ZeroTrace : **Oblivious Memory Primitives from Intel SGX**

Sajin Sasy
University of Waterloo
sajin.sasy@gmail.com

Sergey Gorbunov
University of Waterloo
sergey.gorbunov@uwaterloo.ca

Christopher Fletcher
Nvidia
chris.w.fletcher@gmail.com

1 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-structure 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. These works have disadvantages by themselves. 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.

We combine these two lines of work by designing a working prototype library of oblivious memory primitives, which we call ZeroTrace, on top of SGX. 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 6 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. Finally, we demonstrate how to re-parameterize our system for the remote oblivious storage setting, where we can service a 4 KB request in 267 ms, at less than an order of magnitude WAN bandwidth overhead.

2 INTRODUCTION

Cloud computing is a paradigm, ever growing in popularity, that offers on-demand compute and storage resources for users. Applications such as machine learning, AI, analytics, web, and mobile services are now frequently hosted in public clouds. Protecting users' data in these environments is challenging due to their underlying complexity and shared infrastructure model. As a result, multiple attacks vectors from infrastructure and service providers, other users, and targeted adversaries remain open.

Up until recently, secure cloud computing could only be achieved through cryptography (e.g., fully homomorphic encryption – FHE [11]), or through course-grain hardware isolation techniques (e.g., Intel TPM+TXT [16, 25, 47]). Both of the above have severe performance and usability limitations. FHE introduces many orders of magnitude overheads. 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.

Recently, Intel released an instruction set extension called *Software Guard Extensions* (SGX) which addresses the above challenges [8, 26–28]. 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 was 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.

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 [41, 61]. 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 [2, 4, 15, 48]. 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.

Our Approach. We address this challenge 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 [5, 12, 51, 59]. 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 read(addr) and write(addr, 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 it is issued by the user and the arguments issued to those commands. That is, it runs *obliviously* [12, 29, 30, 32]. 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. First, 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., [29, 30, 32]) to prevent these leakages. Second, SGX enclaves do not support direct IO to disk. To support disk backend storage, we partition the controller between trusted and un-trusted zones in a secure fashion. Third, 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.*

Contributions. To summarize, this paper 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: remote file server (see above) and plug-and-play data-structures. In the remote file server setting, ZeroTrace can make oblivious read and write calls at 4 KB granularity on a 1 TB dataset in 267 ms. It can also make oblivious read and write calls to 4 B/1 KB memory locations on a 10 GB dataset in 1.2/6 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.

Paper Organization. In Section 3, we describe our usage model and security models. Section 4 gives a background on Intel SGX and ORAM. Section 5 gives we give details on our architecture; including the instantiation process, client and server components, optimizations and security analysis. Section 6 gives a scheme to achieve persistant integrity and fault tolerance. Section 7 describes our prototype implementation and evaluation. Section 8 gives related work, and finally Section 9 concludes.

3 OUR MODEL

3.1 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.

As introduced in Section 2, 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 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 [5]). A similar argument goes for memory services exposing a data-structure interface. For example, Wang et al. [51]

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 Google's NaCl [57]). Data structures and memory controllers naturally have narrow interfaces (compared to more general interfaces, such as POSIX [41]), 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.

(Special case) 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 approach is for the client to use an Oblivious RAM protocol where the client runs the ORAM controller locally [42, 53]. 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.

3.2 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 – thus, our focus for much of the paper – 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 2), which are dealt with via oblivious execution. (Related work calls these *digital side-channels* [32].) 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 4.3.

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 [6, 36] and how/when the enclave suffers page faults [54]. Our system has mechanisms to preserve privacy and integrity despite the above vulnerabilities. We formalize this security guarantee in Section 4.1 and map SGX to these definitions in Section 4.2.

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 [24], and risks irreverable data-loss. We develop novel protocols in Section 6 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[18] or EM emissions [37]), compromised manufacturing (e.g., hardware trojans [55]) or denial of service attacks.

4 PRELIMINARIES

4.1 Oblivious Enclave Execution

We now formalize oblivious execution for enclaves that that we set out to achieve in our system. We first give a general definition for enclave-based trusted 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.

$\mathsf{Execute}(\mathsf{E_P}, \mathsf{in}) \rightarrow (\mathsf{out}, \psi)$. 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 4.2.1, below.

