OBLIVIATE: A Data Oblivious File System for Intel SGX

Adil Ahmad
Purdue University
ahmad37@purdue.edu

Kyungtae Kim
Purdue University
kim1798@purdue.edu

Muhammad Ihsanulhaq Sarfaraz
Purdue University
msarfaraz@purdue.edu

Byoungyoung Lee
Purdue University
byoungyoung@purdue.edu

***Abstract*—Intel SGX provides confidentiality and integrity of a program running within the confines of an enclave, and is expected to enable valuable security applications such as private information retrieval. This paper is concerned with the security aspects of SGX in accessing a key system resource, files. Through concrete attack scenarios, we show that all existing SGX filesystems are vulnerable to either system call snooping, page fault, or cache based side-channel attacks. To address this security limitations in current SGX filesystems, we present OBLIVIATE, a data oblivious filesystem for Intel SGX. The key idea behind OBLIVIATE is in adapting the ORAM protocol to read and write data from a file within an SGX enclave. OBLIVIATE redesigns the conceptual components of ORAM for SGX environments, and it seamlessly supports an SGX program without requiring any changes in the application layer. OBLIVIATE also employs SGX-specific defenses and optimizations in order to ensure complete security with acceptable overhead. The evaluation of the prototype of OBLIVIATE demonstrated its practical effectiveness in running popular server applications such as SQLite and Lighttpd, while also achieving a throughput improvement of 2×-8× over a baseline ORAM-based solution, and less than 2× overhead over an in-memory SGX filesystem.**

I. INTRODUCTION

Hardware-assisted trusted computing solutions are gaining popularity today. One such solution is Intel SGX, which guarantees confidentiality and integrity of a user program despite the program being executed in a remote and potentially adversarial environment. Using SGX, the user program runs within an enclave, a protected region provided by the Intel hardware. The program's execution semantics including registers and memory footprints are isolated from the remaining system components such as kernel, VMM, BIOS, etc. This isolation allows the user program to run securely even when an attacker with higher privilege i.e., kernel, tries to actively modify program execution. SGX also supports a remote attestation mechanism so that the user can verify that his/her application is actually being executed within an SGX context.

An SGX enclave is designed to run user-level programs, and therefore relies on external system components to access I/O resources. Amongst these resources, we note that accessing files is especially important in order to realize various security applications with SGX. This is because most of the computing data is present in the form of files. For example, running database systems in SGX environments, where the database is backed up by files, can enable the concept of private information retrieval (PIR) [13]. As another example, web servers or content delivery networks (CDNs) in SGX environments can ensure user's privacy, and most of this data is presented in a file (e.g. HTML file, JavaScript code, and images). Other potential applications include cloud-based backup storage where the user wants to securely store and access their data without going through the hassle of maintaining a local copy.

Model	Metadata Location	Side-Channel Attacks			Implementation for SGX
		Syscall	PF	Cache	
Naive (§III-A)	Kernel	×	×	×	SCONE [7], Intel FS [3]
In-memory (§III-B)	Enclave	✓	×	×	Haven [9], Ryoan [18]
Hybrid (§III-C)	Enclave	△	×	×	Graphene [47, 48]
OBLIVIATE (§VI)	Enclave	✓	✓	✓	This paper

TABLE I: The comparison of file systems models for SGX. Columns under side-channel attacks indicate whether each filesystem model is secure against corresponding attacks: Syscall denotes syscall snooping based attack, PF denotes page fault based attacks, and Cache denotes cache based attacks.

Recognizing these needs, Intel recently released Protected File System Library for SGX [3]. To perform filesystem operations, it allows an enclave to delegate such operations to the untrusted kernel, i.e., relaying all filesystem system calls from an enclave to the untrusted kernel. It also provides an encryption module in order to protect confidentiality. However, this can still result in critical security breaches because the untrusted kernel can observe detailed interactions between the enclave and the kernel (i.e., parameters in the system call including filename and file offset) while the enclave is accessing a certain file. In-memory filesystems for SGX [9, 18] prevent the above mentioned system call based attack. In the in-memory filesystem, an enclave pre-loads all the required file contents within its protective memory region and handles all filesystem interactions from within the enclave. Thus, when the enclave program tries to access a file, the request does not need to be forwarded to the kernel and can be served by the enclave. Unfortunately, in-memory filesystems are still susceptible to side-channel attacks adapted for the SGX environments, namely page fault based attacks [50] and cache based attacks [10, 40]. By observing memory access patterns, these attacks can gain insight into the internal processing semantics of the enclave application.

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.2018.23284
www.ndss-symposium.org

To demonstrate the feasibility of these attacks on the filesystem, we performed concrete attacks using current filesystem models for SGX. For our demonstration, we assume a scenario where an insurance company runsa SQLite database storing medical records. The company wants to use a secure cloud infrastructure to protect its medical data as well as associated operations. As such, the company runs SQLite inside an SGX enclave, where the database is stored in an encrypted file and the database communication channel is encrypted as well. According to our results, sensitive information carried in the database query can be leaked to potential attackers (e.g, the untrusted kernel) using current filesystems for Intel SGX. More specifically, through syscall snooping and page fault based side-channel attacks, we confirm that both system call traces and memory access patterns can be exploited to learn which row and column a specific database query is processing. We believe our case study on this attack advocates the strong need for introducing a side-channel resistant filesystem model for Intel SGX.

This paper presents OBLIVIATE[1], a data oblivious file system for Intel SGX. The key idea behind OBLIVIATE is to employ ORAM operations for SGX to hide file access patterns. OBLIVIATE carefully materializes the conceptual components of ORAM for SGX such that it can seamlessly function as a filesystem while providing systematic preventions against above mentioned side-channel attacks. At a higher level, OBLIVIATE runs an isolated filesystem enclave in a separate process, where the application enclave relays filesystem related operations through encrypted communication channels. In order to optimize communication overheads due to this isolated and separate filesystem design, OBLIVIATE utilises *exitless* communication schemes, namely *message queues* and *shared memory*, each of which facilitates intra-process and inter-process communication. In terms of adopting ORAM, since the ORAM implementation itself is exposed to side-channel attacks against the enclave (e.g., page fault based [50] or cache based attacks [10, 40]), OBLIVIATE uses data oblivious algorithms in accessing key data structures of ORAM.

OBLIVIATE presents two performance optimization techniques in applying ORAM for SGX: (1) To efficiently maintain ORAM server storage, OBLIVIATE develops an additional security memory region with non-encrypted memory regions of SGX (i.e. more precisely, non-EPC memory §II-A). This enables OBLIVIATE to avoid costly context switches, which arise due to limited EPC memory, if it were directly stored within the enclave memory. (2) To reduce ORAM latency, we exploit the internal working characteristics of ORAM and employ asynchronous ORAM server update schemes. As such, OBLIVIATE returns the required data as soon as it becomes available and performs path updates asynchronously, rather than waiting for expensive ORAM path updates.

To summarize, this paper makes following contributions:

- We demonstrate the feasibility of side-channel attacks against current filesystems for SGX by developing concrete attacks for them. We assumed a realistic usage scenario where the victim runs SQLite within an enclave, and our attacks confirm that the security of current SGX filesystems can be breached through side-channel attacks.
- We design and implement OBLIVIATE, a secure file system support for SGX. It systematically adopts ORAM protocols to hide file access patterns. As the ORAM implementation itself running inside an SGX enclave can be vulnerable to side-channel attacks, OBLIVIATE employs data oblivious algorithms in accessing ORAM's data structures. OBLIVIATE also employs systematic performance optimizations while keeping in mind the nature of applications it services.
- We provide a security analysis of OBLIVIATE against page fault based side-channel attacks and explain how and why these attacks become futile under OBLIVIATE.
- We evaluate OBLIVIATE on real SGX hardware. To show its practical aspects, we not only run IOZone filesystem benchmarks [30] but also real-world applications including SQLite [33] and Lighttpd [22]. We show that OBLIVIATE achieves a throughput improvement of 2×- 8× compared to a baseline solution employing traditional ORAM, and slows down within the range of 1.5×- 2× compared to the in-memory SGX filesystem.

The rest of this paper is organized as follows. §II provides background of paper, and §III describes existing filesystem models for SGX. §IV provides our case studies on launching side-channel attacks against existing filesystem models. §VI describes the design of OBLIVIATE, and §VIII evaluates various aspects of OBLIVIATE. §IX discusses potential applications of OBLIVIATE. §X provides related work of this paper, and lastly §XI concludes the paper.

II. BACKGROUND

A. Intel SGX

Intel SGX [4] is a set of hardware instructions introduced by Intel, and recently commoditized with the Intel Skylake CPU architecture. The primary motivation behind Intel SGX is to provide confidentiality and integrity of a user program while reducing the trust model up to the CPU itself. Thus, a user program can be securely run on top of hostile, potentially adversarial system components, including high-privileged software such as kernel, VMM, and BIOS. The key enabling technology of Intel SGX is in its protected execution region, termed as an *enclave*, which uses hardware protections guaranteed by Intel SGX. In particular, the user program is run within an enclave, and any transition to the untrusted execution context (e.g. transition to the untrusted kernel, either by synchronous or asynchronous exit events) is preceded by complete encryption of all enclave execution contexts, including CPU registers and enclave memory footprints. Since this encrypted execution context can be only decrypted under the enclave execution context, all other system components cannot harm its confidentiality and/or integrity. In order to validate the initial integrity of a user program, SGX provides a remote attestation mechanism. Moreover, SGX also provides a sealing mechanism (i.e. generating per-enclave or per-authority unique encryption key) to support persistent data storage for an enclave program.

