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 W. Fletcher
NVIDIA/UIUC
cwfletch@illinois.edu

***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-the-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 speed-up 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 B 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).**

I. 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 attack 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 [12]), or through course-grained hardware isolation techniques (e.g., Intel TPM+TXT [18], [26], [14]). Both of the above have severe performance and usability limitations. FHE, for example, introduces many orders of magnitude overheads. On the other hand, these techniques provide very strong security guarantees (stronger than TPM+TXT) needed for applications that operate over highly sensitive data (e.g., federal, military, government data, etc.). They can be used to protect even against malicious operating systems snooping on the data access-pattern.

Recently, Intel released an instruction set extension called *Software Guard Extensions* (SGX) which addresses the above performance challenges [9], [27], [28]. In SGX, user-level sensitive portions of ring-3 applications can be run in one or more application containers called enclaves. While running, SGX uses a set of hardware mechanisms to preserve the privacy and integrity of enclave memory. However, using SGX to achieve whole-program privacy against software adversaries still faces multiple challenges. First, the user must map its application to enclave(s) in a way that gives a favorable trade-off in trusted computing base (TCB) size, performance and code isolation. The default approach, natively supported by Intel SGX, is to manually partition the application into trusted and untrusted code [40], [59]. This is non-trivial and must be done sparingly: code within enclaves is trusted and enclaves have limited functionality (e.g., no support for IO/syscalls and a bounded memory size). Alternatively, a number of works study how to load unmodified applications into enclaves [2], [4], [17], [46]. While more automated, these approaches induce a larger TCB. Second, the user must carefully write enclave code to avoid numerous software side-channels [6], [21], [35], [48], [53]. Taken together, leveraging SGX to achieve small TCB and side-channel free trusted execution environments remains an open problem.

A. This Work

We address this challenge by designing and implementing ZeroTrace – an oblivious library enabling applications to be built out of fine-grained building-blocks at the application's data-structure interface boundary. Any operation on the data stored by the library is protected using SGX enclaves and remains secure against all software attacks, including all known side-channels.

Partitioning applications at the oblivious data-structure boundary hits a sweet spot for several reasons. First, the data-structure interface is narrow, which makes it easier to sanitize application to data-structure requests—improving intra-

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23239
www.ndss-symposium.org

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. Lastly, each data-structure can seamlessly support multiple clients and can be oblivious to where each client is physically running. For the latter point, clients can attach to data-structure enclaves remotely, providing performance improvements to related systems (e.g., oblivious file servers; Section II-A2).

As part of this research, *we implement and evaluate the first oblivious memory controller running on a real secure hardware platform.* Our memory controller, which implements an Oblivious RAM (ORAM) protocol [13], can be called as a subroutine in a larger application and defends against any active software adversary. A key insight that drives our design is that with SGX, ORAM state (both untrusted storage and trusted ORAM client logic) *can safely live anywhere in the system* (e.g., cache, DRAM, disk, etc), even outside the SGX enclave, despite the adversary running concurrent to the victim and controlling the software stack. For data inside enclaves, the SGX mechanism prevents direct inspection of data. Thus security against software attacks reduces to accessing in-enclave data in a data oblivious fashion [13], [29], [30], [32]. For data living outside enclaves, enclave code can add a second layer of protection (via encryption, integrity checks, etc) to securely extend the ORAM algorithm working set as needed.

B. Contributions

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. We provide two implementations, one using Path ORAM [43] and one using Circuit ORAM [49] and compare both across multiple backend memory organizations (DRAM and HDD). All designs protect against an active software adversary and provide secure fault-tolerance across asynchronous SGX enclave terminations (a common challenge for SGX applications). These extensions may be of independent interest.

2) We design and implement ZeroTrace, an application library for serving data-structures obliviously in an SGX environment. In this paper, ZeroTrace's core primitive is the above oblivious memory controller.

3) We evaluate system performance for ZeroTrace as a stand alone oblivious memory controller and for plug-and-play data structures on an SGX-enabled Dell Optiflex 7040. Our system can make oblivious read and write calls to 1 KB memory locations on a 10 GB dataset in 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.

Our design is open source and available at https://github.com/ssasy/ZeroTrace.

C. Paper Organization

In Section II, we describe our usage and security models. Section III gives a required background on Intel SGX and ORAM. In section IV we give details on our architecture; including the instantiation process, client and server components, optimizations and security analysis. Section V gives a scheme to achieve persistent integrity and fault tolerance. Section VI describes our prototype implementation and evaluation. Section VII gives related work, and finally Section VIII concludes.