Security. When a program P is loaded in an enclave, and a set of inputs $\overrightarrow{y} := (\mathsf{in}_M, ..., \mathsf{in}_1)$ are executed by this enclave, it results in an adversarial view $V(\overrightarrow{y}) := (\mathsf{trace}_{(\mathsf{E_P},\mathsf{in}_M)}, ..., \mathsf{trace}_{(\mathsf{E_P},\mathsf{in}_1)})$. We say that an enclave execution is oblivious, if given two sets of inputs $\overrightarrow{y}$ and $\overrightarrow{z}$, their adversarial views $V(\overrightarrow{y})$ and $V(\overrightarrow{z})$ are computationally indistinguishable to anyone but the client.

4.2 Intel SGX

In this section we give a brief introduction to Intel Software Guard Extensions (SGX) and highlight aspects relavent to ZeroTrace. (See [1, 8] give 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 IO/communications interface between them. In SGX, security is bootstrapped from an underlying trusted processor, not trust in a remote software stack. We now describe how Intel SGX implements the Load(P) and $\mathsf{Execute}(\mathsf{E_P}, \mathsf{in})$ functions from the previous section.

$\mathsf{Load}(\mathsf{P}) \rightarrow (\mathsf{E_P}, \phi)$. 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.

$\mathsf{Execute}(\mathsf{E_P}, \mathsf{in}) \rightarrow (\mathsf{out}, \psi)$. SGX protects enclave program execution by isolating enclave code and data in Processor Reserved Memory (PRM), referred as Enclave Page Cache (EPC), which is a subset of DRAM that gets set aside securely at boot time. 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. Cryptographic keys for these operations are owned by the trusted processor. 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 IO. 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 [41].

4.2.1 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 demonstrated attacks that extract sensitive data through hardware resource pressure (e.g., cache [6, 36] and branch predictor [20]) and the application's page-level access pattern [54].

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 [24]. 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.

4.2.2 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 (Figure 3 in [2]).

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.

4.3 ORAM

We now describe the popular definition for ORAM from the literature [44, 45]. Afterwards, we provide additional details of the Path ORAM scheme, used in our system [45].

An ORAM scheme can be used to store and retrieve blocks of memory on a remove server, such that the server learns nothing about the data access patterns. Informally, no information should be leaked about: (a) the data being accessed, (b) whether the same/different data is being accessed relative to a prior access (linkability), (c) whether the access is a read or write.

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

Security. Let

$$\overrightarrow{y} := ((\mathsf{op}_M, \mathsf{a}_M, \mathsf{data}_M), \ldots, (\mathsf{op}_1, \mathsf{a}_1, \mathsf{data}_1))$$

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

4.4 Path ORAM

We now give a summary of Path ORAM [45], the ORAM used in our current implementation. Which ORAM is used isn't fundamental, and this can be switched behind the memory controller interface. That said, ORAM bandwidth to untrusted storage and ORAM controller trusted storage are inversely proportional [44, 45, 50]. Further, the SGX and oblivious settings decrease performance when using larger controller storage (due to EPC evictions [24] and the cost of running oblivious programs; see Section 7). Path ORAM provides a middle ground here: better bandwidth/larger storage than [50]; worse bandwidth/smaller storage than [44].

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 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 $\omega(\log N)$ [45]. 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.

Operation. 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, to perform a read/write request to block a (mapped to leaf l), we perform the following steps: First, 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. Second, Fetch the entire path from the root to leaf bucket in server storage. Third, retrieve the block from the combination of the fetched path and the local stash. Fourth, 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 $O(N)$ 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.[44] and Shi et al. [38]. In short, the idea is to store the position map itself as a smaller ORAM on the server side and then recurse. Each smaller "position map" ORAM must be accessed in turn, to retrieve the leaf label for the original ORAM.

Extension: Integrity. Path ORAM assumes a passive adversary by default. To provide an integrity guarantee with freshness, one can construct a Merkle tree mirrored [45] 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* [35].

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

5 ZEROTRACE **MEMORY CONTROLLER**

We now describe how the core memory controller is implemented on the server. We focus on supporting our strongest level of security: obliviousness against an active adversary (Section 3.2). The entire system is shown in Fig. 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.