Limitation: Memory Resource Use. The strong security guarantees of SGX are achieved with a few constraints on

[1] The term "Obliviate" is used as an incantation in the Harry Potter series which is used to wipe a person's memory.

hardware resource usage. One key limitation is that the enclave has a limited memory allowance for its execution. More precisely, Enclave Page Cache (EPC), the physical memory space for enclave programs, is allowed to utilize only up to 128 `MB`. It is worth noting that an enclave program can have more than 128 `MB` of virtual address space. This is supported by the untrusted kernel (i.e., Intel SGX SDK drivers), which performs swap in/out of memory pages between EPC and non-EPC physical memory regions.

Limitation: Side-Channel Attacks. Intel SGX does not provide systematic protection mechanisms against side-channel attacks and relies on the application developer. While there can be many possible attack vectors (e.g. power monitoring attacks, electromagnetic attacks, and access pattern analysis based on bus snooping attacks [15]), we mainly focus on following three feasible attacks, which do not require physical accesses to the machine: (a) syscall snooping, (b) page fault based and (c) cache-based side-channel attacks.

The root cause of syscall snooping attacks is that an enclave program has to rely on other system components (e.g. kernel) for accessing computing resources including files or network functionality. This is because an enclave is designed to be run with user-level privileges (i.e. ring-3) while accessing system resources require higher privileges (i.e. ring-0). As a result, Intel SGX SDK [19], a programming toolkit for SGX platform released by Intel, provides `ocall`, which can be used to forward system calls from an enclave to the kernel. However, since the untrusted kernel can now observe the syscall operation, this may harm the confidentiality guarantee provided by an SGX enclave. For example, consider filesystem operations where the untrusted kernel observes which file has been accessed and/or what exact offset has been read from or written to. Haven [9] and Ryoan [18] attempt to address this issue by introducing an in-memory file system for SGX. However, in-memory filesystem is also insecure which we discuss further in §III.

Page fault based side-channel attacks [50] are memory access pattern based attacks for the SGX context. The untrusted kernel can mark all of these pages to be non-accessible by either manipulating page table permissions or directly evicting them. This allows the kernel to learn which memory page has been accessed by an enclave program through page faults. It is worth noting that the SGX clears the offset of the page before switching context to the kernel and therefore the granularity of this attack is a paging unit (i.e., 4`KB`). Several works [34, 42] have presented solutions to mitigate this attack, but each of these has its own limitations, especially when we consider the filesystem §X.

Recent reports [10, 40] have shown that SGX is vulnerable against cache-channel attacks, specifically the `Prime+Probe` attack. Although these reports show how cache based attacks can be used to obtain cryptographic keys in an SGX setting, similar attacks can be mounted against the filesystem (specifically the in-memory filesystem) to find out what offset within the filesystem was accessed. This attack is conceptually similar to its variant in the non-SGX setting but previous solutions [21, 51] considered a trusted OS to perform various mitigations. Those solutions are not applicable within the SGX setting since the OS is now untrusted.

B. ORAM

ORAM [16] provides obfuscated access to encrypted memory which prevents an attacker from learning information about the user/program even though the attacker can observe the data access patterns. The key idea behind ORAM is to obfuscate access to the same memory region each time (by re-shuffling and re-encrypting with a new nonce) so that the attacker is unsure which memory region is being accessed despite observing multiple runs of the same program. ORAM was originally introduced for the remote setting where a client stores his/her encrypted data in an untrusted memory region (i.e., remote machine) and does not want an attacker to learn what data is being accessed as he/she tries to retrieve the data. To achieve this, ORAM assumes the availability of a trusted memory region which stores some *metadata* corresponding to the untrusted memory. This metadata is essential to keeping track of the actual location of stored data in the untrusted memory. In the traditional sense, the trusted memory region could be located within the client's personal machine or some other trusted machine. We explain some of the popular ORAM designs [28, 46], below.

Path ORAM [46] is an optimization of ORAM which uses a complete binary-tree structure to store encrypted blocks of memory in an untrusted machine. The tree is made up of multiple nodes where each node holds multiple blocks. A path ORAM tree contains both real blocks (i.e., blocks that hold client's data) and dummy blocks (i.e., useless blocks). Since all blocks are encrypted, the untrusted entity cannot distinguish between them. The number of real blocks in a path ORAM tree is always equal to the number of leaf nodes within the tree. The remaining blocks are filled up with the dummy blocks. All these blocks are randomly distributed within the tree, and the client maintains a *Position Map* which points to the leaf node on whose path, the block is located. Moreover, the client needs maintain a *Stash* region to retrieve and store blocks after an access. Figure 1 illustrates an example on how Path ORAM accesses a block.

In Path ORAM, read and write operations are almost the same and the data access is at the granularity of a block. To read or write a specific real block, the client first consults the `Position Map` and obtains the leaf `L` corresponding to the block in the untrusted memory region. To maintain access confidentiality, the client retrieves all blocks on the path from the root to the corresponding leaf `L`. On receiving the blocks, the client discards all dummy blocks and only saves the real block(s) (there can be more than 1 block in the path) into the *stash* region. In case of a write operation, the targeted real block is updated and in the case of a read, it is simply copied onto a separate buffer. In order to obfuscate access to this real block for the next time, a new leaf node `L'` is chosen, uniformly at random. Then the client writes back all retrieved real blocks to the old path `L` with the constraint that the targeted real block should be placed on the newly chosen path (i.e., from the root to `L'`). It is important to note here that all real blocks (along path `L`) are re-encrypted with a random nonce. Also, all nodes (along path `L`) are filled with their regular quota of nodes by adding newly generated dummy blocks in case they are not filled by real blocks. Path-ORAM's obliviousness is achieved by the fact that each access to the same block will yield a new path. Another popular tree-based ORAM is

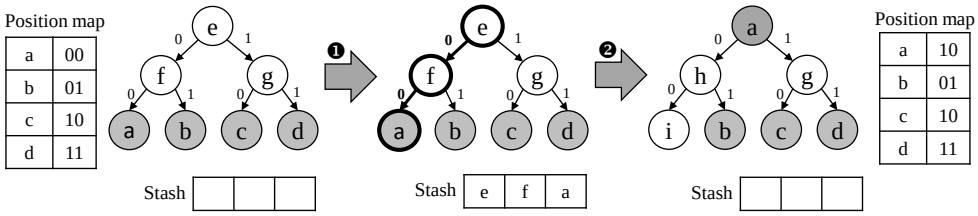


Fig. 1: A Path ORAM example in reading a block 'a' (assuming one block per node for simplicity). Filled blocks (from 'a' to 'd') are real blocks and unfilled blocks are dummy blocks (from 'e' to 'i'). ❶: It attempts to read the block 'a', so all blocks on path '00' are first loaded to the stash. ❷: It randomly picks a new leaf path (assume '10'), and writes back the real block 'a' to the block on the new leaf path. All other blocks on the path '00' are filled with newly generated dummy blocks (i.e., 'h' and 'i').

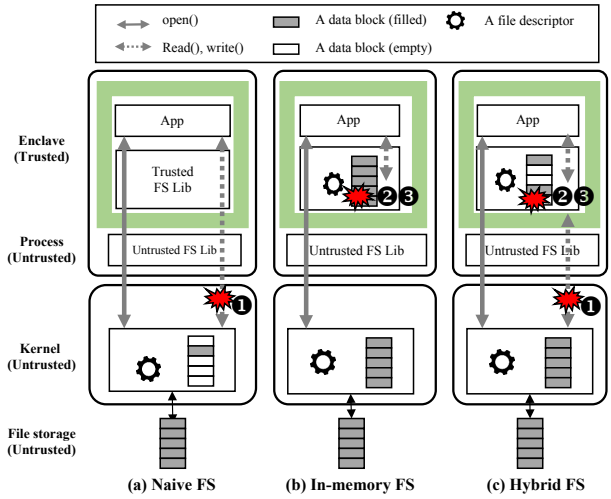


Fig. 2: Available file system designs for Intel SGX. All threes are vulnerable to either syscall snooping attacks (❶), page fault based side-channel attacks (❷) and cache based attacks (❸).

the recursive ORAM [28]. The author's claim to reduce the communication overhead by 30% over traditional Path-ORAM. These two ORAM algorithms have inspired the ORAM-tree structure of OBLIVIATE which we explain in §VI.

III. CURRENT SGX FILESYSTEM MODELS

In this section, we describe current filesystem schemes for Intel SGX as well as pointing out potential attacks to them. Depending on how filesystem related system calls are processed and where filesystem metadata is maintained, existing filesystems can be categorized into three different models: (a) Naive SGX filesystem, which simply forwards all system calls and the OS maintains the metadata; (b) In-memory SGX filesystem, which handles the metadata, filebuffers and syscalls within the enclave; (c) Hybrid SGX filesystem, which uses a combination of the above two models.

A. Naive SGX Filesystem

This model is a natural extension of the traditional filesystem access mechanism for SGX (shown in Figure 2-a), which is used by Intel's Protected File System Library Using SGX [3] and SCONE [7]. In this model, all filesystem operations (including metadata handling) are performed by the kernel. In particular, since SGX does not allow direct syscall invocation, this model simply forwards all filesystem related system calls to its untrusted library, which is running outside an enclave, through *ocall*. Then the untrusted library invokes a corresponding system call which is then processed by the kernel. Therefore, the kernel maintains all filesystem related key metadata, including the file descriptor as well as its associated file buffer cache. The return procedure of the system call is also similar — the kernel first returns the results to the untrusted library, which in turn relays it back to the enclave. In order to provide integrity and confidentiality, an encryption scheme with integrity checks can be used together, where the encryption key (i.e., a sealing key in the SGX context) can be bound with either the enclave's identity itself or the authority that owns the enclave program [4].