II. Our Model

A. 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 Extensions (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 I, 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.

1) 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. [50] 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 by Google's NaCl [56]). Data structures and memory controllers naturally have narrow interfaces (compared to more general interfaces, such as POSIX [40]), 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-communication between enclaves gives flexibility in deployment, as shown in the next paragraph.

2) (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 [41], [52]. The ORAM controller interacts over the network with the server, which acts as a disk. While benefiting 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 in typical parametrizations. 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.

B. 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 software 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 implements an ORAM protocol. We default to this level of security because a known limitation of SGX is its software-based side-channel leakages (Section I), which are dealt with via data oblivious execution. (Related work calls these *digital side-channels* [32].) Data 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:

1) Security of memory: First, the memory accesses made by the SGX enclave to any memory outside the enclave. 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 III-C.

2) 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], [35] and how/when the enclave suffers page faults [53]. Our system has mechanisms to preserve privacy and integrity despite the above vulnerabilities. We formalize this security guarantee in Section III-A and map SGX to these definitions in Section III-B.

3) 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 [25], and risks irreversible data-loss. We develop novel protocols in Section V to make the ORAM+enclave system fault tolerant and secure against arbitrary enclave terminations.

4) Security non-goals: We do not defend against hardware attacks (e.g., power analysis [20] or EM emissions [36]), compromised manufacturing (e.g., hardware trojans [54]) or denial of service attacks.

III. Preliminaries

A. 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 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.

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

2) $\mathsf{Execute}(\mathsf{E_P}, \mathsf{in}) \rightarrow (\mathsf{out}, \psi)$: 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 proof 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 III-B5, below.

3) 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 $\mathsf{V}(\overrightarrow{y}) := (\mathsf{trace}_{(\mathsf{E_P},\mathsf{in_M})}, ..., \mathsf{trace}_{(\mathsf{E_P},\mathsf{in_1})})$. We say that an enclave execution is oblivious, if given two sets of inputs $\overrightarrow{y}$ and $\overrightarrow{z}$,

their adversarial views $\mathsf{V}(\overrightarrow{\mathsf{y}})$ and $\mathsf{V}(\overrightarrow{\mathsf{z}})$ are computationally indistinguishable to anyone but the client.

B. Intel SGX

In this section we give a brief introduction to Intel Software Guard Extensions (SGX) and highlight aspects relevant to ZeroTrace. (See [1], [9] 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 IO communication 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 $\mathsf{Load}(\mathsf{P})$ and $\mathsf{Execute}(\mathsf{E_P}, \mathsf{in})$ functions from the previous section.

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

2) $\mathsf{Execute}(\mathsf{E_P}, \mathsf{in}) \rightarrow (\mathsf{out}, \psi)$*:* SGX protects enclave program execution by isolating enclave code and data in Processor Reserved Memory (PRM), referred to 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.

3) 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.

4) 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 [40].

5) Security Challenges in Intel SGX: We now detail aspects of Intel SGX that present security challenges and motivate the design of ZeroTrace.

a) 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], [35], [48] and branch predictor [21]) and the application's page-level access pattern [7], [53].

b) 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 [25]. More generally, SGX provides no privacy/integrity guarantees for any memory beyond the EPC (e.g., non-volatile disk). Ensuring persistent integrity for data and privacy/integrity for non-volatile data is delegated to the user/application level.

c) 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.

6) 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.

a) 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]).

b) 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 persistent system integrity if the enclave can be terminated/swapped at any point in its execution.

C. ORAM

We now describe the popular definition for ORAM from the literature [42], [43]. Afterwards, we provide additional details for the Path ORAM [43] and Circuit ORAM [49] schemes, used in this paper.

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.

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

2) 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 $\mathsf{read(a_i)}$ or a $\mathsf{write(a_i)}$ 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 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.

D. Path ORAM

We now give a summary of Path ORAM [43], one of the ORAMs used in our 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 'client' storage are inversely proportional [42], [43], [49]. Further, the SGX and oblivious settings decrease performance when using larger controller storage (due to EPC evictions [25] and the cost of running oblivious programs; see Section VI). Path ORAM provides a middle ground here: better bandwidth/larger storage than Circuit ORAM [49]; worse bandwidth/smaller storage than SSS ORAM [42].

1) 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.