5.1 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 4.2.1), we will design the ORAM controller to run as an *oblivious program.* (A similar approach is used to guard against software side channels by Olga

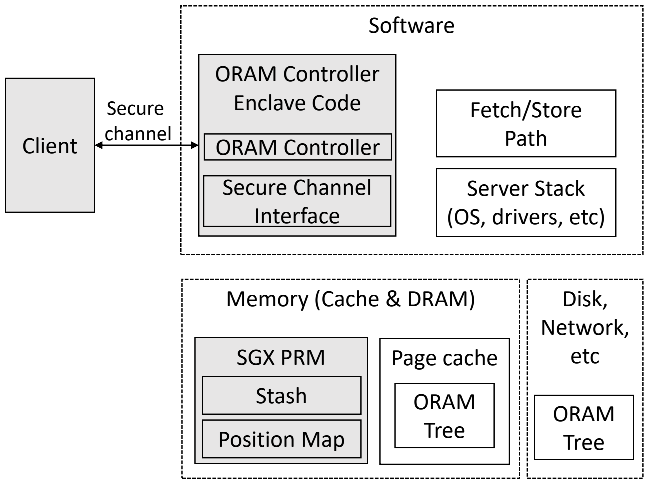

Figure 1: System components on the server. Trusted components (software and regions of memory) are shaded. Depending on the setting, the client may be connecting from a remote device (not on the server) or from another enclave on the same machine.

et al.[30] and Rane et al.[32].) 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 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 4.2), incurs large EPC paging overheads (Section 4.2.2) 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 [10, 33, 50]) 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 5.5 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 ~ 95 MB), untrusted in-memory code (which has access to DRAM ~ 64 GB) 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 [2]. 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).

5.2 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 4.4).

5.3 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 5.5). Since Fetch/Store path are not in the TCB, these optimizations do not effect security.

5.4 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.

5.4.1 Initialization. For initialization, the server performs the function Load(P) → (E_P, ϕ), 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 $\mathsf{K_e}$ from an asymmetric key pair $(\mathsf{K_e}, \mathsf{K_d})$ sampled within the enclave. The client encrypts a secret key K under this public key $\mathsf{K_e}$ for the enclave. The user and enclave henceforth communicate using this K for an authenticated encrypted channel.

5.4.2 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 [30, 32]. 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 `dst` to a memory location by default. Additional instructions (not shown) are needed to move the result of a register `dst` CMOV to memory.

5.4.3 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 (e.g., Fetch/Store path) may perform arbitrary syscalls without impacting the TCB.

5.4.4 Building Block: Encryption & Cryptographic Hashing. Our implementation relies on encryption and integrity checking via cryptographic hashing in several 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 re-generated by Fetch/Store Path (Section 5.3) need to be decrypted/re-encrypted and integrity verified. These routines must also be oblivious. 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: SGX has separate encryption/hashing mechanisms to ensure privacy/integrity of pages evicted from the EPC [8]. Since our design accesses ORAM through a Fetch/Store Path interface, we cannot use these SGX built-in mechanisms for ORAM privacy/integrity.

5.4.5 ORAM Controller. The ORAM Controller handles client queries of the form (op, id, data∗), where op is the mode of operation, i.e. read or write, id corresponds to an identifier of the data element and 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 5.4.1) 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 moniter 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 [32].

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 an access request (op, id, data∗), it must read and update a location in the position map (Section 4.4) using oupdate calls, as shown in the pseudocode below.

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

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 6, 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 4.4) using a Fetch Path call (Section 5.3). 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 [45]. 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 [23, 45] 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 4.4) using internal logic and a Store Path call (Section 5.3). 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 4.4. 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 5.4.4. At the same time, an oblivious implementation of the Merkle tree (Section 4.3) checks and is re-built to verify integrity with freshness.

5.5 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 Map(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 [23, 33]. 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 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 allocate regular process memory to store the top portion, and explicitly store the lower portion behind disk IO calls.

