uses ORAM recursion to be asymptotically efficient.

Data-structures supported. Our current implementation supports oblivious arrays, sets, dictionaries and lists. Array is a passthrough interface to our oblivious core memory controller, suppporting the same interface read(addr) and write(addr, data). Sets support the operations insert(data), delete(data) and contains(data). Dictionaries support put(tag, data) and get(tag). Lists support insert(index, data) and remove(index). These options are implemented obliviously in the enclave followed by the necessary ORAM lookups.

Implementation and results. In our current implementation, each data-structure maintains a primitive array which stores information used to lookup the data block stored by the memory controller. For example, sets and dictionaries use the array to store cryptographic hashes of data blocks, which map array indices to addresses in the memory controller. (Given our interface for set, above, the data storage is simply the array of

hashes. Thus, set does not have a datasize.) The data-structure logic obliviously scans the array in $O(N)$ time, to find the block, and then makes a single memory controller access to fetch the block. Figure 5.7 shows the performance for these data structures. While our design is efficient for reasonably sized data-structures ($\leq 10^5$ elements), the $O(N)$ time scan dominates for larger datasets. The $O(N)$ effect can be improved with optimized data-structures from Wang et al. [75], which makes use of ORAMs and can use our core memory controller as a primitive as well.

Chapter 6

Towards Secure Remote Computation

As we mentioned in the start of the thesis, the problem that interests us is how can we improve upon existential secure computation models by leveraging secure hardware tools. In this chapter, we discuss different mechanisms of secure computation that exist today, and how and where our work fits in this spectrum.

6.1 Intel TPM + TXT

Intel TPM (Trusted Platform Module) is a hardware based security device that secures boot process integrity. It protects the system start-up process and ensures that it is tamper-free before releasing control to the operating system. It does so by ensuring that measurements of the boot process matches previously known "trusted" values or Static Root of Trust Measurements (SRTM).

Intel Trusted Execution Technology (TXT) is Intel's implementation of the Trusted Computing Group's specification for trusted computing hardware, designed to ensure that cloud infrastructure as a service (IaaS) has not been tampered with. Maintaining a chain of trust while operating in an environment that is constantly exposed to unknown software entities is non trivial. Hence Intel TXT provides instructions to establish another RTM, named Dynamic Root of Trust for Measurement (DRTM), which allows the launch of a measured trusted program without resorting to a platform reset.

TXT makes use of TPM and reduces the software inside the secure container to a virtual machine hosted by the CPU's hardware virtualization features. TXT isolates the software inside the container from untrusted software by ensuring exclusive control over the entire computer while it is active. This is accomplished by a secure initialization authenticated code module (SINIT ACM) that performs a warm system reset before starting the container's VM. In short, this module stops and blocks all ongoing processes, interrupts and I/O and disables all but one processor, and this is what is referred to as the Measured Launched Environment (MLE). Dynamic RTM begins upon user or program request at any time during or after boot. The TXT enabled computer will measure the about-to-start program(s) in a similar fashion to the static RTM ensuring that the started programs have not been altered since a previously identified trusted configuration. Furthermore, because of the MLE, the resulting program can be run with secure input and output, and memory curtaining / protected execution. By measuring all programs that are launched into a secure I/O and memory/execution protected partition, TXT guarantees that none of the programs in the trusted partition have been altered and protects their data and execution from corruption while running.

The primary goal of using Intel TXT is to validate that there have been no unauthorized changes to critical parts of the code that provides the secure environment. This check is performed each time the environment launches, whether it is a cold boot, warm boot, or exiting one hypervisor and launching a new one. These components are measured by creating an SHA-1 hash of the component and validating that hash against a set of securely stored values.

However Intel TPM+TXT has several shortcomings. First, using a system for secure computing via Intel TPM+TXT implies that this system cannot be used for any other purposes, i.e. it yields poor hardware utilization. TXT does not implement DRAM encryption or HMACs, and therefore is vulnerable to physical DRAM attacks. Wojtczuk et al. [79, 81] have demonstrated several attacks against Intel TXT. In their work, they point out that TXT inherently relies on a trusted System Management Mode (SMM). However, there have been attacks on SMM handler's [18, 56, 80, 20], and using these they demonstrate how to attack memory used by the TXT container. To summarize, Intel TPM+TXT's fundamental flaw is the inherent trust required on the SMM of the remote server. This design flaw was resolved in Intel SGX, by ensuring that the PRM set aside by SGX, is not accessible to even SMM. Although on the flip side, an advantage of using Intel TPM+TXT is that it requires no reprogramming or tailoring at an application level as opposed to other solutions.

6.2 Fully Homomorphic Encryption (FHE)

Somewhat homomorphic encryption constructions have been around for a while, but it wasn't until 2009, when Gentry introduced "bootstrapping" [24] did we have a full-fledged FHE scheme that was semantically secure and compact[1]. Over the last few years there have been several constructions for FHE [25, 10, 21] that made use of this bootstrapping technique, which lead to recent efforts to standardize FHE [55]. Fundamentally, bootstrapping enabled one to convert a levelled homomorphic encryption schema that could homomorphically compute it's own decryption circuit and slightly more, to a fully homomorphic encryption scheme that can compute arbitrary depth circuits homomorphically.