2) Controller Storage: The Path ORAM controller storage consists of a stash and a 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 $\omega(\log N)$ [43]. 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.

3) 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.

4) 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.

5) 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. [42] and Shi et al. [37]. 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.

6) Extension: Integrity. Path ORAM assumes a passive adversary by default. To provide an integrity guarantee with freshness, one can construct a Merkle tree mirrored [43] 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* [34].

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

E. Circuit ORAM

We now briefly highlight the differences between Circuit ORAM [49] and Path ORAM. In the interest of space, we describe our work using Path ORAM as the memory controller since it is the conceptually simpler ORAM. Circuit ORAM was designed with the intent of having smaller 'circuit complexity'[1] while managing ORAM controller storage, which also improves efficiency when running ORAMs in a data oblivious manner. Both of these construction operate identically up to 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/dummy slots that are deeper in the path. This however requires some amount of "foresight" for which blocks can potentially move 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 (performance-related) differences between Path ORAM and Circuit ORAM in the context of ZeroTrace:

- Circuit ORAM introduces $\sim 50\%$ more I/O bandwidth than Path ORAM. In particular, Circuit ORAM has to fetch and evict two additional paths per access but can operate with $Z = 2$.
- The 'stash' required by Circuit ORAM is much smaller than that of Path ORAM ($O(1)$ as opposed to $\omega(\log N)$

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

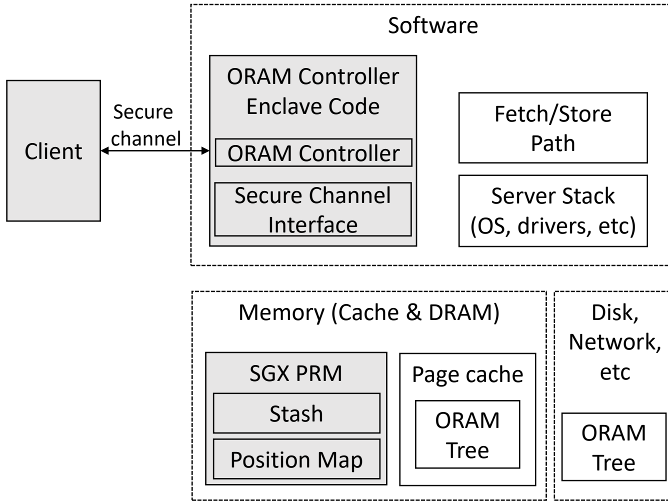

Fig. 1: System components on the server. Trusted components (software and regions of memory) are shaded. Depending on the setting, the client may be connecting from a remote device (not on the server) or from another enclave on the same machine.

blocks). This means data oblivious execution under Circuit ORAM is more efficient than with Path ORAM, as we will see in the next section.

IV. 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 II-B). 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. As mentioned in Section III-C, we will explain the controller assuming a Path ORAM backend for exposition.

A. Design Summary

1) Security challenges and solutions: Since ZeroTrace's ORAM controller runs inside an enclave, and is therefore vulnerable to software-level side channel attacks (Section III-B5), 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 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 persistent storage difficult because the EPC is volatile (Section III-B), incurs large EPC paging overheads (Section III-B6) 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 [33], [37], [49]) 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 unexpected TCB benefits: we propose optimizations in Section IV-E which bloat the path fetch/store code. By delegating these parts to untrusted code, they can be implemented with no change to the TCB.

2) 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 $3\times$ and $1000\times$ – 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).

B. Client Interface

The ORAM Controller Enclave exposes two API calls to the user, namely $\mathsf{read(addr)}$ and $\mathsf{write(addr, data)}$. Under the hood, both the API functions perform an ORAM access (Section III-D).

C. 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):

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

1) 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.

2) 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 IV-E). Since Fetch/Store path are not in the TCB, these optimizations do not effect security.

D. 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.

1) Initialization: For initialization, the server performs the function $\mathsf{Load}(\mathsf{P}) \rightarrow (\mathsf{E_P}, \phi)$, 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 $\mathsf{K_e}$ 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.

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 does not 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.

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.

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 IV-C) 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 [9]. Since our design accesses ORAM through a Fetch/Store Path interface, we cannot use these SGX built-in mechanisms for ORAM privacy/integrity.

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 IV-D1) 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 [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.