5.6 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:

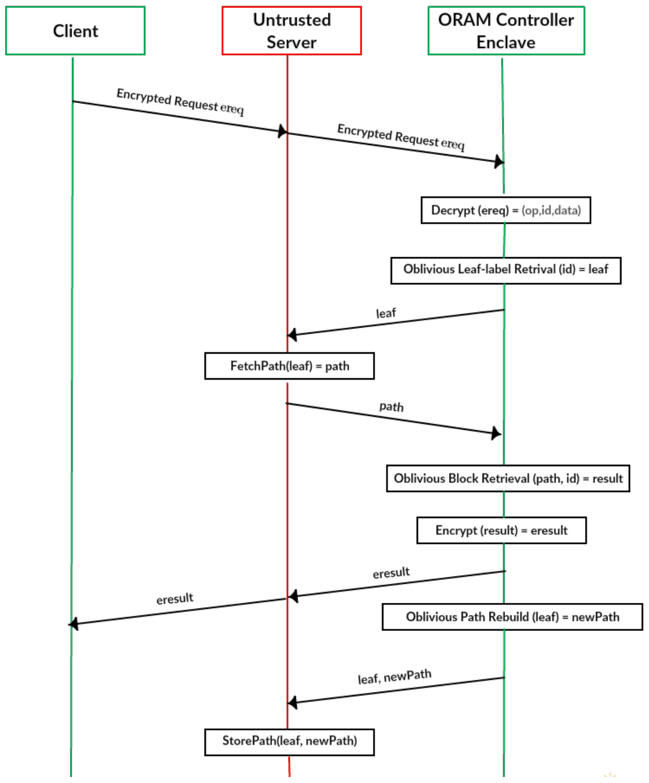
Client
Untrusted Server
ORAM Controller Enclave
Encrypted Request ereq
Encrypted Request ereq
Decrypt (ereq) = (op,id,data)
Oblivious Leaf-label Retrival (id) = leaf
leaf
FetchPath(leaf) = path
path
Oblivious Block Retrieval (path, id) = result
Encrypt (result) = eresult
eresult
eresult
Oblivious Path Rebuild (leaf) = newPath
leaf, newPath
StorePath(leaf, newPath)

Figure 2: Execution of an access request

Theorem 5.1. *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 4.3.*

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 2.

Claim 5.1.1. *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 4.2) 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 5.1.2. *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 5.1.3. *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 5.4.2) 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 5.1.4. 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 4.4).

Claim 5.1.5. *Verifying fetched path leaks no information*

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 [45]. 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 5.1.6. *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 5.4.5. Since the length of `path` and `stash` are data-dependent, obliviousness reduces to the security of oupdate (see Claim 5.1.3).

Claim 5.1.7. *Oblivious Rebuild leaks no information*

Same argument as Claim 5.1.6, since `new_path`, `bu` and `stash` have data independent size.

Claim 5.1.8. 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.

6 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 (i.e., all of Section 5).

Definition 6.1 (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 [58, 61]. 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 3.2, 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 4.1). In our setting, we must analyze two places in the protocol. First, the path written back to untrusted storage after each request (Section 5.4.5) 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 5.3) must be *deterministic* with respect to previous requests. This property is achieved by re-assigning leaf labels using a pseudo-random number generator.

7 IMPLEMENTATION AND EVALUATION

7.1 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 3.1). 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).

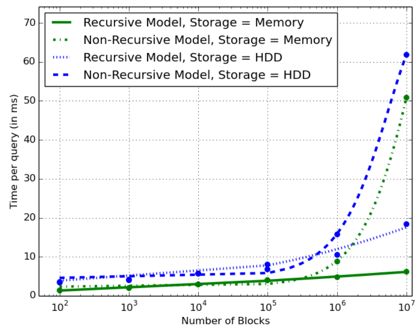

Figure 3: Representative result. Shows the number of data blocks vs. time per request, with data blocks of size 1 KB.

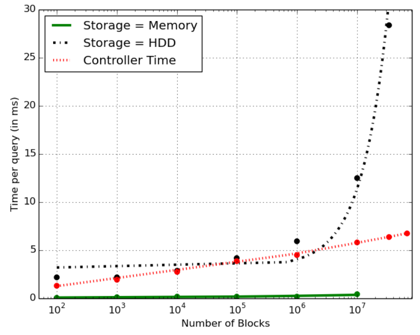

Figure 4: Detailed performance breakdown. Shows the number of data blocks vs. time spent in different parts of the request, with different storage backends, with a block size of 8 Bytes. Total time per request is the sum of controller and storage (DRAM or HDD) times. DRAM is shown until 10^7 blocks, which is our DRAM capacity.