Limitation: Syscall Snooping Attacks. Since the kernel performs all syscall operations, it has complete knowledge about (a) which file is being processed during `open` and (b) at which file offset the processing is currently taking place during `read` and `write`. It is worth to note that, even though encryption schemes using an SGX sealing mechanism has been employed, it is still possible that the untrusted kernel may learn much information out of the encrypted file. To be more specific, the order of blocks will stay the same as before being encrypted, because a block-wise encryption scheme (which allows an encrypted memory block to be decrypted as it is without decrypting the whole file) is used. As a result, the offset information in `read` would reveal such an order thereby allowing attackers to guess which part of the file has been accessed.

B. In-memory SGX Filesystem

The in-memory filesystem (shown in Figure 2-b) performs the majority of filesystem interactions within the enclave i.e. EPC memory. Haven [9] and Ryoan [18] use an in-memory filesystem design to overcome syscall snooping attacks. The key difference from §III-A is that the application buffers the complete file data, along with associated metadata, within its enclave. Thus, all following filesystem operations can be performed on the buffered data without involving adversarial system components. For example, in response to `open`, the trusted FS library (that is linked together with an enclave application) opens the file and reads in all file data with the help of the untrusted kernel similar to §III-A. This file data is stored in buffer pages, located within the enclave, which is pre-allocated beforehand. All corresponding filesystem operations, i.e., `read`, `write`, etc. are handled within the enclave.

Limitation: Page Fault and Cache Based Attacks. This model is vulnerable to both page fault based [50] and cache based [10, 17, 40] side-channel attacks launched by the untrusted kernel. In the case of page fault attack, the untrusted

SGX driver is capable of marking all EPC memory pages non-accessible by manipulating page table permissions or directly evicting mapped pages from the EPC regions. This leads an enclave execution context to raise a page fault onto the page it is accessing which is first delivered to the kernel. Then the kernel re-enables access onto the page so that the enclave program can resume its execution. As a result, this attack allows the adversarial kernel to learn the file buffer access information up to the granularity of a paging unit (i.e., 4 KB).

Similarly, the cache based attack is also feasible. The kernel can monitor one of the caches (L1 to LLC) to find out which cache-set and corresponding file offset was accessed. Assuming the adversarial kernel has prior-knowledge on the rough semantic information of enclave's memory layout (e.g., where file buffers are located), the attacker will learn which part of the file has been accessed.

C. Hybrid Filesystem

The hybrid filesystem model blends previously mentioned designs, naive FS model and in-memory FS model. In this model, unlike the in-memory filesystem model, the trusted library does not load the complete file data into the enclave but instead does so *on-demand*. To gain a clearer understanding, the file is buffered within the non-enclave memory (but within the DRAM) and copied into the enclave as required. Graphene [47], particularly the version ported for SGX environments [48], employs this filesystem model.

Limitation. Since the hybrid model mixes up two filesystem models without special security mechanisms, its attack surface also inherits from both models. Thus, although there can be subtle differences in the attacker's capability, the hybrid model is basically vulnerable to all aforementioned attacks, from system call based attacks to page fault and cache based attacks.

IV. Case Study: Launching Attacks

To clearly demonstrate the feasibility of attacks, we performed concrete attacks against current filesystems for SGX. In this attack, we assume that an enclave application runs a popular database application, SQLite [33], where the database file is encrypted and the database communication channel is encrypted as well. SQLite stores user-data persistently through files which are created using the regular Linux Filesystem API, i.e., `open`. Afterwards, all database operations including `insert, select, etc.` are completed by indexing into the database file using `read, write, pread, pwrite`, etc. For simplicity of the attack, we also assume that the attacker has knowledge about database schema (e.g., the count of the tables stored in the database file, and the size of a single row within the table). The attacker can also infer these details by closely monitoring the access patterns onto the database file since SQLite uses data-dependent access to optimize performance.

In this setting, we assume a usage scenario that an insurance company maintains a database storing medical records in order to set insurance premiums. The company wants to use a cloud infrastructure while ensuring that the data is completely isolated and secure. For this purpose, the company runs SQLite inside an SGX enclave. All data (outside the enclave) is encrypted and therefore even a privileged attacker cannot directly read the data. However, the attacker knows that one of the database files contains privacy sensitive information, indicating whether a person has a history of heart disease within his/her family or not. The goal of the attacker is to leverage this schema information to find out whether a given query returns a row with heart disease or not.

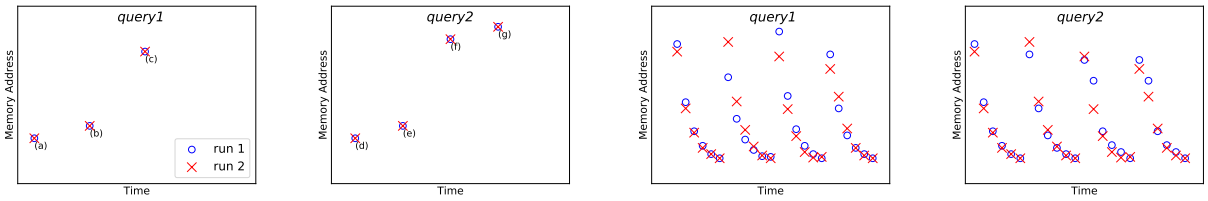
```
1  // Table -> (id (4bytes), history (4KB), no-history (4KB))
2  open("heart.db", O_RDWR, 0666);
3
4  // Query 1: For patient with heart disease.
5  pread64(3, 0x2783933, 4096, 0);
6  pread64(3, 0x2637298, 4096, 4096);
7  pread64(3, 0x2732123, 4096, 32768);
8
9  // Query 2: For patient without heart disease.
10 pread64(3, 0x2637221, 4096, 0);
11 pread64(3, 0x2738212, 4096, 4096);
12 pread64(3, 0x2632119, 4096, 40960);
13 pread64(3, 0x2637223, 4096, 45056);
```

Fig. 3: Syscall traces observed by an attacker, i.e., an untrusted kernel. The first 8KB correspond to the metadata of the SQLite database.

Elaborating more details on this medical database schema, each row corresponds to a 8194 memory chunk which is divided into one column of 4 bytes and two columns of 4 KB each. The first column contains identification information about the person to which this specific row belongs. The second column contains information if the person has a history of heart diseases, and the third column contains information if the person does not have a history of heart diseases. Also, the company runs a single query on the database. The query checks if the provided person ID is associated with a history of heart disease or not, and returns information from one of the intended columns.

Syscall Snooping Attack. In this attack, we run a victim SQLite server within an enclave, where SQLite is built with Intel's Protected File System Library (i.e., naive FS model). Figure 3 shows the syscall traces that can be collected by a privileged attacker. The victim first opens a database file using the `open` syscall (line 1) and the host OS now knows which file is being used (i.e., "heart.db"). Next, the database always reads the first two pages (8KB) in order to maintain metadata information from the database file. The victim runs two queries shown by lines 5-7 and lines 10-13. Furthermore, the attacker observes that the first query (lines 5-7) accesses the fourth row within the database by calculating it against the size of a row (8KB). Also, since the database only reads the first 4KB from then on (line 7), the attacker can infer that only the first column was accessed, which means that the query was meant for a person with history of heart disease. In the second query (lines 10-13), the attacker observes that the offset corresponds to the fifth row, and since it reads two 4KB offsets (lines 12-13), the attacker can infer that this query hits the second column, i.e., a patient without heart disease.

Page Fault based Attack. As far as the in-memory filesystem is concerned, enclave memory space is pre-allocated to store data from various files that need to be accessed. Depending on the underlying development environment, it is possible that the location of this memory region might be randomized (i.e., using ASLR [41]) Even in scenarios where it is randomized, the host OS still can deduce the location of the memory buffer because the location is always fixed after initializing an enclave and the attacker would be able to leverage the repeated memory access patterns at runtime.



(a) In-memory FS **(b)** OBLIVIATE

Fig. 4: Page fault traces observed through the Linux SGX driver for both an in-memory FS and OBLIVIATE. As expected, the in-memory FS exhibits the same page faults across runs whereas OBLIVIATE shows a data independent access pattern accross runs which is indistinguishable. OBLIVIATE offers protection on two fronts here: (1) it protects the confidentiality by normalizing the query to a pre-set parameter agreed on by the application by adding dummy ORAM accesses and (2) it protects the offset that is accessed by the application.

Figure 4-(a) shows the access pattern observed in case of an in-memory filesystem. The first two page fault(s) for each query (labeled as `(a,b)` and `(d,e)`) are observed since the database first reads the metadata from the database file. Thus, the attacker knows that `(a)` and `(d)` indicate the starting of the in-memory file buffer. Using this starting point as a reference, the next page faults (labeled as `(b)`, `(c)`, `(e)`, `(f)` and `(g)`) show the offset within the file that was accessed. Based on the size of each row and position of metadata in the database, the attacker can find out which row was accessed. Also, the attacker can tell which column (i.e., the column with heart disease or without heart disease) was accessed. In the first query, shown by `(a,b,c)`, the victim attempted to reference the fourth row and first column, i.e., heart disease. In the next run, in the next run, shown by `(d,e,f,g)`, the victim attempted to access the fifth row and second column, i.e., no heart disease.

Therefore, using page faults, the attacker can tell (a) whether the same query was run or another query was run, (b) which row in the table was accessed and (c) which column within the row was accessed. It is also worth mentioning that the attacker can find out about the size of a column and row (provided they are greater than 4KB) using page fault based attack.

Cache Based Attacks. In the case of the in-memory filesystem, the enclave application is also susceptible to cache based side-channel attacks. For example, consider the Last-Level Cache (LLC), which is a unified cache that holds both code and data from the running applications. The data from the in-memory file will also be cached in the LLC and will be accessed from within the LLC. Once the application tries to access the same rows, an attacker monitoring the cache can trace the cache-sets that were `disturbed` using the `Prime+Probe` attack. Since subsequent accesses will affect the same cache-sets, he/she can build similar inferences as for the page fault based attack and compromise the security of the application.

