OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX

Adil Ahmad*† Byunggill Joe*‡ Yuan Xiao§ Yinqian Zhang§ Insik Shin‡ Byoungyoung Lee†¶

†Purdue University ‡KAIST §Ohio State University ¶Seoul National University

ahmad37@purdue.edu byunggill@cps.kaist.ac.kr {xiao.465, yinqian}@cse.ohio-state.edu
insik@cs.kaist.ac.kr byoungyoung@snu.ac.kr

***Abstract*—Program obfuscation is a popular cryptographic construct with a wide range of uses such as IP theft prevention. Although cryptographic solutions for program obfuscation impose impractically high overheads, a recent breakthrough leveraging trusted hardware has shown promise. However, the existing solution is based on special-purpose trusted hardware, restricting its use-cases to a limited few.**

In this paper, we first study if such obfuscation is feasible based on commodity trusted hardware, Intel SGX, and we observe that certain important security considerations are not afforded by commodity hardware. In particular, we found that existing obfuscation/obliviousness schemes are insecure if directly applied to Intel SGX primarily due to side-channel limitations. To this end, we present OBFUSCURO, the first system providing program obfuscation using commodity trusted hardware, Intel SGX. The key idea is to leverage ORAM operations to perform secure code execution and data access. Initially, OBFUSCURO transforms the regular program layout into a side-channel-secure and ORAM-compatible layout. Then, OBFUSCURO ensures that its ORAM controller performs data oblivious accesses in order to protect itself from all memory-based side-channels. Furthermore, OBFUSCURO ensures that the program is secure from timing attacks by ensuring that the program always runs for a pre-configured time interval. Along the way, OBFUSCURO also introduces a systematic optimization such as register-based ORAM stash. We provide a thorough security analysis of OBFUSCURO along with empirical attack evaluations showing that OBFUSCURO can protect the SGX program execution from being leaked by access pattern-based and timing-based channels. We also provide a detailed performance benchmark results in order to show the practical aspects of OBFUSCURO.

I. INTRODUCTION

Program obfuscation [1, 2] is a popular cryptographic construct which has interesting and wide-ranging applications towards protecting the intellectual property of software owners. As computing trends are rapidly shifting towards cloud-based computing, there exists a strong need for systems supporting this notion of program obfuscation. One could envision various cases where the owner of a program would want to shield his/her proprietary algorithm from the cloud provider and/or other tenants. For example, consider a company like 23andMe [3], the frontrunner in DNA testing, which could prevent the theft of their algorithm from competitors despite it being hosted on cloud servers.

Under program obfuscation, a sender, who owns a program, transforms it to create an obfuscated version of the program which is: (a) functionally identical to the original version, and (b) runs for a fixed time before returning an output. The sender then sends this obfuscated program to a receiver. The receiver runs the obfuscated program within a black box-like environment — the receiver cannot see (or infer) intermediate computational results and/or footprints from the obfuscated program. Consequently, even though the receiver can run the obfuscated program using any input of his/her choice, he/she will learn nothing about the original program. Therefore, as far as the attacker is concerned, he/she is interacting with a virtual black box, which takes an input and gives the intended output.

In the past, there has been significant (mostly cryptographic) research [4–7] in achieving program obfuscation, but with crippling performance overheads. Recently, there has been a systematic breakthrough, HOP [8], in achieving program obfuscation through relaxed assumptions of trust on the underlying hardware. However, HOP relies on special-purpose hardware, severely limiting its practicality. In particular, their system relies on custom RISC-V processors to *conveniently* transplant the root of trust to implement the core security logic and securely contain the program code. We believe such convenience is not free — it would be challenging and unrealistic to deploy such custom-built hardware to a majority of end-user machines or cloud-computing machines.

In this paper, we propose OBFUSCURO, the first system achieving program obfuscation on commodity hardware. Unlike existing work relying on special-purpose hardware, OBFUSCURO is specifically designed to run on Intel SGX, already shipped with millions of machines in the market. Since the trusted boundary of Intel SGX terminates at the CPU, OBFUSCURO enforces the security protocol of Oblivious RAM (ORAM) [9] to support secure code/data access between CPU and memory subsystems, similar to HOP. However, it is quite challenging to support program obfuscation on commodity hardware since commodity hardware comes pre-packaged with a plethora of features which can be abused to invalidate a key security assumption behind program obfuscation (i.e., the

⋆ The two lead authors contributed equally to this work.

‡ The author did part of this work while visiting Purdue University.

The corresponding author is Byoungyoung Lee (byoungyoung@snu.ac.kr).

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23513
www.ndss-symposium.org

obfuscated program should be running within a black box-like environment). Furthermore, the unprivileged execution of Intel SGX ensures that these features cannot be controlled (or disabled) by SGX programs.

More specifically, researchers have identified that Intel SGX has critical access pattern-based side-channel security flaws. These allow adversaries to infer computational semantics within SGX thereby breaking the black box execution environment. Memory-based side-channels, namely page fault [10, 11], cache [12–14], and branch-prediction [15] attacks, allow system components with high privileges, e.g., OS, to infer substantial information from the execution of an SGX enclave. For example, previous work [13] has shown how cache attacks can be abused to leak an RSA private key from an SGX enclave.

As far as access pattern-based side-channels are concerned, the root cause of the problem is that it is challenging to completely hide memory access patterns from privileged adversaries in the current Intel SGX architecture. The reason for this is that the CPU is designed to rely on other subsystems to perform computation. In particular, Intel SGX is not designed to secure communication patterns between the CPU and memory-management hardware units (e.g., the MMU/TLB, cache, DRAM, branch-predictors, etc.). For performance reasons, the communication channels and hardware units are designed to be *partially* shared between trusted and untrusted entities, allowing potentially adversarial entities to observe and collect memory traces exhibited by an SGX enclave.

To address these challenges, OBFUSCURO[1] makes use of three main ideas. First, OBFUSCURO employs a *data-oblivious* ORAM implementation. Our work improves on the previously proposed secure ORAM implementations [16–18] by designing an efficient *register-based* stash. Second, OBFUSCURO designs side-channel resistant *scratchpad*-based code execution and data access models, in order to neutralize the memory access patterns observed by attackers as well as bridge the gap between traditional ORAM and native program execution. Lastly, OBFUSCURO ensures *start-to-end* obfuscation of the target programs by providing execution time normalization to all applications thereby protecting the programs against information leakage through timing-based channels.