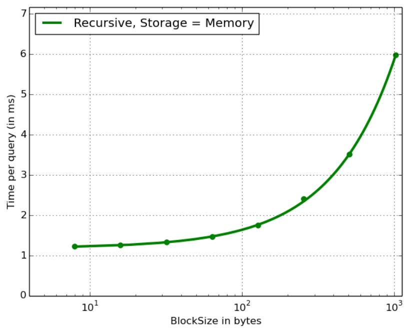

Figure 5: Performance as a function of data ORAM block size for a dataset with 10^7 blocks.

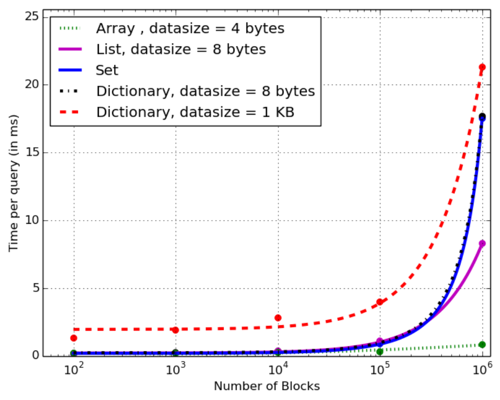

Figure 6: Evaluation of our oblivious memory controller library for Set/Dictionary/List/Array. Array is a direct call to our core memory controller, which uses ORAM recursion to be asymptotically efficient.

7.2 Evaluation of our 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 3.2). Figure 3 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 points using the ORAM recursion technique, we use a position map of size 400 KB within the EPC pages 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.

Figure 5 shows the controller request time varying the data ORAM block size. For data ORAM block sizes, the curve is flat for ORAM because the cost of recursion dominates. Comparing to Figure 3, we see recursive ORAM pays off for large datasets. This matches the theory [45] 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 4. The main results are that the oblivious controller is the bottleneck given fast untrusted storage devices (e.g., DRAM). We will discuss this issue further in Section 7.4. 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. We show the effect for completeness.

7.3 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:

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

$\mathsf{Access}(\mathsf{op}, \mathsf{req})$. 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.

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 $\mathsf{read}(\mathsf{addr})$ and $\mathsf{write}(\mathsf{addr}, \mathsf{data})$. Sets support the operations $\mathsf{insert}(\mathsf{data})$, $\mathsf{delete}(\mathsf{data})$ and $\mathsf{contains}(\mathsf{data})$. Dictionaries support $\mathsf{put}(\mathsf{tag}, \mathsf{data})$ and $\mathsf{get}(\mathsf{tag})$. Lists support $\mathsf{insert}(\mathsf{index}, \mathsf{data})$ and $\mathsf{remove}(\mathsf{index})$. 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 6 performance figures. 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. [52], which makes use of ORAMs and can use our core memory controller as a primitive as well.

7.4 Evaluation as Oblivious Remote Storage

We now re-parameterize and compare our system to Oblivistore [43], a state-of-the-art system the oblivious file storage setting (see Section 3.1). Our HDD is very similar to the HDD used in [43], thus we compare our system to theirs using HDDs. We assume a 50 ms one-way latency WAN connection between client and server.

Specifically, we compare to Oblivistore using their 1 TB ORAM capacity, 4 KB block size, 7 HDD experiment (Section VII.C and Table 4 in [43]). In that case, they achieve 364 KB/s client bandwidth with a response time of 196 ms, at the cost of 40-50× client-server and server-disk bandwidth overhead. By "client bandwidth" we mean the amount of client data fetched by the ORAM protocol. Client-server and server-disk bandwidth overheads refer to the amount of communication over the LAN/WAN and to the server disks.