V. THREAT MODEL

This paper assumes that a target application is running within an SGX enclave. We further assume that high-privileged system components, including the kernel, hypervisor, and BIOS, have been compromised or are adversarial. During the execution, the target enclave application accesses file resources located in the storage medium (i.e., Hard Disk Drive, Solid State Drive, USB, etc.) with the help of privileged system components, as the enclave itself does not have privilege to access such resources. In this setting, an adversarial component attempts to infer which data in a file has been accessed by the enclave application through launching side-channel attacks, namely syscall snooping attack, page fault based attacks, and/or cache attacks. This paper does not consider other types of sophisticated side-channel attacks, such as power monitoring attacks, electromagnetic attacks, and hardware bus snooping based side-channels, as most of these require direct physical accesses to the underlying SGX machine and/or are costly to launch.

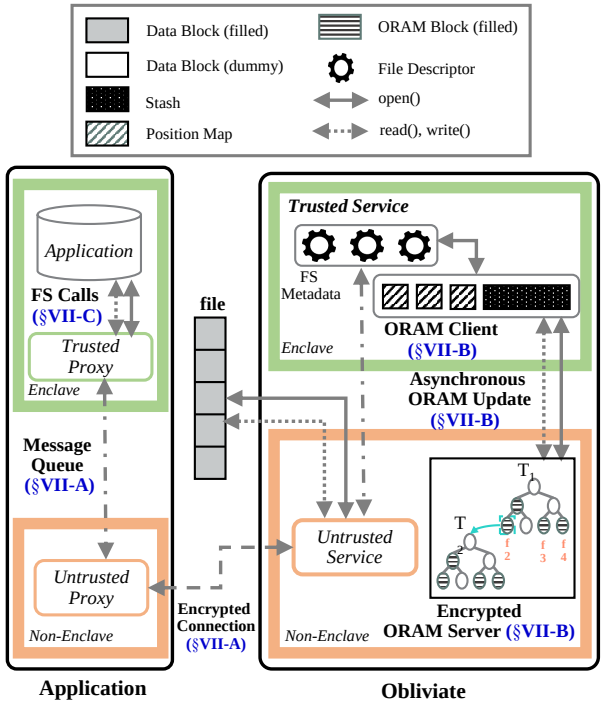


Fig. 5: A design overview of OBLIVIATE.

VI. DESIGN

Now we present the design of OBLIVIATE, a data oblivious filesystem for Intel SGX. The key idea behind OBLIVIATE is to employ ORAM-based access protocols in order to ensure that

filesystem-related operations performed by an SGX application remain confidential.

Design Overview. OBLIVIATE is a library filesystem, which runs within a separate SGX enclave, alongside the application enclave. More specifically, a filesystem enclave runs in the other process, while the application enclave relays all filesystem related operations to the isolated filesystem enclave through encrypted inter-process communication channels. While we made this design decision to minimize TCB, OBLIVIATE can be easily extended to support in-enclave filesystem (i.e., running the filesystem service and the target application in the same enclave) if needed.

Following this design principle, the overall design sketch of OBLIVIATE is depicted in Figure 5. OBLIVIATE is consisted of four components: the trusted and untrusted service, and the trusted and untrusted proxy Trusted service is running within an enclave, which performs key oblivious filesystem operations of OBLIVIATE; Untrusted service is running outside an enclave, which delegates syscall invoking operations for trusted service. These two services run in the same process, sharing the virtual address space for non-EPC memory regions. Moreover, Trusted proxy is a library linked together to an enclave application, which forwards all filesystem related system calls to the trusted service. Untrusted proxy is similar to untrusted service—it runs outside an enclave and delegates system calls for trusted proxy.

Looking at the design of OBLIVIATE from the perspective of its key component, the trusted service, it serves three major operational roles and each role is described in following subsections: (1) Connecting an enclave application with the library to receive filesystem operation requests (i.e., `open`, `read`, `write`, etc.) (§VI-A); (2) Orchestrating ORAM client and server to hide access patterns onto files (§VI-B); and (3) Managing a file descriptor and other metadata to keep the compatibility of filesystem operations (§VI-C).

A. Communication Channel for Filesystem Service

OBLIVIATE establishes a secure communication channel between an enclave application and OBLIVIATE's trusted service in order to transmit data for filesystem services. The dataflow of this communication starts from the *trusted proxy*, flowing through the *untrusted proxy* and *untrusted service* and finally reaching *trusted service* (or vice-versa). Since this communication involves untrusted components, OBLIVIATE performs end-to-end encryption to assure confidentiality of interactions. Before starting an enclave application, trusted proxy contacts trusted service to perform the initial handshaking, allowing them to share a secret key for the communication. Then all the following communication is encrypted using this secret key. OBLIVIATE uses a Diffie-Hellman [36] secret key exchange scheme, which is also used in Intel's Linux SGX SDK [5]. To prevent potential side-channel attacks on the communication layer, OBLIVIATE normalizes features related to data messages [18], i.e., all messages have a fixed size with fixed time gaps between the transmission. In addition, the application developer can predefine some parameters to ensure that each query to the *trusted service* accesses the ORAM server storage a fixed number of time or can oblige the *trusted service* to perform dummy ORAM accesses in order to mislead the attacker.

In this communication channel, there are two main types of communication: (1) communication between trusted and untrusted components (i.e., *intra-process* communication between enclave and non-enclave); and (2) communication between untrusted proxy and untrusted service (i.e., *inter-process* communication). First, to facilitate an efficient communication between trusted and untrusted components, OBLIVIATE employs exitless message queues similar to [7, 32]. A naive solution, that is also practiced by Intel SGX SDK, would be to rely on `ocalls`, but it would result in a context switch, incurring costly latency due to TLB flushes and LLC pollution [32]. To this end, OBLIVIATE implements `exitless` message queues that are established using non-EPC memory region shared between the trusted and untrusted components of both the application enclave and the filesystem enclave. Both components run its own separate thread, which keeps polling the status of the message queue. In §VIII-B, we quantify the performance improvement achieved through an exitless design over a naive design.

In addition, OBLIVIATE creates a communication channel between untrusted proxy and untrusted service. The *untrusted service* initializes a shared memory region which is acquired by the application enclave, i.e., *untrusted proxy* upon initialization. This shared memory is essentially a combination of two lock-free *single-producer, single-consumer* queues. After initialization, the *untrusted service* simply polls the *request queue* and waits for a request. As the *untrusted proxy* obtains a message from the *trusted proxy*, it simply enqueues the message into the *request queue*. The *untrusted service* receives the message from the *request queue* and passes it along to the *trusted service*. When a response is made available by the *trusted service*, the same path is followed in the reverse direction but now using the *response queue*.

B. Orchestrating ORAM Client and Server

1) ORAM Client: The client storage in ORAM comprises of two data structures, a `position map` and `stash`. ORAM assumes that these data structures are always stored within a trusted memory region, because the security critical mapping information is stored in the position map and the decrypted blocks are stored in the stash. Toward this end, OBLIVIATE stores the position map and stash within an enclave, leveraging its confidentiality guarantee (shown in Figure 5).

Position Map. The position map in ORAM helps to locate real blocks in an ORAM tree. It contains mapping information from each real block to the corresponding tree path as determined by the leaf node. Since the position map only holds the mapping information, it requires a fairly small amount of space. To be more precise, if `N` is the height of the tree, we require $2^{(N-1)} \times log(N)$ bytes of memory to hold the position map.

Stash. The stash is another security critical data structure stored in the ORAM client. Recall that the stash stores all the blocks that OBLIVIATE reads from a specific tree path in the ORAM. Unlike the position map, which is simply a mapping array, the stash is a large memory region which holds multiple blocks filled with both real and dummy data extracted from the oram tree. To be more precise, after each access, the stash is filled with at least $B \times log(N) \times D$ bytes of memory where `B` corresponds to blocks-per-node, `N` denotes the height of the tree, and `D` denotes the data size of a single block in

bytes. OBLIVIATE employs a fixed size stash configured during initialization.

Securely Accessing Position Map and Stash. While the security guarantee of an SGX enclave ensures that a potential adversary cannot directly access the position map and stash, the adversary can still launch side-channel attacks. Therefore, the adversary can observe access patterns onto these data structures, thereby inferring the hidden ORAM structures and breaking the ORAM's security model. For example, using the page fault side-channel attack, the adversarial kernel can learn page granularity (i.e., 4 KB) access patterns onto position map or stash regions. On the other hand, the cache attack can allow the attacker to gain cache-line (i.e., 64B) granularity onto these regions. This is especially harmful for the position map, as the attacker can know which index (upto 64B granularity) in the position map was accessed and consequently leak information about the corresponding block which was accessed.

To mitigate these risks, OBLIVIATE employs data oblivious algorithms [34] to access the position map and stash. In a data-oblivious algorithm, instead of accessing a specific data entity, an algorithmic operation accesses all relevant data entities (i.e., all corresponding memory pages and cache lines from OBLIVIATE's perspective). As a result, the adversary learns nothing about the operational semantics onto these data structures, since it cannot pinpoint which data entity is linked to a certain algorithmic operation. OBLIVIATE leverages the conditional move instruction (i.e., `cmov`) in the x86 architecture as a security primitive of data obliviousness. `cmov` uses a flag to distinguish between actual and dummy writes while ensuring the attacker observes the same access patterns as a regular `mov` instruction. Hence, both the position map and the stash are completely accessed irrespective of the position of the required index in the position map or the required block within the stash.

In the case of the position map, OBLIVIATE ensures that each cache-line (and consequently memory page) that corresponds to memory regions within the position maps are accessed ensuring complete privacy of access (illustrated in Figure 6). Also, OBLIVIATE has to maintain multiple position maps (in the case of multiple files) and uses the same technique to ensure that these accesses are secure. Similar to the position map, OBLIVIATE reads all candidate data blocks in the stash. If the stash is unprotected, the attacker can find out which block is real from within the path that was extracted, and consequently break the security of ORAM. For example, as illustrated in Figure 7 for the stash, the condition flag is set `true` only if OBLIVIATE copies the corresponding real block, and the flag is set `false` otherwise. From the attacker's perspective, each block or index was accessed and therefore, he/she cannot correlate the current access to a specific block. As a result, the ORAM protocol operates as if it was running in a completely isolated environment without any sort of memory leakages.