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

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

b) Oblivious Block Retrieval: P must now fetch the path for leaf (Section III-D) using a Fetch Path call (Section IV-C). 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 [43]. 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 [24], [43] showed that a stash size of 89 to 147 is sufficient to achieve failure probability of $2^{-\lambda}$ with the security parameter values from λ = 80 to λ = 128. In our implementation, we use a static stash size of 90. [2]

c) Oblivious Path Rebuilding: Finally, P must rebuild and write back the path for leaf (Section III-D) using internal logic and a Store Path call (Section IV-C). 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)
```

[2]For our Circuit ORAM variant we use a fixed stash size of 10 which is known to be sufficient from [49] .

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 III-D. At the end, P then calls Store Path on the rebuilt path, causing the server to overwrite the existing path in server storage.

d) 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 IV-D4. At the same time, an oblivious implementation of the Merkle tree (Section III-C) checks and re-build are performed to verify integrity with freshness.

E. 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.

1) 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:

```
//s_index is subtree_index
```

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

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.

2) Caching the ORAM tree: A popular Tree ORAM optimization is to cache the top portion of the ORAM tree in a fast memory [24], [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 to store the top portion of the tree, as seen later in Fig 4 and Table 7. In this case, Fetch/Store Path allocate regular process memory to store the top portion, and explicitly store the lower portion behind disk IO calls.

F. 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 4.1: 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 III-A.

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 4.1.1: 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 III-B) 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 4.1.2: 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 4.1.3: 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 IV-D2) operations. oupdate performs

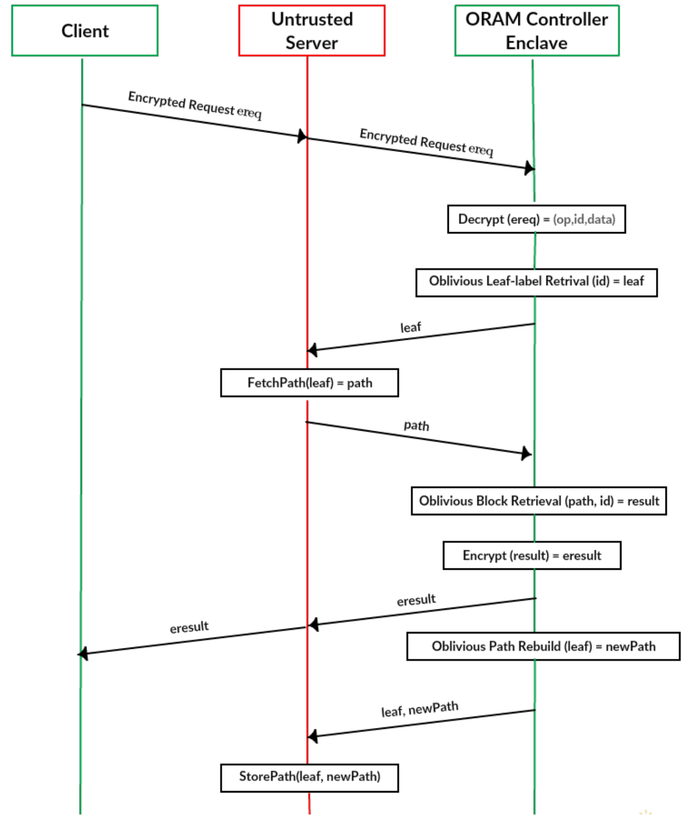

Fig. 2: Execution of an access request

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 4.1.4: 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 III-D4).

Claim 4.1.5: 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 [43]. 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 4.1.6: Oblivious Block Retrieval leaks no information.

Once FetchPath completes, the only code that processes the path is the decryption logic plus the oblivious subroutine given in Section IV-D5. This loads the real blocks from the path into the stash and return the requested block to the user, Since the length of `path` and `stash` are data-independent, obliviousness reduces to the security of oupdate (see Claim 4.1.3).

Claim 4.1.7: Oblivious Rebuild leaks no information.

Same argument as Claim 4.1.6, since `new_path`, `bu` and `stash` have data independent size.

Claim 4.1.8: 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.

V. Persistent Integrity

An important attribute in storage systems is to be persistent 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 persistent 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 IV).

Definition 5.1 (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 [57], [59]. Finally, the above definition isn't specific to ORAM controllers, however we will assume an ORAM controller for concreteness.

a) 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.

b) Security: To maintain the same security level as described in Section II-B, 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 consistent 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 III-A). In our setting, we must analyze two places in the protocol. First, the path written back to untrusted storage after each request (Section IV-D5) 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 IV-C) must be *deterministic* with respect to previous requests. This property is achieved by re-assigning leaf labels using a pseudo-random number generator. We note that similar mechanisms are used to prevent replay and mix-and-match attacks in Nayak et al. [**?**].