Our current infrastructure only supports a single 500 GB disk HDD. We normalize Oblivistore to a single 1 TB disk, and extrapolate our own bandwidth to a 1 TB setting. In that case, their system achieves 52 KB/s client bandwidth at the same response latency and client-server/server-disk bandwidth overhead. (Oblivistore parallelizes disks across requests; thus reducing the number of disks doesn't change a single request's latency.) In this configuration, our system achieves 5.9 KB/s client bandwidth with a 267 ms response latency and **2× client-server bandwidth**

overhead and a 184× server-disk bandwidth overhead. Here, 184× is amount of data fetched from untrusted DRAM/disk to service a recursive Path ORAM access.

The main take-away is our system has a competitive response latency with an ∼ 22.5× **drop in client-server bandwidth overhead** (the dominant bandwidth cost). The area to improve in future work is our memory controller's client-server bandwidth (52 KB/s vs. 5.9 KB/s). The drop in bandwidth is due to our use of Path ORAM 184× bandwidth vs. 40-50× bandwidth and the overhead of running an oblivious controller (which takes ∼ 30% of our request time).

We make two additional remarks. First, our current infrastructure cannot support multiple HDDs, whereas Oblivistore is evaluated at 7 HDDs. In Section 5.5, we present an extension that allows our disk bandwidth to scale with additional disks, *for a single* ZeroTrace *node*. Thus, we expect a similar result as seen for 1 HDD, with 7 HDDs. Second, Oblivistore bandwidth decreases when SSDs are used. Our bandwidth is bottlenecked with SSDs due to the $O(\log^2 N)$ time oblivious code in Section 5. This will be improved by switching our ORAM implementation to Circuit ORAM [50] (see Section 4.4), which increases server-disk bandwidth by ∼ 50% but reduces oblivious code runtime to $O(\log N)$ (a factor of ∼ 80× with our parameters).

8 RELATED WORK

Our work is the first demonstration of a completely oblivious data structures library built on a real secure hardware platform. For this project, we rely on research in several foundational areas:

Oblivious RAMs and Secure Hardware. Research in ORAM began with the seminal work by Goldreich and Ostrovsky [13], and has culminated in practical constructions with logarithmic bandwidth overhead [34, 45, 50]. 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 by combining homomorphic encryption with ORAM [9], and by the ORAM-based systems Oblivistore [42] and ObliviAd [3] (which assume hypothetical secure hardware). The latter two works have a weaker threat model than this paper: our goal is to protect against all remote software attacks, whereas the latter two focus only on hiding ORAM protocol-level access patterns.

Systems. A number of systems investigate the question of protecting applications running in enclaves. Raccoon [32] 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, [32]). In our design, obliviousness is guaranteed even when an adversary compromises the entire software stack including the OS. Finally, while Racoon can run on an Intel SGX-enabled processor, the architectural limitations of SGX are not taken into consideration in their design.

GhostRider [21] 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, their assume "unbounded resources, and no caching" and do not target any modern processor (Introduction, [21]). In contract, the focus of this work is to design a real-world system capable of running on a widely available Intel CPU architecture.

T-SGX is a system that helps to protect against controlled-channel attacks within enclaves [39]. SGXbounds is a memory-safety system for shielded execution within Intel SGX [19]. Opaque 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 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 (e.g., a cache [14, 17, 22, 31, 46, 49, 56, 60] or branch predictor [20]) can be detected by an adversary and can reveal fine-grain details about program execution. In SGX-based systems, there is an arms race currently underway between defenses (e.g., T-SGX[39] and Deja Vu[7]) and new attacks (e.g., Brasser et al.[6]) related to shared resource usage. Similarly, a malicious OS can monitor application page fault behavior to learn program memory access patterns [40, 54]. ZeroTrace protects against all shared resource and page fault-related attacks by converting the program to an oblivious representation.

9 CONCLUSION

This paper designs and implements ZeroTrace, the first library of oblivious memory primitives for a real secure hardware platform, optimized for Intel's SGX. Our work argues for building applications out of modules at the memory-service interface level. We provide several oblivious memory services, the core block being an oblivious block-level memory controller that can defend against software attacks from an active adversary. While these services can be connected directly to co-located applications in the cloud, we also show how they can be used to implement remote file storage systems – granting constant WAN bandwidth overhead solutions at the expense of trusting the SGX mechanism.

REFERENCES

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 689–703.

[3] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. 2012. Obliviad: Provably secure and practical online behavioral advertising. In *Security and Privacy (SP), 2012 IEEE Symposium on*. 257–271.