However, FHE is an extremely expensive tool, that introduces orders of magnitude of overhead on a computation. For instance computing the AES-128 circuit today through FHE using HElib [30] takes over 18 minutes for 180 blocks, i.e. an amortized 6 seconds per block [31] . Native AES-NI on the other hand has a throughput of about 2607 MB/sec [12]. We note that all of this is without bootstrapping. Bootstrapping aggravates this problem all the more, if our desired goal was to make large computations, such as running machine learning algorithms on a large dataset, then having to periodically bootstrap becomes an extremely inefficient process that scale with size of the underlying ciphertext. Another concern with computing through FHE is the lack of integrity guarantees. In order to attain authenticity guarantees on the computation, one has to additionally use a proof system (such as SNARKS [7] or homomorphic signature schemes [28]) in conjunction with it.

Hence, we see that there are several efficiency throttles for FHE as we know it today. The most efficient levelled FHE scheme today induces a per-gate asymptotic overhead of $\tilde{\mathsf{O}}(\lambda^2)$. As we mentioned earlier, the plaintext spaces are confined to short integer vectors, and hence adapting large datasets to make use of FHE, is itself a challenging task. This problem of handling input data, is aggravated by the innate data size blowup, each individual data element gets expanded into matrices of much larger dimensions depending on the required security. Moreover, this is without accounting for the practical constraint of unrolling one's desired program into an FHE compatible circuit which involves reprogramming the entire computation. All of these practical constraints would make it seem that today FHE isn't practically feasible today and is unlikely to be so in the recent future.

[1] Compactness is a property expected of FHE schemes today, which states that ciphertext size must be independent of the circuit depth

6.3 Concluding remarks

Secure computation is an ever growing area of interest. Over the last decade, we have envisioned and seen the rise of several interesting applications for it. However, the techniques we see today are limited either computational capabilities, or are not efficient enough to be practical. Intel SGX offers a new hope of efficient secure computation, however it has security gaps that need to be patched.

In this work, we designed and implemented ZeroTrace, a secure memory controller that can assure security guarantees for sensitive data while working with Intel SGX for secure computation. ZeroTrace is a stepping stone to achieving practically viable, scalable and secure computation via Intel SGX.

Bibliography

[1] Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing.

[3] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[4] M. Backes, A. Kate, M. Maffei, and K. Pecina. Obliviad: Provably Secure and Practical Online Behavioral Advertising. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 257–271, 2012.

[5] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 307–314. ACM, 1968.

[6] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, page 8, 2015.

[7] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 326–349. ACM, 2012.

[8] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. 2011.

[9] K. D. Bowers, A. Juels, and A. Oprea. Proofs of Retrievability: Theory and Implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, pages 43–54, 2009.

[10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. *CoRR*, 2017.

[12] Calomel.org. AES-NI SSL Performance.

[13] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 668–679, 2015.

[14] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017.

[15] V. Costan and S. Devadas. Intel SGX Explained, 2016.

[16] I. Damgård, S. Meldgaard, and J. Nielsen. Perfectly Secure Oblivious RAM Without Random Oracles. *Theory of Cryptography*, pages 144–163, 2011.

[17] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. In *Theory of Cryptography Conference*, pages 145–174, 2016.

[18] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU System Management Mode to Circumvent Operating System Security Functions.

[19] E. S. M. L. Elaine Shi, Hubert Chan. Oblivious RAM with O((log N)3) Worst-Case Cost. Cryptology ePrint Archive, Report 2011/407, 2011. http://eprint.iacr.org/2011/407.

[20] S. Embleton, S. Sparks, and C. C. Zou. SMM Rootkit: A New Breed of OS Independent Malware. *Security and Communication Networks*, 6(12):1590–1605, 2013.

[21] J. Fan and F. Vercauteren. Somewhat Practical Fully Homomorphic Encryption.

[22] C. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov. Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM. Technical report.

[23] C. W. Fletcher, M. v. Dijk, and S. Devadas. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, pages 3–8, New York, NY, USA, 2012. ACM.

[24] C. Gentry. Fully Homomorphic Encryption using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[25] C. Gentry, A. Sahai, and B. Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.

[26] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)*, pages 431–473, 1996.

[27] M. T. Goodrich and M. Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In *International Colloquium on Automata, Languages, and Programming*, pages 576–587. Springer, 2011.

[28] S. Gorbunov, V. Vaikuntanathan, and D. Wichs. Leveled Fully Homomorphic Signatures from Standard Lattices. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 469–477. ACM, 2015.

[29] D. Gullasch, E. Bangerter, and S. Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 490–505, 2011.

[30] S. Halevi. HElib.