Our implementation of OBFUSCURO is based on the LLVM compiler suite with an installed runtime library. Through compiler instrumentation, we transform a native SGX program's code into cache-line-granular (and ORAM-compatible) basic blocks. OBFUSCURO restricts each basic block to a single data and code access, at fixed offsets within the basic blocks thereby neutralizing branch targets. The code and data access instructions are translated into equivalent branch instructions targeting OBFUSCURO's runtime library functions. The runtime library *obliviously* serves the program with code and data blocks extracted from the ORAM storage onto pre-allocated memory regions called C-Pad and D-Pad respectively. Code execution and data access (irrespective of the target program) is always performed at these locations, thereby neutralizing the program's memory footprints. Lastly, the program is instrumented to keep executing till a user-configured time interval has elapsed to mitigate the threat of timing channels.

[1]OBFUSCURO is a play on words combining Obscuro and Obfuscation. The former is a memory charm in the Harry Potter series.

Furthermore, we highlight that although OBFUSCURO's performance overhead is quite high, it is still much faster than the state-of-the-art cryptographic obfuscation schemes. In particular, cryptographic obfuscation techniques (which rely on homomorphic encryption and/or circuit construction as security primitives) are still far away to be adopted in practice largely due to severe performances overheads or limited generality to support generic programs (detailed discussion in §IX). However, leveraging the root of trust in the underlying commodity hardware, OBFUSCURO demonstrates *comparatively* moderate performance overheads on real-world programs.

In broad terms, the contributions made by this paper can be described as follows:

- We dissect *commodity-off-the-shelf* hardware to find out the key hardware features which hinder the adoption of program obfuscation in Intel SGX. We also provide a comparison with existing work illustrating how their approaches are insecure if directly applied to Intel SGX.
- We present, OBFUSCURO, the first program obfuscation system built on top of commodity hardware. Motivated by the hardware limitations of Intel SGX, OBFUSCURO provides a complete *start-to-end* program obfuscation solution which can be readily-adopted without any modifications to legacy code written for Intel SGX.
- We provide a thorough security analysis of OBFUSCURO showing how it can prevent information leakage through both access pattern-based and timing-based side-channels.
- We provide a performance comparison of OBFUSCURO using a diverse set of benchmarking applications as well as a real-world application, OpenSSL. Our experiments indicate that OBFUSCURO incurs an average overhead of $51\times$ over native SGX execution for our custom benchmarks and an overhead of $16-57\times$ while executing OpenSSL [19].

II. BACKGROUND

A. Intel SGX

Intel SGX [20] is a new set of x86 instructions which were introduced with the Intel Skylake architecture. SGX allows user-level programs to create a protected memory region called an *enclave* which is inaccessible from other user-level programs as well as privileged components such as BIOS, OS, hypervisor, etc. At boot-time, the processor reserves contiguous physical memory pages, called the Enclave Page Cache (EPC). The CPU explicitly revokes access to EPC pages outside an enclave. Each enclave process is provided its own virtual address space which is divided into trusted and untrusted parts. The trusted part is allocated pages from the EPC to provide memory integrity and confidentiality. The page tables that deal with translation of virtual to physical address for EPC pages are maintained by untrusted system components.

B. SGX Side-Channel Attacks

The three most prominent categories of side-channel attacks against Intel SGX are summarized as follows.

Page Table Attacks. As with regular non-enclave processes, the untrusted OS handles page tables for the EPC pages to flexibly provision EPC resources. Previous works [10, 11] have

shown that a privileged attacker can exploit page faults and page table walks in order to gain page-level granular insight into the execution of an enclave process. Since the OS handles the page tables, it can invalidate access onto all EPC pages which will result in page faults, thereby capturing trace of all page accesses performed by the enclave. Similarly, the attacker can monitor the *access/dirty* bit present within the page table to find out which page was last accessed without invoking a page fault.

Cache Attacks. Caches are designed to reduce the access latency of code and data by exploiting temporal and spatial locality of an application's execution. The caches are divided into a number of cache-sets, which are further divided into fixed-size cache-lines (64 B). Recent reports [12–14] have shown that the SGX enclave is insecure against the Prime+Probe [21] attack. As part of this attack, the attacker runs an attack application which monitors the cache usage of a victim application, performing some security critical operations. During the *Prime* phase, the attacker fills one or more cache sets with his/her own data and during the *Probe* phase, he/she tries to access the data. If the victim has accessed any of these cache sets, it must have evicted some of the cache lines of the attacker, and subsequent access by the attacker will take longer time than if the lines had not been evicted. Therefore, an attacker, with prior knowledge of the victim application, can infer what operation took place (assuming different operations will access different cache sets).

Branch Prediction Attacks. Last Branch Record (LBR) saves the history of the recently *taken* branches which can be referenced by developers for further optimization The LBR stores information including source/target address of a branch, and a flag whether the branch is taken or not, etc. SGX disables direct reporting of the LBR information outside the enclave. However, recent reports [15] have shown how it can be indirectly inferred from outside the enclave. To perform this attack, the attacker leverages prior information on the source and destinations of the branches in a target program. Next, the attacker writes a *shadow* code for a set of branches within the program. The attacker executes both victim and shadow code in parallel. Finally, the attacker monitors the shadow code for mis-predictions (penalized by extra CPU cycles), to figure out which branch was taken by the enclave.

C. ORAM

ORAM [9] is a well-known cryptographic technique which provides secure access to an encrypted memory region located in a remote and untrusted server. ORAM achieves secure memory access by (a) accessing multiple memory locations instead of a single memory location and (b) re-shuffling and re-encrypting the extracted memory regions with a random seed. Path ORAM [22] is an improved variant of ORAM which uses a binary tree-like formation to store the encrypted memory on the server. Each node within the tree is composed of K blocks, where K is a constant defined during initialization. An ORAM tree contains both real blocks, i.e., with actual client data, and dummy blocks, i.e., with dummy data, meant to fool an attacker. The number of real blocks within a tree of L leafs can be at most L in order to provide the security guarantees of ORAM. The tree is stored within the untrusted storage in an encrypted format.