2) ORAM Server: The ORAM server stores the ORAM tree, which is updated by the ORAM client. In the following, we first describe the data structure of the ORAM server. Then we describe where the ORAM server is located and how it is accessed by the ORAM client. Lastly, we introduce an optimization technique, asynchronous ORAM server updates, which leverages a semantic gap between ORAM protocols and filesystem operations.

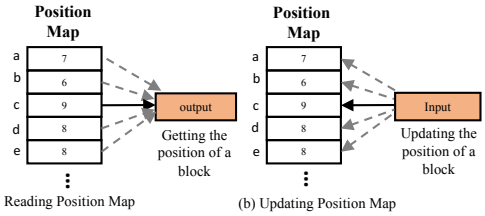


Fig. 6: Data oblivious algorithms in accessing position map. A solid line denotes using `cmov()` instruction with a true flag (i.e., actually copy the data). A dashed line denotes using it with a false flag (i.e., only impose access patterns but no copy).

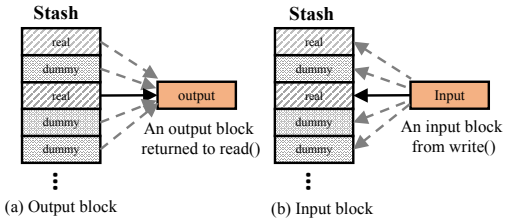


Fig. 7: Data oblivious algorithms in accessing stash.

ORAM Server Structure. As a filesystem, OBLIVIATE has to provide access to multiple files for an enclave application and it is therefore important that OBLIVIATE not only prevents the attacker from obtaining knowledge about the accessed offset within an individual file but also restricts the attacker from knowing which file was accessed. This is a common scenario, e.g., consider simple webservers such as Apache [1], Nginx [35] etc. which cater to multiple security-sensitive files. A webserver running within an SGX enclave should not leak information about what file was just accessed by the client. To provide this security, OBLIVIATE uses a simple hierarchical oram structure to load various files into its protective sphere during enclave initialization.

The hierarchical ORAM structure is a two-tiered ORAM tree. OBLIVIATE lays out all files to be used for an application into the first tier of the hierarchical tree structure. For instance, T1 in Figure 5 depicts the first tier of OBLIVIATE's tree structure, where T1 contains four different files from f1 to f4. Each filled node in T1 corresponds to a file, and an empty node represents dummy blocks in ORAM. Moreover, OBLIVIATE maintains *a filename table*, which maps a filename to a `file-block` in T1. As we will describe more on `open` syscall handling in §VI-C, OBLIVIATE employs data oblivious searching algorithms onto this table, which always streams the whole table and avoids a data-dependent access. In summary, using ORAM-based access, OBLIVIATE will access multiple blocks belonging to different files and the adversary monitoring side-channels will remain unaware about the file that was actually accessed.

OBLIVIATE allows the ORAM tree structure to be configurable by the application developer. The application developer can specify the number of leaf `K` which determines the number of files in T1, i.e., tier-1 tree in OBLIVIATE's ORAM. As each file is read into T1, OBLIVIATE sets the default data block size `D` to be used as 4KB. The value of `D` is also configurable by the program developer through separately defined API. As noted, the value of `D` is important in minimizing performance overheads since we would like to access just a single path per read or write request. The number of `Position Maps` for the ORAM

server storage is always K+1 since we have a Position Map for each of the K ORAMs and one position map serves as the filemap mentioned previously.

ORAM Server Placement. A naive solution is to place the ORAM server within the enclave memory (i.e., EPC), because this may easily leverage the security guarantee of Intel SGX. More specifically, since all data stored in EPC is automatically encrypted by SGX's memory encryption engine, this design does not need to employ an additional encryption scheme as required in the traditional ORAM protocol. However, we observe this would not a good design choice due to the hardware resource constraints imposed by SGX. SGX only allows 128 MB physical memory space for EPC. If an enclave requires more than 128 MB, the Intel SGX kernel driver [2] can swap-in and -out memory pages to extend the memory space for the enclave. The problem is, however, this swap-in and -out forces two expensive context switches (i.e., from the enclave to the kernel, and then vice-versa), degrading the performance of an enclave application as noted by [7, 32].

In order to avoid this issue, OBLIVIATE places the ORAM server in non-EPC memory with a general encryption scheme. Because non-EPC memory can be directly accessed from the enclave execution context (i.e., the ORAM client execution context), OBLIVIATE encrypts the data blocks through hardware accelerated AES scheme [37] supported in the x86 architecture. The above mentioned implementation of AES is constant-time and therefore side-channel resistant. OBLIVIATE also maintains a Merkle Hash Tree [27] in order to verify the *integrity* and *freshness* of encrypted data outside the EPC.

To maintain the server storage in a memory friendly format, OBLIVIATE constructs an array-like structure according to the pre-configured parameters for the ORAM server structure. Within this array, each node in tier-1 is placed next to each other. The nodes further contain smaller ORAM trees (tier-2 file oram trees). Because the ORAM tree structure in OBLIVIATE is a complete binary tree, this not only offers compact representations of the tree but also efficient indexing of the tree node (i.e., $O(1)$).

Asynchronous ORAM Server Update. As OBLIVIATE moves to deploy a filesystem based on ORAM operations, we observe there is an opportunity to make systematic performance optimizations. To be more specific, from ORAM's operational perspective, both read and write can be divided into three phases: (a) reading the required ORAM path, (b) processing the stash, and (c) writing back the ORAM path. Once the ORAM client completes (a) and (b), the required block has already been fetched and securely processed.

Therefore, OBLIVIATE does not need to wait until the completion of phase (c) in order to complete the filesystem operation and consequently resume the application. Instead, OBLIVIATE immediately performs the read/write operation based on the block present in the stash, and employs a separate worker thread to complete (c) in the background. This offers an opportunity for OBLIVIATE to leverage the CPU cyles in an application, which are not related to filesystem operations. During this period of time, OBLIVIATE can parallelize the write back to the ORAM path. In §VIII, we provide more evaluation results of how this optimization technique can improve performance of real-world applications.

As a result of the design decisions mentioned in this subsection, OBLIVIATE achieves a performance within 1.5×-2× of the in-memory filesystem while providing complete security. §VIII-B provides a more complete breakdown of the performance benefits achieved through these decisions.

C. Supporting Filesystem Syscall Compatibility

OBLIVIATE supports most of the native filesystem syscalls, i.e., read, write, close, etc., without requiring any changes in the enclave application layer. The rest of this section describes how we provide such compatibility by orchestrating OBLIVIATE's client and server storage. It is worth noting that one restriction of OBLIVIATE is that it does not provide concurrent access to files over the lifetime of an enclave application. While this does not introduce security issues, this can be still considered as OBLIVIATE's limitation in terms of functionality, one that we intend to achieve as part of future work.

Initializing File System. OBLIVIATE initializes all required data structures, including the client and server storage, before an enclave application starts execution (i.e., during the loading time of filesystem library). The configurable parameters such as number of files K and data-block size D are established using a manifest file agreed to by the application enclave. Since these parameters are not security-sensitive for OBLIVIATE, the manifest file can exist in plaintext. During initialization, OBLIVIATE creates the client storage (i.e., position maps and stashes) and server storage (i.e., OBLIVARRAY) according to the list of provided files. As noted in §VI-B1, this information is necessary to prevent the untrusted kernel from finding out which file is currently being accessed by the enclave application.

OBLIVIATE populates the server storage using data from non-empty files it reads in. Here, it is worth mentioning that we assume the data is integrity-protected using custom encryption which can only be decrypted by the secret key within the SGX enclave. More specifically, OBLIVIATE reads the data in each regular file per data size, and writes them to the server storage. This data population is also achieved obliviously. To do so, we again use cmov to stream through the server storage and write data blocks at random locations. We present evaluation in §VIII-B regarding the latency that data population incurs.

open(). The operational semantics of open is to return the file descriptor based on the provided file path, associated flags and mode. This file descriptor facilitates all following file system operations such as read and write. In order to create the file descriptor, OBLIVIATE first obliviously locates the data block in the first-tier of the ORAM tree structure (i.e., T1) using the given filename (i.e., using the filename table in §VI-B2). If the filename does not exist and O_CREAT is specified, OBLIVIATE assigns new (empty) block in T1 and adds it to the filename table with the corresponding filename. It should be noted here that we over-provision T1 with more leaf than required to provide support for extra files on-the-fly. Lastly, OBLIVIATE creates a file descriptor structure, and returns the reference (i.e., the file descriptor number) to the enclave application.

read() and write(). For read and write, OBLIVIATE utilizes read and write operations defined in the ORAM protocol. Using the parameters of these syscalls including the file descriptor,

OBLIVIATE first obtains the block-id in the first-tier of the ORAM tree structure, T1. Then, OBLIVIATE performs ORAM-based access (i.e., read from the path to leaf) recursively along T1-T2 to get to the required block. This process essentially involves updating both client and server storage multiple times, but it is secure against page based side channel attacks since we recursively apply ORAM protocols on each tier.

fsync(). OBLIVIATE also supports fsync requested by the enclave application. In order to preserve which parts of the applications have been written to, we simply write-back the whole file. OBLIVIATE always writes back into the regular linux file type to support compatibility with other systems or applications.

close(). `close` closes a file descriptor, which may or may not flush the data buffers. This deferred flush does not cause a consistency issue in the traditional OS, because the OS implements a global buffer and all file accesses are always performed using this un-flushed buffer. In the current version of OBLIVIATE, a final write-back is performed when the library enclave is terminated. OBLIVIATE can support an encryption such that written-back files retain their data confidentiality. OBLIVIATE simply uses the hardware sealing key provide by SGX as a key for the encryption.