VI. Implementation and Evaluation

A. 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 (referred to as "memory"). Beyond DRAM, our system utilizes a Western Digital WD5001AALS 500 GB 7200 RPM HDD as backing untrusted storage. Unless otherwise specified, the core memory controller uses tree top caching in DRAM (Section IV-E2) whenever the ORAM capacity spills to disk.

ZeroTrace is implemented purely in C/C++ (and assembly) for both performance and easier compatibility with Intel SGX as enclave code is limited to purely C/C++ code. Our implementation consists of 6600 lines of code in total, with almost 4000 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 a stand-alone enclave – to best model their performance as plug-and-play memory protection primitives (Section II-A). 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). We predominantly evaluate 8 B and 1 KB ORAM block sizes, which serve as proxies for word-level ("plug-and-play") and file-level size blocks. We note that our experiments apply sequential memory access patterns to the memory controller.[3]

B. 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 II-B). 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 500 KB within the EPC pages and always set the recursion ORAM block size to 64 B (a processor cache line). When recursion is not used, the position map (which

[3] Sequential access patterns maximize stash pressure [43]. Since we use a static stash size (Section IV), this does not effect our response time.

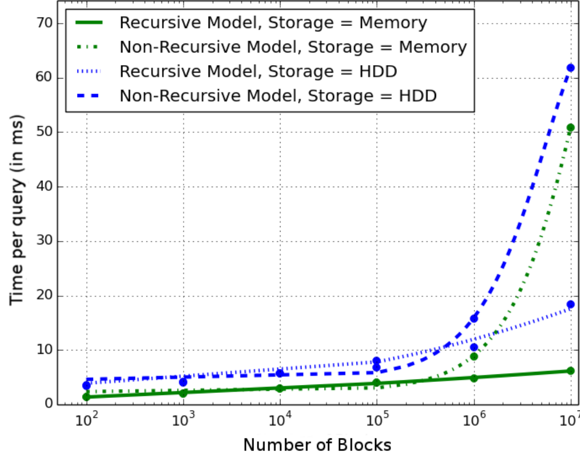

Fig. 3: Representative result. Shows the number of data blocks vs. time per request, with data blocks of size 1 KB with Path ORAM as the the underlying ORAM for ZeroTrace.

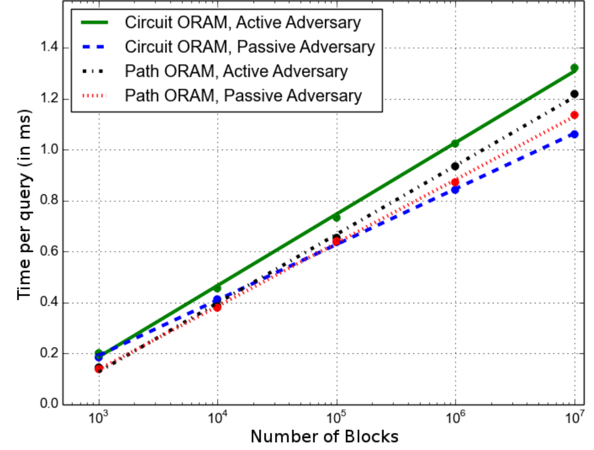

Fig. 5: Comparison of Circuit ORAM and Path ORAM as the ORAM schemes for ZeroTrace under passive and active adversarial models. Each ORAM uses a data block size of 8 bytes.

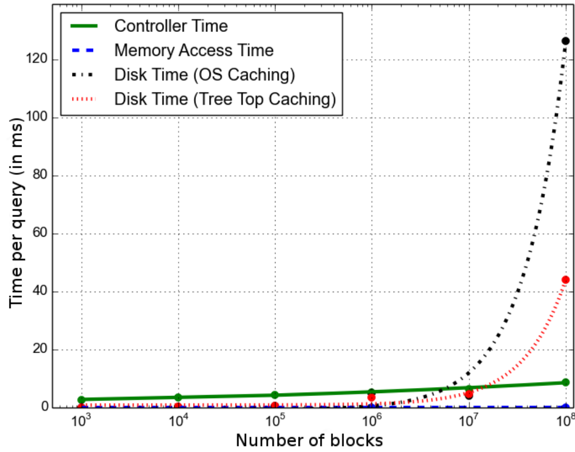

Fig. 4: Detailed performance breakdown for ZeroTrace with Path ORAM as the underlying ORAM, given a 1 KB block size. Total time per request is the sum of controller and storage (DRAM or HDD) times. The ORAM spills to disk given $\geq 10^7$ blocks.

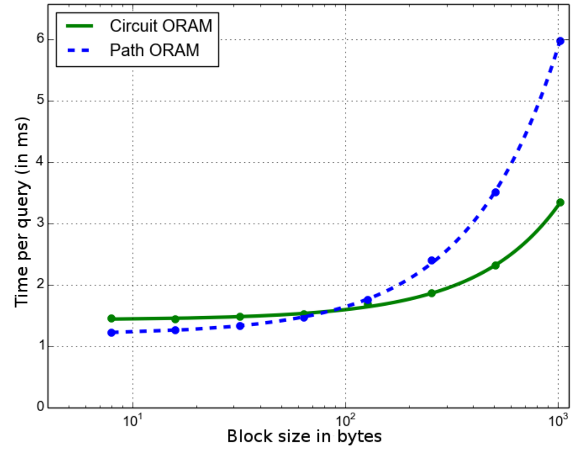

Fig. 6: Performance as a function of data ORAM block size for a dataset with $N = 10^7$ blocks, using recursion and DRAM as the storage backend.

is unbounded in size,) is streamed through the EPC, paging as necessary, incurring the overhead of paging EPC pages as mentioned in III-B6. From Figure 3, we see recursive ORAM pays off for large datasets. This matches the theory [43] and our system uses whichever configuration achieves the best performance, depending on public parameters.

1) Performance breakdown: Figure 4 breaks down the time taken to run oblivious enclave code in the memory controller, vs. the time spent servicing untrusted memory requests. We compare two ways to cache ORAM in DRAM when capacity spills to disk: automatic OS caching and manual tree top caching (Section IV-E2) and find that tree top caching significantly improves performance. For sufficiently large ORAMs, disk time dominates access time. This issue isn't fundamental; our system can use an SSD to improve disk latency. For smaller ORAMs, which will be common in the data-structure/plug-and-play setting, the oblivious controller is the bottleneck, given fast untrusted DRAM. Hence, to improve performance in the context of our proposed plug-and-play memory controller, we designed and implemented an oblivious variant of Circuit ORAM (Section III-E) to serve as the backend ORAM scheme.[4]

Figure 5 compares ZeroTrace between Circuit ORAM and Path ORAM backends, under both active and passive adversarial models. Contrary to expectation, Circuit ORAM does not perform significantly better than Path ORAM given a small (word-level) block size, which will be common in a data-structure setting. The primary reason for this is SGX ECALL/OCALLs have a large constant overhead of 0.015ms in addition to the taking time proportional to the path length.