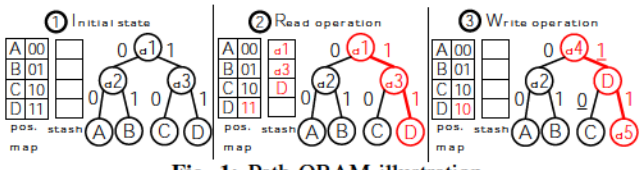

Fig. 1: Path ORAM illustration

Using Path ORAM, the client runs an ORAM controller within a small, completely trusted memory region. There are two key data structures for Path ORAM, i.e., the Position map and the Stash. The position map can be a simple integer array, which links the real block to its corresponding leaf-index within the ORAM tree. Whenever the client needs to access a block from the ORAM tree, the ORAM controller finds the corresponding leaf from the position map and extracts the path from the root to the leaf. The extracted blocks are stored within the stash memory.

Figure 1 illustrates the Path ORAM algorithm. In this figure, the client attempts to access the block D from the untrusted storage containing the ORAM tree (①). First, the client looks-up the leaf index corresponding to block D, which is 11 in our example(②). Then, the client extracts the complete path from the root of the tree to the leaf (i.e., $d1, d3, D$) and saves it in the stash as shown. The dummy blocks (i.e., $d1, d3$) are discarded at this point to keep the stash size small. After accessing the block D, the client randomizes its position, i.e., initial leaf was 11 and final leaf is 10, and re-encrypts the block with a random seed (③). The client then tries to write-back to the tree from the old leaf (11) back to the root. To ensure consistency, the client only writes back a real block on a certain node, iff, that node is the new leaf, i.e., 10, or that node is in the path to the new leaf. If the client does not have a real block to put into the node, it generates dummy data, encrypts it (using random nonce) and writes it to that node. For example, in the figure, $(d4, d5)$ corresponds to the generated dummy data.

III. Threat Model

We assume a scenario where a user runs an SGX enclave program with some security-sensitive program. The enclave program, OBFUSCURO's runtime and compiler, and the CPU are the only trusted components, and all other software and hardware components (including operating systems, hypervisors, memory hardware units, etc.) are untrusted. *The user's goal is to ensure that the program's logic is not leaked to any attacker observing the enclave's execution.* Therefore, the program executable is securely provided to the remote SGX enclave through an encrypted channel (e.g., Diffie-Hellman [23] between enclaves). We assume that the enclave is already provisioned with all prerequisite memory and/or files that it would require to correctly execute before it starts executing. Therefore, we can safely assume that the enclave does not perform a synchronous exit (e.g., for system call) after the start of its execution till termination. *The attacker's goal is to obtain the underlying algorithm or program logic.* To achieve this, the attacker can probe[2] the enclave using any input of his/her choice and get the correct output. Furthermore, the attacker can observe the program's access patterns through a combination of

[2]This assumption can be easily relaxed to ensure input/output confidentiality as we describe in §IX

bus snooping attacks and software side-channel attacks using page tables, caches, and branch prediction units. The attacker can also measure the program's execution time and use that to leak some information.

As far as access patterns are concerned, we assume a worst-case attack scenario: a powerful attacker who learns perfect execution traces at their finest resolution (i.e., 64 B from a combination of page table, cache and bus-snooping, and exact branch targets from branch prediction attacks) of both physical and virtual memory addresses that an enclave program accesses. More formally, let Φ be an SGX enclave program, and its runtime memory access trace $\Phi_k(I)$ $(0 \leq k \leq n)$ denotes a sequence of stripped code and/or data addresses (i.e., stripped addresses depending on the attacking method's granularity) while running an input I. $\Phi_0(I)$ denotes the first address that the program accesses (i.e., an instruction at the program's entry point) and $\Phi_n(I)$ denotes the last address that the program accesses (only if the program terminates on the input I). Furthermore, the attacker can learn some information about the program by monitoring timing channels. The attacker can infer the entire execution time T of the program on his/her provided inputs to leak some information. Given these memory and timing traces, attacker tries to learn the security sensitive information (e.g., the algorithm or some part of it) of the program.

We do not consider software vulnerabilities in an enclave program (i.e., memory corruption vulnerabilities or semantic/logical vulnerabilities) or physical attacks (power-based, electromagnetic etc.) and security solutions [24, 25] to these issues are orthogonal to OBFUSCURO. Furthermore, we consider Spectre [26] and Meltdown [27] attacks out of scope as well. Traditional program obfuscation assumes that the program cannot directly disclose the memory contents of the application which is what these attacks do. Also their patch [28] has already been provided by Intel and can be rigorously checked through the CPUSVN number provided during SGX remote attestation.

IV. CHALLENGES

As mentioned before, the goal of OBFUSCURO is to achieve a strong notion of security — program obfuscation (also referred to as virtual black box (VBB) obfuscation) on market-available commodity trusted hardware, Intel SGX. Unlike supporting program obfuscation on special-purpose hardware, such as HOP [8], there are numerous challenges involved in supporting the same on Intel SGX. These challenges can be attributed to the unprivileged execution supported by SGX enclaves, which either creates new side-channels or amplifies existing side-channels. In particular, these challenges include — (a) how to enforce secure ORAM-based program execution in SGX? and (b) how to secure the ORAM controller in SGX? Unlike special-purpose hardware, SGX enclaves cannot control the page tables, caches and/or the branch-predictor, which can be abused by an attacker to infer significant information from *naive* ORAM-based execution. Also, while special-purpose hardware supports a large trusted on-chip memory which holds the ORAM controller as well as the target program's code, SGX enclaves only provide a very small trusted memory region (i.e., CPU registers) due to side-channels.

A. Comparison with Existing Schemes

In this subsection, we provide a comparison of OBFUSCURO with all existing schemes tailored to provide *oblivious* and/or *obfuscated* execution. For the ensuing discussion, it is imperative that we make a clear distinction between *side-channel obliviousness* (and its weaker version, *memory trace obliviousness*) and *program obfuscation*. In particular, side-channel obliviousness assumes that the program is known to the attacker but the input (securely provided to the program) is sensitive and therefore has to be protected. Program obfuscation assumes that the program is unknown and is itself sensitive whereas input and output pairs can be known to the attacker. It is also worth mentioning that program obfuscation can also be extended to protect the input and output pairs to the program (through employing encryption/decryption of input and output pairs) but it is not its primary goal. Figure 2 provides a comparison of all existing work with OBFUSCURO.