Other syscalls. Based on above four syscall implementations, we have added most of basic file system related syscalls, including `read`, `readv`, `pread`, `preadv`, `write`, `writev`, `pwrite`, `pwritev`, `lseek`, `access`, `stat`, etc. We believe the above syscalls are elemental functions, especially with respect to securing the file access information, and we were able to run realistic real-world applications including SQLite. We leave the implementation of the rest as our future work.

VII. IMPLEMENTATION

In this section, we describe implementation details of OBLIVIATE. On the whole, OBLIVIATE's filesystem library is implemented on Intel SGX SDK [5], an open-source development environment provided by Intel to develop SGX applications. In terms of implementation complexity, OBLIVIATE's trusted service library consists of around 1987 lines of code whereas the untrusted service consists of 454 lines of code. At the application enclave side, we modify Graphene's LibOS in order to establish the communication channel and exitless message queues. In total, this required around 685 lines of code addition to Graphene's LibOS. It should be mentioned here that Graphene's LibOS is just one of the example LibOS that can be used with OBLIVIATE. Depending on the application, the developer can choose more TCB-friendly solutions such as Panoply [44] or even Intel SGX SDK [5].

As dictated by OBLIVIATE's ORAM tree structure (§VI-B2), the client storage (position maps and stashes) are stored in an enclave memory. OBLIVIATE implements and maintains these data structures in its OBLIVSHIM, which is as an interface within the trusted service. OBLIVIATE's *trusted service* maintains the server storage outside the EPC but within the confines of its application boundary. The test applications, running with Graphene's LibOS, direct all syscalls to the *trusted service* of OBLIVIATE. At the time of `open`, the *trusted service* creates and maintains the complete file handling information including file descriptors, file offsets, etc. The *trusted service* provides seamless transition from the application's perspective onto the server storage.

Attack	Attacking Vectors	Defense Mechanism of OBLIVIATE
Syscall	File and File-offset	FS metadata in enclave (§VI-C)
PF/Cache	File-offset	ORAM operations (§VI-B2)
PF/Cache	File	Two-tiered ORAM tree (§VI-B2)
PF/Cache	Block-id from position map	Data oblivious schemes (§VI-B1)
PF/Cache	Real/dummy block from stash	Data oblivious schemes (§VI-B1)

TABLE II: A list of attack vectors and their corresponding defense mechanisms of OBLIVIATE.

VIII. EVALUATION

In this section, we begin with a security analysis of OBLIVIATE. Next, we provide a detailed performance benchmarking using both benchmarking tools and real-world applications.

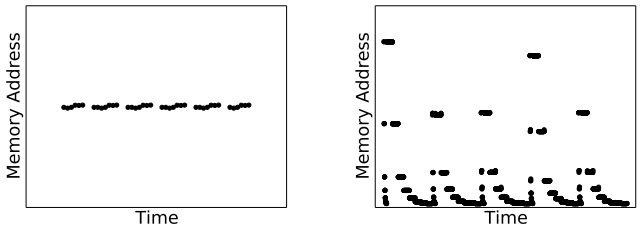
Experimental Setup. All our evaluations were performed on Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz (Skylake with 4 `MB` cache) with 16 `GB` RAM (128 `MB` for EPC). We ran Ubuntu 16.04 with Linux 4.4.0.59 64-bit. The applications running on OBLIVIATE employ Graphene LibOS to run but as mentioned before, OBLIVIATE only uses the Intel SGX SDK [5].

Reference Filesystems. Since porting applications on SGX without a LibOS is challenging, we do not port applications into a naive SGX FS (refer §III-A) but present results based on our experimentation with native (non-SGX) FS (labeled as Native FS). To gauge the performance of in-memory SGX-based FS (refer §III-B), we developed a reference in-memory file system based on Graphene's LibOS [48] (labeled as In-memory FS). The hybrid SGX-based file system (refer §III-C) is the defacto file system used by Graphene LibOS [47, 48] (labeled as Hybrid FS).

A. Security Evaluation

In order to ascertain the protection of OBLIVIATE against the side-channel attacks (mentioned in §IV), we provide an in-depth security analysis followed by experimental evaluation.

Security Analysis. Table II provides a brief overview of possible attack vectors against OBLIVIATE and the defense that OBLIVIATE provides to mitigate these attacks. Since all metadata and file buffer handling is performed by OBLIVIATE within the enclave, syscall snooping attacks do not leak any information. The key challenge is how to mitigate the risk of page fault based and cache-based attacks on OBLIVIATE. As far as these attacks on the file are concerned, OBLIVIATE's security guarantees stem from the security guarantees of ORAM. ORAM ensures that, regardless of access semantics, each access exhibits a different memory access traces to an attacker. Because this is the underlying assumption for both page fault and cache based attacks, OBLIVIATE is provably secure against these. For example, consider a victim enclave attempting to access a file at an offset `X` twice. An in-memory filesystem would exhibit the same memory access patterns both times whereas ORAM



(a) In-memory FS **(b)** Obliviate

Fig. 8: Runtime memory access patterns (page faults) captured by the SGX driver. As expected, the in-memory FS exhibits the exact same memory footprint since the same offset is being accessed. OBLIVIATE conversely shows different memory footprints on each access.

would exhibit a different pattern. An attacker observing the different pattern on the second run cannot tell whether this was the same offset or another offset within the file. Furthermore, with OBLIVIATE, the attacker cannot even tell whether this was the same file or a different file as OBLIVIATE employs two-tiered ORAM tree. Therefore, an attacker cannot obtain a meaningful context by observing the access pattern onto the ORAM server storage.

As mentioned in §VI-B1, naively deploying ORAM within an SGX enclave leaves an attack surface to side-channel attacks. This is true for both the position map and the stash which are key metadata structures for ORAM. Elaborating further, the attacker can pinpoint the block being accessed by observing which index within the position map is accessed. The attacker can also know which block is real and which is dummy by observing which block was actually copied onto the provided buffer. The data oblivious schemes used by OBLIVIATE ensure that both of the aforementioned transactions always exhibit the same trace on each run, preventing potential inferences by an attacker §VI-B1. Since each access will follow the same path, i.e., from the start to the end of the data structure, an attacker is again provided with no information that could help him/her.

Experimental Security Evaluation. In order to show how OBLIVIATE leaves memory access patterns with its ORAM implementation, we show observed page faults for OBLIVIATE. Recall that, when demonstrating our attack case study using SQLite in §IV, we have also provided OBLIVIATE's memory trace for comparison (shown in Figure 4-(b)). In this figure, while the in-memory FS showed the same access pattern for the query with the same semantics, OBLIVIATE showed different access patterns for those due to ORAM based operations. From attacker's perspective, this implies that she/he would not be able to infer query semantics based on the access patterns.

In addition, we also performed another experiment which attempts to access the same file offset multiple times. Then we captured the corresponding memory access patterns made in an enclave, using the SGX driver (which is part of the untrusted kernel). Figure 8 shows the results on the in-memory filesystem as well as on OBLIVIATE. We can observe the following: (a) OBLIVIATE exhibits more page faults than the in-memory FS and (b) OBLIVIATE's execution pattern is different on each access whereas the in-memory filesystem leaves the same memory footprint. ORAM-based access dictates that

	16M	128M	512M	1G
Open	3,145	7,200	14,765	21,624
Populate	1,990	5,100	7,878	12,323
Write-back	2,967	4,900	10,907	16,635

TABLE III: Performance of open and write-back operations in OBLIVIATE (in milli-seconds): **Open** captures the complete time that it takes to open the file, populate data from original file and allocate space for `Position Map` and `Stash`; **Populate** is the taken time to write real blocks to the server storage; **Write-back** corresponds to the time taken to write-back to a regular file format.

	16M	128M	512M	1G
IPC overhead	368	1322	2298	4235

TABLE IV: Time taken to pass various messages from *untrusted proxy* to *untrusted service* through shared memory channel in chunks of 4KB for OBLIVIATE (in milli-seconds).

multiple blocks have to be read to access a block, which naturally results in a higher number of overall page faults (considering the attacker invalidates all pages). Also, since the experiment attempts to access the same index, we observe that each access using OBLIVIATE leaves a different memory footprint to be observed by the attacker. This is in accordance with the principles of ORAM which ensure that access patterns are completely indistinguishable from each other. As a result, the attacker is blinded as to whether the same file offset was accessed or a different one.

B. Micro-Benchmarks

We attempt to answer the following questions through this subsection: (1) How does OBLIVIATE compare in performance to the other available filesystems?; (2) What is the degree of improvement observed when using an exitless scheme over a naive scheme?; (3) What is the performance improvement achieved by using the non-EPC memory for ORAM server placement instead of the EPC memory?; (4) What is the latency added by the communication channel on the overall performance?; and (5) What is the memory overhead imposed by OBLIVIATE over other available filesystems?

To this end, we show the results of sequential and random `read`/`write` operations and also discuss OBLIVIATE's overhead on `open` and `close` operations. We use Iozone [30], which is an open-source tool, widely used to benchmark filesystem performance. Iozone provides throughput numbers which are amortized over different runs. Since it is designed specifically in order to gauge the performance of a filesystem, we evaluate OBLIVIATE using Iozone. In order to facilitate the reader, we present a component-wise breakdown of the numbers.