[31] S. Halevi and V. Shoup. Algorithms in HElib. In *International Cryptology Conference*, pages 554–571. Springer, 2014.

[32] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 533–549, 2016.

[33] Intel. Intel Trusted Execution Technology. http://www.intel.com/technology/security/, 2007.

[34] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A Fast, Cross-VM Attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319, 2014.

[35] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in cryptologyCRYPTO99*, pages 789–789, 1999.

[36] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (In) security of Hash-Based Oblivious RAM and a New Balancing Scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 143–156. Society for Industrial and Applied Mathematics, 2012.

[37] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 205–221, New York, NY, USA, 2017. ACM.

[38] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-Grained Control Flow Inside SGX enclaves with Branch Shadowing. *arXiv preprint arXiv:1611.06952*, 2016.

[39] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A Hardware-Software System for Memory Trace Oblivious Computation. *ACM SIGARCH Computer Architecture News*, pages 87–101, 2015.

[40] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622, 2015.

[41] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 311–324, New York, NY, USA, 2013. ACM.

[42] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback Protection for Trusted Execution, 2017.

[43] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for Tcb Minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, Apr. 2008.

[44] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel&Reg; Software Guard Extensions (Intel&Reg; SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 10:1–10:9, 2016.

[45] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel&Reg; Software Guard Extensions (Intel&Reg; SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 10:1–10:9, 2016.

[46] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10:1–10:1, 2013.

[47] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *International Conference on Information Security and Cryptology*, pages 156–168, 2005.

[48] M. Naveed, S. Kamara, and C. V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[49] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors.

[50] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers Track at the RSA Conference*, pages 1–20, 2006.

[51] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 431–446, 2015.

[52] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas. Constants count: Practical Improvements to Oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, 2015.

[53] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *Proceedings of the*

24th USENIX Conference on Security Symposium, SEC'15, pages 415–430, Berkeley, CA, USA, 2015. USENIX Association.

[54] L. Ren, C. W. Fletcher, X. Yu, M. Van Dijk, and S. Devadas. Integrity verification for path oblivious-ram. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, 2013.

[55] M. Russinovich. Homomorphic Encryption Standardizaton Workshop.

[56] J. Rutkowska and A. Tereshkin. Bluepilling the Xen Hypervisor.

[57] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using SGX to conceal cache attacks. 2017.

[58] N. Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic. Spectral profiling: Observer-effect-free profiling by monitoring EM emanations. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–11, 2016.

[59] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with O ((logN) 3) Worst-Case Cost. In *Asiacrypt*, volume 7073, pages 197–214. Springer.

[60] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o ((logn) 3) worst-case cost. In *International Conference on The Theory and Application of Cryptology and Information Security*, pages 197–214, 2011.

[61] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[62] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328, 2016.

[63] S. Shinde, D. L. Tien, S. Tople, , and P. Saxena. PANOPLY: Low-TCB linux applications with sgx enclaves. In *NDSS*, 2017.

[64] E. Stefanov. Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns.

[65] E. Stefanov and E. Shi. Oblivistore: High Performance Oblivious Cloud Storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267, 2013.

[66] E. Stefanov and E. Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 253–267, Washington, DC, USA, 2013. IEEE Computer Society.

[67] E. Stefanov, E. Shi, and D. Song. Towards Practical Oblivious RAM. *arXiv preprint arXiv:1106.3652*, 2011.

[68] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310, 2013.

[69] G. E. Suh, C. W. O'Donnell, and S. Devadas. AEGIS: A single-chip secure processor. *Information Security Technical Report*, 10(2):63–73, 2005.

[70] E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES and Countermeasures. *Journal of Cryptology*, pages 37–71, 2010.

[71] Trusted Computing Group. Trusted Computing Platform Alliance (TCPA) Main Specification Version 1.1b. https://www.trustedcomputinggroup.org/specs/TPM/TCPA_Main_TCG_Architecture_v1_1b.pdf, 2003.

[72] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 9:1–9:14, 2014.

[73] J. van de Pol, N. P. Smart, and Y. Yarom. Just a Little Bit More. In *Cryptographers Track at the RSA Conference*, pages 3–21, 2015.

[74] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861, 2015.

[75] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226, 2014.

[76] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[77] P. Williams and R. Sion. Single Round Access Privacy on Outsourced Storage. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 293–304, 2012.

[78] P. Williams, R. Sion, and B. Carbunar. Building Castles Out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 139–148. ACM, 2008.

[79] R. Wojtczuk and J. Rutkowska. Attacking Intel Trusted Execution Technology.

[80] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning.

[81] R. Wojtczuk and A. Tereshkin. Attacking Intel Bios.

[82] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[83] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. A2: Analog Malicious Hardware. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 18–37, 2016.

[84] Y. Yarom and K. Falkner. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 719–732, 2014.

[85] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[86] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2, 2012.

[87] S. Zahur and D. Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 493–507, Washington, DC, USA, 2013. IEEE Computer Society.

[88] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

[89] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, 2017.