[4]We note that Circuit ORAM was designed to be asymptotically efficient when coded in an oblivious manner, but it still needs to be written in terms of CMOV in our setting.

Circuit ORAM requires three path fetch and stores from the server for each access, the ORAM controller logic for Circuit ORAM is about 2-3x faster than that of Path ORAM, however the overhead of moving these three paths in and out of the enclave memory controller throttles Circuit ORAM's performance. Moreover this overhead is aggravated by recursion as well, since Circuit ORAM pays this cost for each level of recursion.

Breaking this down further, Figure 6 shows the controller request time varying the data block size, between Path and Circuit ORAM. For small data block sizes, the curve is flat because the cost of recursion dominates. In Figure 6, we see that despite the aforementioned limitation, Circuit ORAM's eviction circuit begins to outperform Path ORAM significantly at larger block sizes. This is because the cost of obliviously moving blocks becomes dominant at larger block sizes, and Path ORAM's eviction procedure has to perform significantly more of these oblivious move operations than Circuit ORAM. The reason for these additional move operations in Path ORAM is two fold; first, recollect that Path ORAM has to iterate over the entire stash for each bucket on a fetched path while performing oblivious updates as explained in Section IV-D5c, whereas Circuit ORAM makes a single stash + path pass. Second, as mentioned in Section III-E, Circuit ORAM requires a smaller stash size of $O(1)$ as opposed to $\omega(\log N)$ blocks required by Path ORAM. [5] Additionally, we note that scaling block sizes has a discretized performance effect since we work with the blocks at a granularity of 64 B registers. A block of 1 KB performs 16 iterations of CMOV instructions within an oupdate function, whereas a block of 8 B performs a single CMOV instruction.

We show a detailed performance breakdown for ZeroTrace while varying the underlying ORAM scheme, data block size and storage backend in the table in Figure 7. The table illustrates the overhead of I/O for Circuit ORAM as mentioned in Section III-E. From this table, it is clear that if the application requires HDD backends, ZeroTrace should use Path ORAM instead of Circuit ORAM, whereas in the plug-and-play memory setting Path ORAM outperforms Circuit ORAM at small block sizes and vice versa at large block sizes.[6] Thus, being able to flexibly change the underlying ORAM scheme based on public initialization parameters allows ZeroTrace to optimize its performance. Additionally, as mentioned before if the application requires weaker security guarantees, ZeroTrace can revert to passive-only protection to optimize its performance (as seen in Figure 5).

C. Evaluation of Data-Structure Modules

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

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

b) 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.

[5] In our implementation we use a static stash size of 10 for Circuit ORAM and 90 for Path ORAM.

[6] We see from Figure 6 that the switch over point is at block size 100 bytes.

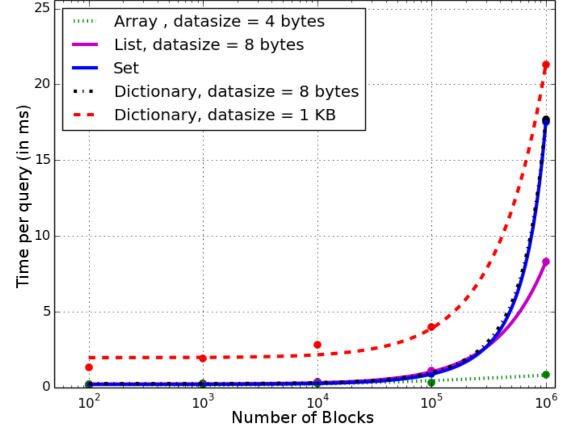

Fig. 8: Evaluation of our oblivious memory controller library for Set/Dictionary/List/Array. Array is a direct call to our core memory controller, which uses ORAM recursion to be asymptotically efficient.

c) 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.

d) 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 8 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. [51], which makes use of ORAMs and can use our core memory controller as a primitive as well.

VII. 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:

1) 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 [33], [43], [49]. In the context of ORAM, our work moves the ORAM controller close to storage, exploiting the fact that ORAM bandwidth overhead

Underlying ORAM	Block Size	Backend	Controller Time	Backend Time	Total Time
Path ORAM	8	DRAM	1.2141	0.0048	1.2189
Path ORAM	1024	DRAM	5.9938	0.0152	6.0091
Path ORAM	8	HDD	1.223	40.2137	41.4367
Path ORAM	1024	HDD	5.9921	43.8868	49.8789
Circuit ORAM	8	DRAM	1.304	0.0167	1.3207
Circuit ORAM	1024	DRAM	3.3242	0.0645	3.3887
Circuit ORAM	8	HDD	1.327	132.5139	133.8409