Open/Close. Table III depicts the performance overhead (in terms of seconds) imposed in opening and closing the file by OBLIVIATE. Firstly, OBLIVIATE has to transform the regular file into an ORAM tree layout, OBLIVARRAY. This step involves: (a) creating the tree and (b) populating the tree with data from the original file. OBLIVIATE incurs overheads ranging from 2-20s to complete these tasks depending on the file size. This overhead is unavoidable since a regular file cannot be processed with ORAM-based access protocols. At the time of finishing the file uses (i.e., exiting an enclave), OBLIVIATE

writes back the contents to the regular file, which takes from 3 to 16s. The whole file is written back instead of the modified parts in order to preserve privacy.

Read/Writes. Figure 9 shows the read/write throughput achieved in running iozone. As expected, the native (non-SGX) FS is the most efficient one. The hybrid FS is slower than the native FS but is not as slow as the complete in-memory FS or OBLIVIATE. The reason for this is that the hybrid FS stores the file buffers in the non-EPC memory region (but within DRAM) and only copies in-use pages inside the EPC memory. The in-memory FS is slow since it buffers all file contents within the EPC memory and therefore competes with the LibOS and user application for limited EPC pages. This contention of limited EPC memory result in abundant swap-ins/outs which incur considerable overhead [32].

OBLIVIATE performs 1.5-2.0× worse than the in-memory FS for our workloads. The overhead of OBLIVIATE is unavoidable since it uses expensive ORAM operations and data oblivious algorithms to provide complete security. However, since OBLIVIATE uses the non-EPC memory (with custom encryption) to store the server storage along with other optimizations, it is able to compete with the in-memory FS. It can also be observed that the throughput of both native FS and hybrid FS increases with the increase in file size whereas the throughput of in-memory FS ans OBLIVIATE decreases. For in-memory FS, that is because there is more contention in the EPC memory region whereas for OBLIVIATE, it is simply because we have to perform operations on a larger ORAM tree.

As far as the overhead imposed by OBLIVIATE is concerned, we believe that this is the overhead that any ORAM-based solution is likely to incur. This is justified since ORAM has to access multiple (say N where 2^N is the height of the tree) blocks per memory access and has to write-back the same number of blocks. Therefore, OBLIVIATE has to access a total of $2N$ blocks compared to a single block that has to be accessed by an insecure filesystem.

Optimization Effectiveness: Message Queues and Non-EPC Server Placement. In order to show effectiveness of OBLIVIATE's optimization techniques, namely message queues (§VI-A) and non-EPC ORAM server placement schemes (§VI-B2), we quantize the performance improvements over the native scheme of each optimization technique. First, for OBLIVIATE's message queues, we modified OBLIVIATE to use `ocall` mechanism in order to perform message passing instead of using message queues. Figure 10a shows the results we obtained while running a simple random read over a range of data using iozone. As shown, the message queues provide a performance improvement of $20 - 40\%$ over the naive `ocall` scheme. It has been reported previously [7, 32] that enclave exits are expensive because of context switches and TLB flushes.

To show the effectiveness of using the non-EPC storage as a medium to store the ORAM server storage, we perform an experiment where OBLIVIATE uses either EPC or non-EPC memory region (with our own encryption scheme) in order to store the ORAM server storage. Figure 10b provides a comparison of achieved throughput in both scenarios using Iozone. Since the EPC memory region is small and thus incurring costly swap-in and swap-out, OBLIVIATE's optimization schemes show much better throughput. For a 16MB file, the throughput difference is around 1.25× which reaches to more than 9× as the file size increases to 1GB.

IPC Overhead. In Table IV, we compare the overheads of *inter-process communication* as we send messages of different sizes in chunks of 4KB. OBLIVIATE uses shared memory queues in order to create an IPC channel between the untrusted proxy and untrusted service (refer §VI-A). As can be observed, the overhead of shared memory communication increases linearly to the size of the message. However, there are two things to consider here. Firstly, this cost (per individual message) is negligible as compared to the cost of a single ORAM access. Secondly, OBLIVIATE can be easily adapted to act as a filesystem which is bundled with the same application, which will remove the extra latency added by this communication channel. However, this design choice would have to abandon the security principle, the principle of separation.

Memory Overhead. OBLIVIATE requires more memory than a complete in-memory filesystem since it has to create a multi-tier ORAM tree in-memory. Our evaluations show that OBLIVIATE's ORAM-based tree consumes around 6-8× more memory than the actual file size it tries to map into its tree.

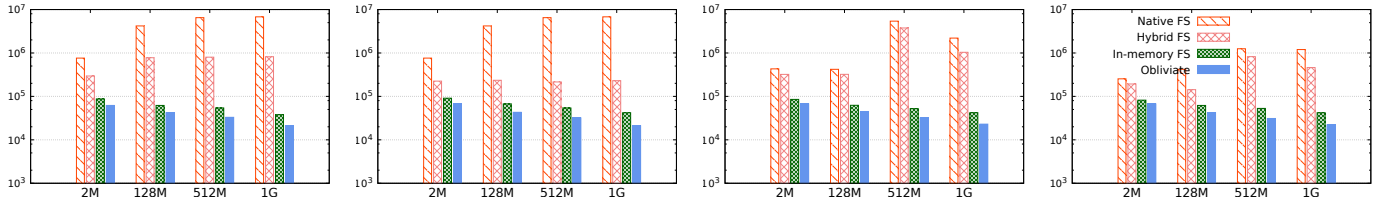
C. Macro-Benchmarks

This subsection evaluates OBLIVIATE and other filesystems in running real-world applications, SQLite [33] and Lighttpd [22]. We chose these applications since both are inherently more I/O-intensive than CPU-intensive. To create a comparison against all SGX-based filesystems, we show the results from in-memory filesystem, hybrid filesystem, and OBLIVIATE.

SQLite [33] is a popular open-source database application. It frequently used to access database files to process SQL queries which rely on `open, read, write`, etc. to fetch/update data. Figure 11a depicts the results in running SQLite. In this experiment, we run `speedtest`, a stressed performance testing script included in SQLite. This inserts 50,000 entries into the database and attempt to select the 50,000 entries that we inserted. The results of our experiments show that OBLIVIATE incurs an overhead of approximately 4× over the hybrid filesystem and approximately 1.5× over the in-memory filesystem.

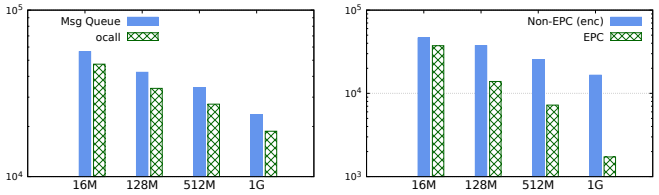
Lighttpd [22] is a popular light-weight web server. It fits OBLIVIATE's criteria since it performs heavy I/O intensive jobs (i.e., reads many files from its memory and transmits them to the client). Our tests use simple workloads that are observed as part of web search operations in datacenters [6]. These workloads show that most of the flows are within a range of 1 KB and 1 MB, and therefore our testing is uniformly within this range. As shown in Figure 11b, OBLIVIATE is not as fast as in-memory and hybrid file systems, but its overhead is low. To be more specific, OBLIVIATE exhibits less than 1.2× the overhead of the in-memory filesystem and around 2× overhead over the hybrid filesystem.

In all our evaluations, we see that a baseline ORAM solution would add a fair amount of overhead to existing filesystem solutions for SGX. Since OBLIVIATE employs an ORAM based access mechanism which guarantees security but is expensive in nature, it is well expected that OBLIVIATE would perform



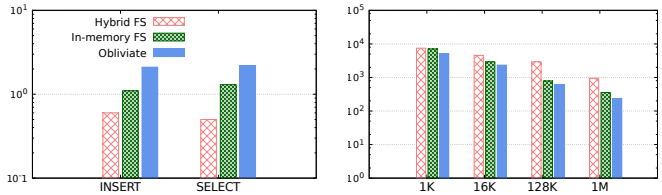
(a) Sequential reads **(b)** Sequential writes **(c)** Random reads **(d)** Random writes

Fig. 9: Iozone benchmark results while varying the file size. X-axis represents a file size, and Y-axis represents the throughput achieved in KB/s. For each experiments, we read 4KB chunks. The parameters for OBLIVIATE are K = 1, B = 3, and D = 4096. The value of leaf nodes is calculated based on the value of D and filesize.



(a) Exitless Vs OCALL **(b)** EPC Vs Non-EPC

Fig. 10: Effectiveness of OBLIVIATE's optimization techniques. The figure (a) depicts the the latency (y-axis) while varying the the data size to be transmitted (x-axis) for exitless scheme [32] and a naive `ocall` scheme. The figure (b) shows the throughput (y-axis) to store the ORAM server storage using either EPC or encrypted non-EPC.



(a) SQLite (secs) **(b)** Lighttpd (Req/secs)

Fig. 11: Runtime performance of SQLite and Lighttpd. For SQLite, the value of K = 1, B = 3, and D = 4096. For Lighttpd, K = 8, B = 3, and D = filesize. Since, lighttpd will read the whole file in anycase, it makes sense to simply store it as a single block. OBLIVIATE prevents the attacker from knowing which file was accessed.

worse than non-secure solutions. However, due to careful design decisions, we ensure that OBLIVIATE's overhead is less than twice that of the in-memory filesystem, a degree of improvement over a baseline ORAM scheme, while it provides complete security. We also believe that OBLIVIATE can be practically used with applications which are more security-critical than performance-critical such as the ones mentioned in §IX-A.

IX. DISCUSSION

A. *Potential Applications of* OBLIVIATE

In this section, we attempt to illustrate the applications that could serve as potential use cases for OBLIVIATE.

Cloud-based Storage. Previous work [8] has proposed using write-only ORAM from a remote user side, in order to achieve secure write-back into a cloud backup storage such as Dropbox, Google Drive etc. However, their work assumes that all storage is kept local and simply updated to keep the latest copy on Dropbox. By using OBLIVIATE, the user can securely store and retrieved his/her data solely on the server since OBLIVIATE employs ORAM at the server side.