First, we compare the existing side-channel oblivious systems with OBFUSCURO. In general, these systems are based on custom hardware [29, 31], software-level [18, 32, 33] or hybrid [30] defenses. The most notable example of a side-channel oblivious system is Raccoon [18] which can protect the input to a known program against all access pattern leakage (page table, cache, bus-snooping and branch-prediction) on commodity hardware. However, all of these schemes do not fulfill the requirements of traditional program obfuscation and are only concerned with protecting the input provided to the program. On the other hand, program obfuscation protects the identity of the program itself, and can also be used to protect the input provided to the program.

The closest existing work is HOP [8], which is the only system apart from OBFUSCURO, which guarantees virtual black box obfuscation to a program. However, HOP is based on special-purpose hardware and further utilizes an on-chip trusted storage for storing and executing the code segments of the program and the ORAM controller. Thanks to the special-purpose hardware, HOP remains unconcerned with protecting its ORAM controller and the program against sophisticated cache and branch-prediction attacks. Conversely, since OBFUSCURO supports obfuscated execution on commodity hardware, its design revolves around the limitations of the hardware and therefore has to deal with the cache and the branch-predictor, to provide the same theoretical guarantees of program obfuscation.

B. Achieving Obfuscation on Commodity Hardware

In this subsection, we attempt to elaborate on the design choices taken by OBFUSCURO in order to achieve the goals set out by program obfuscation. Just to reiterate, to support program obfuscation, OBFUSCURO has to answer the following questions — (a) how to execute a target program's code without leaking memory traces?; (b) how to provide secure access to its data regions (e.g., stack, heap etc.) without leaking memory traces?; and (c) how to ensure that the program leaks no timing information?

The answer to (a) and (b) lies in the design of fixed scratchpad regions for code execution and data access. In fact, simply doing so is enough for specialized hardware (such as the one used by HOP) but not for commodity hardware,

Scheme	Architecture	Protection Scope				Secure ORAM	Program Obfuscation
		Bus Snooping	Cache Attacks	Branch Prediction	Page-level Attacks		
Raccoon [18]	commodity hardware	✓	✓	✓	✓	✓	✗
Phantom [29]	special purpose hardware	✓	✗	✗	✗	✗	✗
GhostRider [30]	special purpose hardware	✓	✗	✗	✓	✗	✗
HOP [8]	special purpose hardware	✓	✗	✗	✓	✗	✓
OBFUSCURO (this paper)	commodity hardware	✓	✓	✓	✓	✓	✓

Fig. 2: An overview of the differences in OBFUSCURO and existing oblivious execution schemes.

since OBFUSCURO risks leaking information within these regions through page table, cache and branch-prediction attacks. OBFUSCURO ensures that the scratchpad regions are a single cache-line (i.e., 64 B) in size to prevent page table and cache attacks. To further secure the code scratchpad against branch-prediction, OBFUSCURO ensures that all branches to/from the scratchpad are at fixed locations. Although the above design nullifies memory-based side-channels, it raises two important questions — (i) how to support code execution and data access at the granularity of cache-line?; and (ii) how to securely fetch these blocks onto the scratchpads?

In order to support cache-line-granular code execution, OBFUSCURO breaks the target program's code into 64 B basic blocks, normalizes branch instructions within each block and instruments each code access instruction. Furthermore, OBFUSCURO also breaks the data region into blocks of 64 B and instruments each data access to ensure correctness of program execution. Lastly, in order to securely fetch code and data blocks onto the scratchpad regions, OBFUSCURO utilizes ORAM to hide access patterns from a privileged attacker. As shown by previous work [16, 17], the ORAM controller has to be further provisioned to avoid leaking information in SGX enclaves. OBFUSCURO supports both the traditional scheme for securing ORAM whilst also providing an alternative and efficient approach.

Finally, to counter the threat of timing channels and consequently answer (c), OBFUSCURO normalizes the execution time of the target programs by extending the program's execution using dummy (but indistinguishable) code blocks. OBFUSCURO automatically provisions the program with these code blocks such that the program runs indefinitely. OBFUSCURO directs the enclave to stop executing after the execution of a fixed number N of code blocks. As we show in §VII-B, each code block execution takes similar time, resulting in execution-time-normalization for the program.

V. DESIGN

A. Overview

OBFUSCURO is a software framework enabling obfuscated execution for SGX enclave programs. The key idea behind OBFUSCURO is to enable cache-line-granular code execution and data access, secured through the use of ORAM operations, thereby exhibiting memory traces oblivious to program execution (illustrated in Figure 3). The core design features of OBFUSCURO can be summarized as follows.

- **Secure ORAM Scheme**. OBFUSCURO implements its ORAM controller using data oblivious algorithms, in order to protect it from side-channel attacks (§V-B). Also, OBFUSCURO implements a *register-based* stash which improves on the existing side-channel resilient ORAM implementations [16, 17].
- **Repurposing Native Programs**. OBFUSCURO transforms native programs (§V-C) through memory layout transformation and virtual address translation in order to bridge the semantic gap between native program execution and ORAM-based operations.
- **Code Execution Model**. OBFUSCURO ensures that the code execution (of a target program) is exclusively performed within a fixed location, C-Pad (§V-D). All instructions are loaded onto the scratchpad using ORAM operations and executed from the start to the end of the scratchpad (❶∼❸). Furthermore, the C-Pad is designed with SGX-aware protections unlike previous work [8, 30].
- **Data Access Model**. OBFUSCURO ensures that all data access is performed at a data scratchpad, D-Pad, which is a fixed memory location updated using ORAM operations (§V-E). The target program's read and write operations are performed at the same memory location regardless of execution context (①∼⑤). OBFUSCURO also ensures that the data access is always performed once per C-Pad, normalizing the number of data accesses patterns.
- **Start-to-End Obfuscation**. OBFUSCURO ensures that the target program continues executing till a certain predefined time to mitigate timing-based channels, irrespective of the program logic (§V-F). OBFUSCURO achieves this by instrumenting the target application to introduce dummy memory blocks, after the termination of the intended logic.