Circuit ORAM	1024	HDD	3.3359	137.4236	140.7595

Fig. 7: Performance numbers for ZeroTrace under different parametrizations of underlying ORAM controller, data block size and backend storages. All timings are in ms. Experiments have $N = 10^7$ blocks, and all experiments that use HDD backends in this table make use of Tree Top Caching. Note that the controller time is also inclusive of time spend by the controller in recursion and time taken by the overheads of ecall/ocall.

occurs *between ORAM controller and storage* and not between client and ORAM controller. This idea has been explored by combining homomorphic encryption with ORAM [10], and by the ORAM-based systems Oblivistore [41] 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.

Another similar direction of research is secure hardware projects such as Phantom [24], Aegis [44] and Ascend [11]. 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[7] 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.

While Phantom achieves similar security goals as that of ZeroTrace, there are several differences between our project and such secure hardware projects. First, since these projects rely on custom hardware that are uncommon commercially (typically unavailable), 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 DRAM, however ZeroTrace is extremely flexible with respect to the underlying storage support. Additionally, ZeroTrace also offers security flexibility, which allows applications to trade their higher level of security for performance efficiency when required.

2) 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 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 [22] 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, [22]). 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 [59] 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.

3) Attacks and Defenses: The primary attack vectors against SGX in 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 [16], [19], [23], [31], [45], [47], [55], [58] or branch predictor [21]) 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 that detect if an enclave is undergoing a shared resource attack based on frequency or magnitude of enclave exits/interruptions (e.g., T-SGX [38] and Deja Vu [8]) and new attacks (e.g., Brasser et al. [6], Wang et al. [48]) that work towards reducing the required enclave exits. Gruss et al. [15] recently demonstrated a new direction for defense mechanisms against cache side-channel by leveraging Hardware Transactional Memory(HTM).

Similarly, a malicious OS can induce and monitor appli-

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

cation page fault behavior to learn program memory access patterns [53]. Bulck et al. [7] demonstrated attacks that infer page accesses through bits set in the page tables without resorting to page faults. Shinde et al. [39] proposed compiler-based defense mechanisms against page-level attacks by moving secret-dependent control and data flows into the same page. However their approach is still susceptible to cache attacks.

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

VIII. 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, they can also be used to implement remote file storage systems – granting constant WAN bandwidth overhead solutions at the expense of trusting the SGX mechanism.

References

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," 2013.

[2] 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)*, 2016, pp. 689–703.

[3] 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*, 2012, pp. 257–271.

[4] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, p. 8, 2015.

[5] 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*, 2009, pp. 43–54.

[6] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," *CoRR*, 2017.

[7] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1041–1056.

[8] 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*, 2017, pp. 7–18.

[9] V. Costan and S. Devadas, "Intel sgx explained," 2016.

[10] 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*, 2016, pp. 145–174.

[11] 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*, ser. STC '12. New York, NY, USA: ACM, 2012, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2382536.2382540

[12] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: ACM, 2009, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536440