[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* (2015), 8.

[5] Kevin D. Bowers, Ari Juels, and Alina Oprea. 2009. Proofs of Retrievability: Theory and Implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. 43–54.

[6] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. *CoRR* (2017).

[7] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. 2017. 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*. 7–18.

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. (2016).

[9] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*. 145–174.

[10] Emil Stefanov Mingfei Li Elaine Shi, Hubert Chan. 2011. Oblivious RAM with O((log N)3) Worst-Case Cost. Cryptology ePrint Archive, Report 2011/407. (2011). http://eprint.iacr.org/2011/407.

[11] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC '09)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[12] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* (1996), 431–473.

[13] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* (1996), 431–473.

[14] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–bringing access-based cache attacks on AES to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*. 490–505.

[15] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 533–549.

[16] Intel. 2007. Intel Trusted Execution Technology. http://www.intel.com/technology/security/. (2007).

[17] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! a fast, cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*. 299–319.

[18] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Advances in cryptologyâĂŤCRYPTOâĂŽ99*. 789–789.

[19] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 205–221. https://doi.org/10.1145/3064176.3064192

[20] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2016. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952* (2016).

[21] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGARCH Computer Architecture News* (2015), 87–101.

[22] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*. 605–622.

[23] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 311–324.

[24] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. (2017).

[25] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An Execution Infrastructure for Tcb Minimization. *SIGOPS Oper. Syst. Rev.* 42, 4 (April 2008), 315–328. https://doi.org/10.1145/1357010.1352625

[26] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. 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*. 10:1–10:9.

[27] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. 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*. 10:1–10:9.

[28] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*. 10:1–10:1.

[29] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*. 156–168.

[30] Olga Ohrimenko, Felix Schuster, Cdric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors.

[31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *CryptographersâĂŽ Track at the RSA Conference*. 1–20.

[32] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *Proceedings of the 24th USENIX Conference on Security Symposium*. 431–446.

[33] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 415–430. http://dl.acm.org/citation.cfm?id=2831143.2831170

[34] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*. 415–430.

[35] Ling Ren, Christopher W Fletcher, Xiangyao Yu, Marten Van Dijk, and Srinivas Devadas. 2013. Integrity verification for path oblivious-ram. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. 1–6.

[36] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. (2017).

[37] Nader Sehatbakhsh, Alireza Nazari, Alenka Zajic, and Milos Prvulovic. 2016. Spectral profiling: Observer-effect-free profiling by monitoring EM emanations. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. 1–11.

[38] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O ((logN) 3) worst-case cost. In *International Conference on The Theory and Application of Cryptology and Information Security*. 197–214.

[39] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*.

[40] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 317–328.

[41] Shweta Shinde, Dat Le Tien, Shruti Tople, , and Prateek Saxena. 2017. PANOPLY: Low-TCB linux applications with sgx enclaves. In *NDSS*.

[42] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*. 253–267.

[43] Emil Stefanov and Elaine Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 253–267. https://doi.org/10.1109/SP.2013.25

[44] Emil Stefanov, Elaine Shi, and Dawn Song. 2011. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652* (2011).

[45] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 299–310.

[46] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* (2010), 37–71.

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

[48] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems*. 9:1–9:14.

[49] Joop van de Pol, Nigel P Smart, and Yuval Yarom. 2015. Just a little bit more. In *CryptographersâĂŽ Track at the RSA Conference*. 3–21.

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

[51] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 215–226.

[52] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. Cryptology ePrint Archive, Report 2014/185. (2014). http://eprint.iacr.org/2014/185.

[53] Peter Williams and Radu Sion. 2012. Single Round Access Privacy on Outsourced Storage. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 293–304.

[54] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. 640–656.

[55] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. 2016. A2: Analog malicious hardware. In *Security and Privacy (SP), 2016 IEEE Symposium on*. 18–37.

[56] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX conference on Security Symposium*. 719–732.

[57] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, Washington, DC, USA, 79–93. https://doi.org/10.1109/SP.2009.25

[58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. 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*. 2–2.

[59] Samee Zahur and David Evans. 2013. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 493–507. https://doi.org/10.1109/SP.2013.40

[60] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 990–1003.

[61] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.