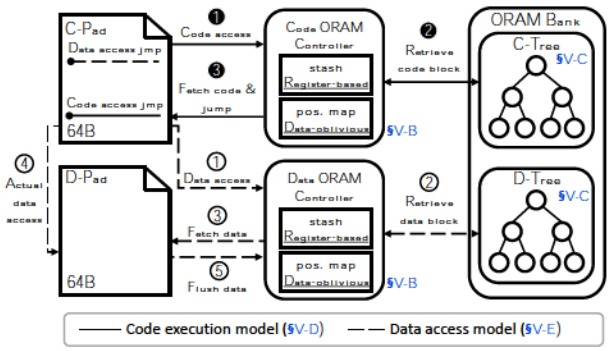

Fig. 3: OBFUSCURO's system-level overview.

Workflow. The input to OBFUSCURO is the source code of a target enclave application. Using the input, OBFUSCURO produces an instrumented executable, fully loaded with a runtime library (containing the ORAM controller). During

initialization, the runtime library *populates* the code and data blocks into different ORAM trees. Afterwards, the ORAM controller extracts the first code-block to be executed, loads it onto the code scratchpad, and ensures execution starts from the beginning of code scratchpad. When the code block performs a branch instruction, the branch instruction is replaced with new jump instruction to the ORAM controller for codes. Then, the ORAM controller loads the required code block onto the code scratchpad using ORAM operations, and jumps back to the beginning of the code scratchpad (§V-D). While accessing data (i.e., global/heap/stack objects), the access instruction is replaced with new jump to the ORAM controller for data. The ORAM controller for data always loads the corresponding data block onto the data scratchpad using ORAM operations, and returns the appropriate address (i.e., base address of data scratchpad + access offset) (§V-E). Finally, OBFUSCURO ensures that the program keeps executing till a certain time period has elapsed before returning an output to the user thereby ensuring complete start-to-end obfuscation (§V-F).

B. Secure ORAM Scheme

In this subsection, we explain how OBFUSCURO designs a secure ORAM scheme to ensure oblivious program execution. Firstly, OBFUSCURO places both the ORAM controller and trees within an SGX enclave. Secondly, in response to side-channel threats against SGX enclaves, OBFUSCURO secures working mechanisms of its ORAM controller, i.e., ensuring that each operation is branch-free (to mitigate the risk of branch-prediction) and data-independent (to mitigate the risk of page table and cache attacks). In this regard, OBFUSCURO constructs two stash designs: CMOV-based and register-based stash for the ORAM controller (§V-B1). Furthermore, OBFUSCURO employs a data-oblivious population scheme to securely populate the ORAM trees (§V-B2).

1) ORAM Controller: In the following, we describe how OBFUSCURO secures the two main data structures of the ORAM controller, i.e., position map and stash, against access-pattern leakage. By securing access onto these data structures, OBFUSCURO also ensures that its code is devoid of conditional branches (i.e., secure against branch-prediction attacks).

Oblivious Position Map. The position map contains sensitive information regarding ORAM blocks, i.e., mapping from block-id to the leaf in ORAM tree. An attacker can leak sensitive information about program execution by observing the access patterns onto the position map. OBFUSCURO employs data oblivious access mechanism to prevent information leakage from the position map. The key security primitive of this mechanism is in leveraging `cmov` instruction in x86 to stream through the entire data structures. Similar to Raccoon [18], we devise a wrapper function for the `cmov` instruction to add additional bogus memory access. Depending on the flag value provided to the wrapper function of the `cmov` instruction, the function performs either the actual memory write (if the flag is true) or a bogus memory access without writing (if the flag is false).

Next, we describe how OBFUSCURO secures access onto the stash. Naively accessing the stash would leave memory traces that can be used to distinguish between real and dummy blocks in the extracted ORAM tree path. OBFUSCURO can utilize two

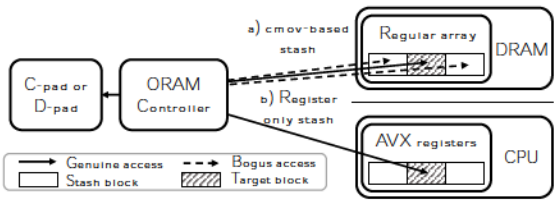

Fig. 4: Register-based stash versus CMOV-based stash. CMOV-based stash has to access an entire array placed in DRAM whereas register-based stash can directly retrieve an item from CPU's AVX registers.

```
void retrieve_from_stash_cmov(void* cpad, int required_blk) {
  bool flag = false;

  for (int i = 0; i < NUM_STASH_BLOCKS; i++) {
    // Check the validity of the condition, i.e.,
    // is this the block to retrieve from the stash
    flag = ((stash[i].blocknum == required_blk));

    // Based on the flag, either perform a real or a dummy copy
    x86_cmov(cpad, stash[i].memblk, flag);
  }
}
```

(a) CMOV-based stash

```
; %rsi points to the base address of ORAM tree block.
movaps (%rsi), %xmm0
vinserti128 $0x0, %xmm0, %ymm5, %ymm5
add $16, %rsi
movaps (%rsi), %xmm0
vinserti128 $0x1, %xmm0, %ymm5, %ymm5
add $16, %rsi
movaps (%rsi), %xmm0
vinserti128 $0x0, %xmm0, %ymm6, %ymm6
add $16, %rsi
movaps (%rsi), %xmm0
vinserti128 $0x1, %xmm0, %ymm6, %ymm6
```

(b) Register-based stash

Fig. 5: Implementation snippets of OBFUSCURO's stash access: (a) OBFUSCURO obliviously retrieves a block from the stash using CMOV; and (b) OBFUSCURO leverages YMM registers to obliviously access stash indices. As can be observed, there are no conditional branches and/or data-dependent access in both cases.

different stash designs, CMOV-based stash and a novel *register-based* stash. While both completely secure stash accesses, it imposes different performance characteristics depending on the underlying hardware architecture.