[13] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, pp. 431–473, 1996.

[14] T. C. 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.

[15] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory." USENIX Association, August 2017.

[16] 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*, 2011, pp. 490–505.

[17] 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)*, 2016, pp. 533–549.

[18] Intel, "Intel."

[19] 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*, 2014, pp. 299–319.

[20] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in cryptologyCRYPTO99*, 1999, pp. 789–789.

[21] 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.

[22] 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*, pp. 87–101, 2015.

[23] 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*, 2015, pp. 605–622.

[24] 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*, 2013, pp. 311–324.

[25] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "Rote: Rollback protection for trusted execution," 2017.

[26] 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.*, vol. 42, no. 4, pp. 315–328, Apr. 2008. [Online]. Available: http://doi.acm.org/10.1145/1357010.1352625

[27] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, 2016, pp. 10:1–10:9.

[28] 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*, 2013, pp. 10:1–10:1.

[29] 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*, 2005, pp. 156–168.

[30] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[31] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers Track at the RSA Conference*, 2006, pp. 1–20.

[32] 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*, 2015, pp. 431–446.

[33] 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*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 415–430. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831170

[34] 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*, 2013, pp. 1–6.

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

[36] 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*, 2016, pp. 1–11.

[37] 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*, 2011, pp. 197–214.

[38] 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.

[39] 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*, 2016, pp. 317–328.

[40] S. Shinde, D. L. Tien, S. Tople, , and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves," in *NDSS*, 2017.

[41] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," in *Security and Privacy (SP), 2013 IEEE Symposium on*, 2013, pp. 253–267.

[42] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *arXiv preprint arXiv:1106.3652*, 2011.

[43] 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 (CCS'13)*, 2013, pp. 299–310.

[44] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *Information Security Technical Report*, vol. 10, no. 2, pp. 63–73, 2005.

[45] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, pp. 37–71, 2010.

[46] 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*, 2014, pp. 9:1–9:14.

[47] J. van de Pol, N. P. Smart, and Y. Yarom, "Just a little bit more," in *Cryptographers Track at the RSA Conference*, 2015, pp. 3–21.

[48] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," *arXiv preprint arXiv:1705.07289*, 2017.

[49] 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*, 2015, pp. 850–861.

[50] 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*, 2014, pp. 215–226.

[51] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 215–226.

[52] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012, pp. 293–304.

[53] 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*, 2015, pp. 640–656.

[54] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *Security and Privacy (SP), 2016 IEEE Symposium on*, 2016, pp. 18–37.

[55] 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*, 2014, pp. 719–732.

[56] 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*, ser. SP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 79–93. [Online]. Available: http://dx.doi.org/10.1109/SP.2009.25

[57] 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*, 2012, pp. 2–2.

[58] 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*, 2014, pp. 990–1003.

[59] 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)*, 2017, pp. 283–298.