Databases. Cloud-based database services especially storing personal information merit the use of SGX. As can clearly be inferred from §IV, database systems running within SGX are insecure. This security issue can be mitigated using OBLIVIATE. As we evaluate in §VIII-C, employing OBLIVIATE with a database system such as SQLite [33] can ensure security with acceptable overhead (around 2× that of an in-memory FS).

Web Servers. Naive application of webservers (e.g., Nginx [35], Apache [1] etc.) would leak which webpage was accessed by which user (through correlating with the IP address). OBLIVIATE protects both file offsets and which file was actually accessed from being leaked and is therefore a good fit for security-critical web servers.

B. *Other Side-Channel Attacks.*

It is difficult to completely prevent side-channel attacks, especially if the attacking method is not known beforehand. Although OBLIVIATE's security guarantees can be also broken due to new side-channel attack methods in the future, we still believe the security primitives that OBLIVIATE provides are general (i.e., exhibiting non-deterministic memory access patterns in accessing files) and thus OBLIVIATE would be able to raise the protection bar of against those attacks. For example, a new side-channel attack against SGX, *branch shadowing* [23] attack, was reported recently. The branch shadowing attack exploits the fact that when an SGX enclave context switches from the EPC to the non-EPC memory, it leaves its branch information uncleared. Using such information, the attacker can gain fine-grained information into the internal workings of the SGX enclave. However, OBLIVIATE's ORAM based access is still secure against such attacks since the underlying assumption with ORAM is that the attacker can see all the memory accesses being performed and yet gain no information. Moreover, the data oblivious access schemes used by OBLIVIATE ensure that, despite fine-grained observation into the ORAM metadata, the attacker is able to learn nothing.

X. RELATED WORK

Attacks against Intel SGX. There are three main side-channel attacks that plague Intel SGX — syscall based, page fault based, and cache based. Unlike syscall snooping, which is a passive

attack on system call interaction, IAGO attacks [12] explores the security implications of trusted systems (like SGX) relying on untrusted syscalls. OBLIVIATE counters IAGO attacks by loading initially encrypted files and mantaining their freshness and integrity through Merkle Hash Trees (refer §VI-B2). Page fault attacks [50] show how data and code based page faults can be used to learn about the execution pattern of an application within an enclave. Cache attacks [10, 40] have shown that both L1 and LLC can be used to mount a successful cache attack on Intel SGX. Recently, branch shadowing attack [23], has been reported to leak fine-grained information from SGX enclaves by exploiting uncleared branch history when there is a context switch from enclave to non-enclave mode. In §IX-B, we discuss how OBLIVIATE is also secure against this attack.

Side-Channel Defenses for SGX. Previous works attempted to prevent (or their design stops) IAGO attacks, which includes syscall based side-channel attacks as well [9, 18, 47, 48]. Focusing on protecting file resources, these works implement an in-memory filesystem inside the SGX enclave in order to hide file-related syscall parameters from the untrusted kernel.

There are several works which can be leveraged to stop page fault based attacks. First, address space layout randomization (ASLR) can be adopted for the SGX environment [41]. ASLR will make it difficult to understand memory layout and therefore file access patterns in case of in-memory FS. However, the memory address will always be the same once an application is launched, repeated page fault information will eventually allow an attacker to decipher the underlying memory layout. T-SGX [42] attempted to solve page fault based side-channels by utilizing Transactional Synchronization Extensions (TSX). Using T-SGX, an enclave application can directly receive all page fault events ahead of the kernel handler. The key limitation of T-SGX is that, as demonstrated in [11], it is vulnerable to attackers who keeps track by keeping track of the access/dirty bit in a page table, which effectively learns about memory page access information. Another work [43] was proposed to mask page fault patterns from revealing information by making deterministic modifications of the programs memory access pattern. However, their evaluation deals with smaller applications (cryptographic in nature), involving a small number of memory pages. In contrast, the filesystem handles far more pages which would result in a very large overhead for an in-memory filesystem employing their scheme. We also note that their scheme cannot protect against cache attacks. Another work [31] proposed a way to process Machine Learning (ML) algorithms in a data oblivious manner. In contrast to OBLIVIATE, their work focuses on redesigning specific ML algorithms.

Cache attacks have been abused to exploit cryptographic keys such as AES and RSA. There are various solutions [14, 21, 51] that have been proposed in order to secure programs against non-SGX cache attacks. However, these will not work with SGX since most of them require a trusted OS and/or are prohibitively expensive. There are also various proposed hardware solutions [25, 49].

Hardware-based Defenses against Side-Channels. Various hardware solutions [24, 26, 29] have been proposed to mitigate the risk of access pattern based attacks. In contrast to these, OBLIVIATE is more practical since it does not impose hardware changes. In the future, the performance issues of OBLIVIATE can be also reduced if hardware changes can be permitted (especially for hardware-based ORAM operations). Previous solutions [26, 29], have shown that hardware-based ORAM schemes offer less overhead than software-based schemes.

SGX-based Systems. Haven [9] provides a Windows-based LibOS for the SGX to run unmodified binaries in an enclave. Graphene [47, 48] similarly provides a Linux-based LibOS. Panoply [44] applies the principle of separation for LibOS. Ryoan [18] provides a sandbox for running applications that are shared amongst mutually untrusted parties. VC3 [39] aims to provide trusted analytics using Intel SGX in an untrusted cloud. Scone [7] devises a mechanism to support asynchronous system calls [45] for SGX and improves enclave performance using user-level threading. Eleos [32] provides user-level paging in order to prevent costly enclave exits. OpenSGX[20] provides an open architecture for SGX research. As we design OBLIVIATE for SGX environments, some of its design are inspired by above mentioned work — for example, principle of separation design from Panoply [44], message queues from Scone [7], and extended secure memory region using non-EPC from Eleos [32].

ORAM-based Systems. TaoStore [38] proposes a design to share a single ORAM-tree structure. Raccoon [34] is another work that aims to prevent executing programs from side-channel attacks by obfuscating the execution patterns in non-SGX environments. OBLIVIATE has adopted Raccoon's data oblivious memory copy using `cmov`. Oblivisync [8] uses write-only ORAM to secure synchronization of local data with Dropbox service.

XI. CONCLUSION

This paper presented OBLIVIATE, a data oblivious file system for Intel SGX. In response to system call or page fault based side-channel attacks, OBLIVIATE adopts an ORAM protocol in accessing files for an SGX environment. The evaluation using the prototype of OBLIVIATE demonstrated its effectiveness in securely running large-scale applications such as SQLite and Lighttpd.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers of the program committee of NDSS 2018 for their insightful comments on this work.

REFERENCES

[1] "The apache http server project," 2017. [Online]. Available: https://httpd.apache.org/

[2] "Intel(r) sgx linux driver," 2017. [Online]. Available: https://github.com/01org/linux-sgx-driver

[3] "Overview of Intel Protected File System Library Using Software Guard Extensions," 2017. [Online]. Available: https://software.intel.com/en-us/articles/overview-of-intel-protected-file-system-library-using-software-guard-extensions

[4] *Intel Software Guard Extensions Programming Reference (rev2)*, Oct. 2014.

[5] 01org, "Intel(r) software guard extensions for linux* os (source code)," 2016. [Online]. Available: https://github.com/01org/linux-sgx

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM computer communication review*. ACM, 2010.

[7] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O Keeffe, M. L. Stillwell *et al.*, "Scone: Secure linux

containers with intel sgx," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[8] A. J. Aviv, S. G. Choi, T. Mayberry, and D. S. Roche, "Oblivisync: Practical oblivious file backup and synchronization," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[9] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017.

[11] 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 *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug. 2017.

[12] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[13] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 965–981, 1998.

[14] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2009.

[15] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[16] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, 1996.

[17] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx." in *EUROSEC*, 2017, pp. 2–1.

[18] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[19] Intel, *Intel(R) Software Guard Extensions SDK for Linux* OS*, 2016, https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

[20] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han, "Opensgx: An open platform for sgx research," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[21] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[22] J. Kneschke, "Lighttpd," 2003.

[23] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug. 2017.

[24] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2015.

[25] F. Liu and R. B. Lee, "Random fill cache architecture," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Cambridge, UK, Dec. 2014.

[26] 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 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Hangzhou, China, May 2013.

[27] R. C. Merkle, "Method of providing digital signatures," 1982.

[28] T. Moataz, E.-O. Blass, and G. Noubir, "Recursive trees for practical oram," *Proceedings on Privacy Enhancing Technologies*, 2015.

[29] K. Nayak, C. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal, "Hop: Hardware makes obfuscation practical," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[30] W. D. Norcott and D. Capps, "Iozone filesystem benchmark," 2003.

[31] 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 Security Symposium (Security)*, Austin, TX, Aug. 2016.

[32] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, Apr. 2017.

[33] M. Owens and G. Allen, *SQLite*. Springer, 2010.

[34] A. Rane, C. Lin, and M. Tiwari, "Raccoon: closing digital side-channels through obfuscated execution," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[35] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[36] E. Rescorla, "Diffie-hellman key agreement method," 1999.

[37] J. Rott, "Intel advanced encryption standard instructions (aes-ni)," *Technical Report, Technical Report, Intel*, 2010.

[38] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, "Taostore: Overcoming asynchronicity in oblivious data storage," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[39] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[40] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*.

[41] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[42] 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)*, San Diego, CA, Feb. 2017.

[43] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing your faults from telling your secrets: Defenses against pigeonhole attacks," in *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–Jun. 2016.

[44] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[45] L. Soares and M. Stumm, "Flexsc: flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[46] 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 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[47] 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 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[48] C.-C. Tsai and D. E. Porter, "Graphene library os with intel sgx support," 2017. [Online]. Available: https://github.com/oscarlab/graphene

[49] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proceedings of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Lake Como, Italy, Nov. 2008.

[50] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[51] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.