CMOV-based Stash. OBFUSCURO can use data-oblivious access (using CMOV) to stream through the complete stash memory region (Figure 4-a), similar to previous schemes [16, 17]. As a result, the CMOV-supported access guarantees that the attacker learns nothing from the leaked access patterns as the attacker observes accesses onto all stash indices. One caveat of this approach is that the stash is a large memory region, i.e., >= $Blog_2N$ bytes; where B is the block-size in bytes and log_2N is the size of the ORAM tree containing N nodes. Therefore, using CMOV within the stash can result in performance overhead as noted by previous works and reported in §VIII-1. Figure 5a shows a code snippet illustrating how the CMOV-based stash functions.

Register-based Stash. OBFUSCURO also designs a novel register-based stash, which leverages Advanced Vector Extensions (AVX) instruction set along with the XMM and YMM registers. We collectively refer to these registers as

AVX registers. The key idea is to reserve these registers for ORAM stash only and restrict the program and associated libraries from using them. An operation performed on any CPU register does not imprint traces on memory-related units (cache, TLB/MMU, DRAM etc.) and is therefore oblivious to even privileged attackers such as the OS (Figure 4-b). Therefore, OBFUSCURO copies each tree block onto a set of AVX registers and performs all required operations on these registers. This limits the involvement of `CMOV` and therefore provides a performance improvement of $30-40\%$ as compared to the `CMOV`-based stash as shown in §VIII-1. Figure 5b shows an example of where the memory located at `rsi` is moved in chunks of 32-bytes into `ymm5` and `ymm6`.

However, there are two things to consider while opting for the register-based stash over the `CMOV`-based stash. Firstly, the register-based stash limits the involvement of AVX registers for other important operations such as AES-NI instruction set and if the enclave program requires these operations, it would be better suited to use the `CMOV`-based stash. Secondly, current desktop hardware only supports AVX2 [34] which provides 16 YMM registers of 32 B memory each, totaling to 512 B of memory for the stash. This size is enough for small ORAM tree size (e.g., 4-8KB) but is insufficient for larger tree sizes. However, the AVX-512 [35] instruction set architecture introduces larger AVX registers (ZMM registers), currently present on high-end hardware [36, 37]. The ZMM registers are 32 registers in total, with each being 512-bit wide and can support a total stash size of 2-kilobytes which increases our tree size that can be supported from 8KB to 256MB.

Workflow. Based on the above building blocks, we now illustrate how OBFUSCURO performs a secure ORAM access. First, OBFUSCURO uses `CMOV` to scan through the whole position map to find the required ORAM block. Then, OBFUSCURO sequentially copies the tree blocks to either memory (if `CMOV`-based stash is used) or the registers (if the register-based stash is used). Afterwards, OBFUSCURO performs an oblivious retrieval of the required block from the stash. In the case of `CMOV`-based stash, it performs a sequential `CMOV` access on each individual stash index and in the case of register-based stash, it performs an inline assembly move operation to move it from the register to the memory. After performing the relevant tasks on the ORAM block, we rewrite the block back using similar approach as mentioned above.

2) ORAM Bank: OBFUSCURO places the ORAM bank, comprising of the ORAM trees, within the enclave memory. OBFUSCURO performs secure ORAM tree population to mitigate side-channel leakage.

Allocation. The ORAM trees are allocated as global arrays within the enclave program's memory space (i.e., within the EPC). OBFUSCURO can avoid encrypting ORAM trees, which is an important step in the ORAM protocol, because the *Memory Encryption Engine* (MEE) in SGX [38] implicitly performs the encryption. There are two things to note here: (a) the allocation step does not leak any important information to the attacker apart from the location of the ORAM tree (which is public information in the ORAM attack model) and (b) the size of the code and data trees should be carefully considered prior to allocation since as per Path ORAM's design, the size of the trees cannot be dynamically adjusted.

Population. As per Path ORAM's requirement, the population of each block into the ORAM tree should be performed as a regular ORAM access. To further illustrate, the population of code and data blocks in C-Tree and D-Tree respectively, is carried out as follows: (a) OBFUSCURO picks a block which is to be added to the ORAM tree. (b) OBFUSCURO determines a random position to store the block within the ORAM tree. The random position is determined using the `RDRAND` hardware instruction, which only involves the trusted CPU. (c) OBFUSCURO performs an ORAM access onto the path that corresponds to the selected position. At first glance, this might leak some information to the attacker. However, since this is an ORAM access, the final destination of the block will be randomized within the path once more which ensures strong secrecy. (d) OBFUSCURO repeats the above steps until all real blocks are populated to the ORAM tree.

C. Repurposing Native Programs

In order to bridge the semantic gap between native and oblivious execution, OBFUSCURO transforms the target program's memory layout into an ORAM-compatible memory layout, provides virtual address translation to support dynamic memory relocation, and introduces scratchpad regions for code execution and data access.

Memory Layout Transformation. OBFUSCURO separates the target program into two sections, i.e., code and data, and allocates a dedicated ORAM tree for each section, namely C-Tree for code and D-Tree for data. OBFUSCURO can estimate the size of the C-Tree since the program's code size remains static. Since the size of dynamically allocated data (e.g., heap and stack) cannot be precisely estimated, OBFUSCURO sets a maximum limit on the size of the D-Tree. This is not a limitation since SGX programs themselves are initialized with a user-provided stack and heap size. Code blocks are prepared during the compilation phase, where the code is divided into blocks of the same size and filled with instrumented instructions by OBFUSCURO (more details in §V-D). During program initialization, OBFUSCURO populates both the code blocks and data blocks into the C-Tree and D-Tree respectively. The initialized data objects (i.e., global variables) are filled in their corresponding blocks whereas the blocks corresponding to uninitialized data blocks are zero-initialized.

Virtual Address Translation. All memory accesses in a traditional program are realized through virtual addresses, while ORAM operations deal in blocks of the ORAM tree. To reconcile this, OBFUSCURO performs *on-the-fly* translation of virtual addresses into ORAM block indices. OBFUSCURO linearly maps the virtual address space of a program into ORAM blocks and performs bitwise right-shift to secure translation.

Heap Management. Since SGX enclaves do not have support for dynamic memory allocation, the maximum heap size required for the application has to be decided at compilation time. To handle runtime requests, OBFUSCURO provides a wrapper for the `malloc` and `free` function calls, i.e., `malloc_ob` and `free_ob`, which are responsible for managing the heap memory (alongside the metadata) requested by the enclave program. In particular, `malloc_ob` *obliviously* picks a block from the D-Tree which is already provisioned with blocks to handle heap memory requests during program initialization. The

wrapper function returns the virtual address corresponding to the selected block. Later, when `free_ob` is called, it deallocates the heap memory region, figures out which blocks from the D-Tree are now free and simply tags them as such.

Scratchpad. In traditional ORAM, the program can simply access the extracted block from the stash. However, doing so within the SGX environment will leak a considerable amount of information. To deal with this problem, OBFUSCURO prepares two fixed locations (determined during program initialization) of fixed size (one cache line, i.e., 64 B) to access code and data blocks, called C-Pad and D-Pad respectively. These memory regions are provisioned with SGX-specific defenses (refer to §V-D and §V-E). After OBFUSCURO performs oblivious operation and locates a target block in stash, OBFUSCURO copies the target block in stash to scratchpad. Note that this copy from stash is oblivious as described in §V-B1. Therefore, by normalizing access location and size through scratchpads, OBFUSCURO can successfully hide actual memory location and the attacker can not infer that information. We provide more details as to how this is accomplished in the next two sections.

D. Code Execution Model

OBFUSCURO ensures the following three security properties in its code execution model: C1) Code execution is always performed within the C-Pad[3]; C2) Code access instructions (i.e., branch instructions which impact the control-flow of a program, including call, return, unconditional branch, and conditional branch instructions) are only executed at a fixed location (i.e., the end of the C-Pad); C3) All code access instructions are replaced with an instruction jumping to a runtime function (i.e., `code_oram_controller`), which performs an ORAM operation to fetch the code block required.

The above mentioned security properties of OBFUSCURO protect code execution from access-based side-channel attacks. Since the size of the C-Pad is the same as the minimum granularity of page table and cache-based attacks (i.e., 64 B), C1 prevents these attacks from gaining any meaningful information. C2 and C3 prevent a branch prediction attack, because all the control-flow changes are made from the same location (i.e., the end of C-Pad as specified by C2) to the same destination (i.e., `code_oram_controller` as specified by C3), irrespective of the semantics of the original branch instruction.

To meet the property C1, OBFUSCURO restricts all basic blocks to be at the size of C-Pad (i.e., 64 B) during the compilation phase. Specifically, OBFUSCURO breaks up larger basic blocks into smaller ones equaling the size of the C-Pad. If the size of the basic block is smaller than the C-Pad, OBFUSCURO inserts `nop` instructions to fill the space. To meet the properties C2 and C3, OBFUSCURO replaces all branch instructions with a sequence of equivalent instructions invoking `code_oram_controller`. This invocation is always performed using `jmp` instruction to `code_oram_controller`, which is aligned at the end of the basic block.

For example, Figure 6a shows how OBFUSCURO replaces a unconditional branch instruction. Given the original `jmp`

[3]The C-Pad is a writable and executable region but it can be secured against memory corruption by employing SFI similar to SGX-Shield [24] and/or dynamic page protection to be available in SGXv2.

```
; Before
  jmp jump_target

; After
  mov R15, jump_target          ; Pass jump_target through R15
  jmp code_oram_controller      ; code_oram_controller loads the code
                                ; block to C-Pad and then jumps to the
                                ; beginning of C-Pad.
```

(a) Unconditional branch (code access)

```
; Before
  mov 4(RAX), RBX             ; Store RBX at where (RAX + 4) points to

; After
  lea R15, 4(RAX)             ; Pass the store address through R15
  mov R14, after_fetch        ; Pass the return address through R14
  jmp data_oram_controller    ; data_oram_controller fetches data block
                              ; and returns address of (D-Pad + offset)
                              ; through R15
after_fetch:
  mov (R15), RBX              ; Write a value RBX to (D-Pad + offset)
```

(b) Store (data access)

Fig. 6: Instrumentation on code and data access.

instruction, OBFUSCURO first instruments an instruction storing the virtual address of the jump target in `R15`. Then, OBFUSCURO inserts a `jmp` instruction to the `code_oram_controller`. The code ORAM controller computes the ORAM block index using the virtual address stored in `R15` (as mentioned in §V-C), and retrieves the required code block from the C-Tree through an ORAM access. Afterwards, OBFUSCURO overwrites C-Pad using the obtained code block and resumes execution from the beginning of C-Pad. In this manner, OBFUSCURO translates all types of control flow instructions, including conditional jump, function call, return.

E. Data Access Model

OBFUSCURO ensures the following security properties in the data access model: D1) Data access is always performed within the D-Pad of size 64 B; D2) Data access instructions are only executed once per C-Pad at a fixed location (i.e., the beginning of the C-Pad); and D3) All data access instructions are replaced with an instruction jumping to a runtime function, `data_oram_controller`, which performs an ORAM operation to load the corresponding data block onto the D-Pad. Similar to the code execution model (§V-D), these properties prevent cache and page table attacks. This is because attackers will always observe the same data access patterns onto D-Pad.

One thing to note here is that D2 enforces each code block to perform a single jump to the `data_oram_controller`. This restriction is partly due to the constraint of the 64-byte code block. In particular, OBFUSCURO's data access instructions take 28-bytes and the code access instructions (mentioned in §V-D) take 20-bytes. Since a code block requires at least one code access instruction, i.e., to access the next code block, it leaves room for only a single data access. However, as a result of this, OBFUSCURO ensures that there is a normalized number of data access per code block, which cannot be exploited by an attacker. OBFUSCURO also prevents branch-prediction attacks by placing the data access instruction at a fixed location. If a certain code block does not require a data access, OBFUSCURO performs a dummy data access in order to portray the same memory footprints for each block.

Unlike the code execution model, the data access model allows offset-based access within the D-Pad such that a memory

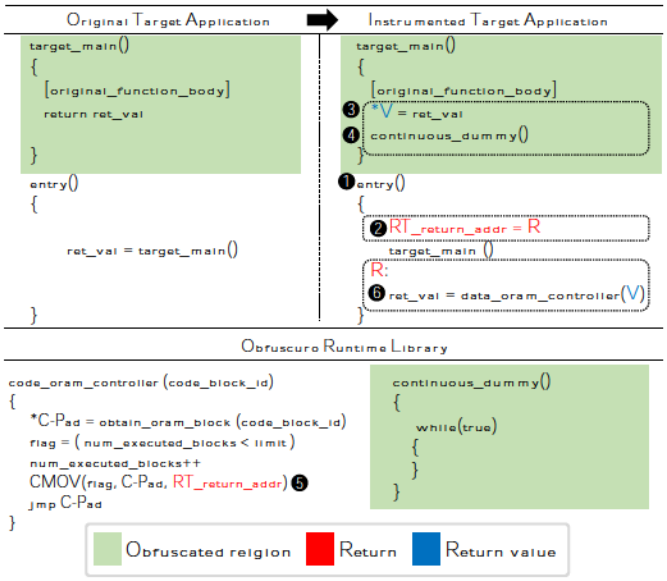

Fig. 7: OBFUSCURO's continuous execution.

access can be directly performed at any location within D-Pad. This offset-based access is secure against memory-based side-channel attacks since the D-Pad is the size of the minimum granularity of attack resolution, i.e., 64 B. In order to reflect changes made by the enclave code on the D-Pad back to the ORAM tree, OBFUSCURO flushes the extracted data block after performing required memory access.

For example, Figure 6b illustrates how OBFUSCURO instruments the store instruction. Similar to the code execution model, OBFUSCURO uses the reserved `R15` register to pass the virtual address (i.e., the memory operand of a store instruction) to the `data_oram_controller`. Then the `data_oram_controller` translates the virtual address into the corresponding ORAM block index, and updates D-Pad after extracting the data block using an ORAM access. Afterwards, the `data_oram_controller` returns the virtual address through `R15`, which points within D-Pad (i.e., `p1 + p2`, where `p1` is the base address of D-Pad and `p2` is the offset within the D-Pad). Therefore, the enclave program correctly performs the store instruction using `R15`, and the data block is later flushed back into the D-Tree.

F. Start-to-End Obfuscation

In the previous subsections, we explain how OBFUSCURO ensures that the target program's code blocks perform a normalized sequence of operations, irrespective of their original logic. However, that is not enough for complete obfuscation. In particular, there is one further distinguishing factor in the program, i.e., execution time of the program. For example, running different programs or just running the same programs with different inputs can result in drastically different execution times, which can be abused by an attacker.

OBFUSCURO handles both of these cases to ensure that, irrespective of program logic, the obfuscated execution always terminates after a fixed amount of time. In order to fix the execution time, OBFUSCURO inserts dummy code blocks within a native program's code ensuring that the program keeps executing even after completing the intended program logic. OBFUSCURO instruments the target application as shown in Figure 7. As shown in the figure, OBFUSCURO injects a dummy function called `continuous_dummy` into the program. The dummy function is meant to execute a `while` loop indefinitely, ensuring that program will not terminate of its own will. As mentioned in §V-D, each code access will go through the `code_oram_controller`. Therefore, OBFUSCURO can stop the program execution after a certain predefined number of code blocks, even if the dummy function never stops executing. However, to do so and provide the required output back, OBFUSCURO needs an address to jump to after reaching the limit on code blocks.

Now, we explain the workflow of the instrumented target program. The application code is defined as `target_main` whereas the enclave officially starts execution from the `entry` function (❶). At the start of the `entry`, OBFUSCURO ensures that the return address `R` is passed to the runtime library by writing `RT_return_addr` (❷). Afterwards, OBFUSCURO starts running the `target_main` function and writes its output to a global memory within the program (❸). It is worth noting that this write will also be achieved through an ORAM access (as per all data access mentioned in §V-E) and is therefore oblivious to the attacker. Then, OBFUSCURO invokes `continuous_dummy` (❹), ensuring that the program continues executing.

As the program executes, it will jump to the `code_oram_controller` on each code access. At this time, OBFUSCURO checks that the predefined limit on the number of code blocks has been reached or not. If the limit has been reached, the program jumps back to `RT_return_addr` instead of jumping to the C-Pad (❺). At this point, we completed the execution of original program logic but have not obtained the output. To get the output, OBFUSCURO calls the `data_oram_controller` to extract the output from the D-Tree (❻). Through the above mentioned steps, OBFUSCURO ensures that there is a *start-to-end* obfuscation of the target program, which always executes the same number of code blocks and thus terminates after a fixed amount of time.

VI. IMPLEMENTATION

We have implemented a prototype of OBFUSCURO based on the LLVM Compiler project 4.0 as well as Intel SGX SDK's enclave loader. OBFUSCURO modified following two components in LLVM: a) LLVM backend to emit 64B of code blocks as well as to instrument code and data access instruction; and b) Compiler runtime library for ORAM controllers. In the LLVM backend, especially the assembly emitter, we arranged a new code emitter to measure the size of instructions in parallel with default emitter. We also utilized built-in machine code builder to redirect the codes and data accesses to the runtime ORAM controllers. The compiler runtime library includes the implementation of data-oblivious ORAM, and interfaces for LLVM backend and applications to employ it. The oblivious stash access is implemented with `vinserti128`, and `vextracti128` AVX register manipulating instructions in the assembly language level. The oblivious position map access is based on the `CMOV` instruction, and we generalized its operation to variable lengths. We also changed the enclave loader of the Intel SGX SDK to make C-Pad using SGX's `EADD` instruction. In total, OBFUSCURO introduces 3,117 LoC in LLVM backend,

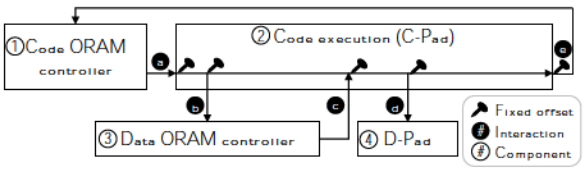

Fig. 8: Data oblivious execution cycle of OBFUSCURO

2,179 LoC in compiler runtime library, and 25 LoC in Intel SGX SDK.

VII. SECURITY ANALYSIS

This subsection provides a security analysis of OBFUSCURO. In general, there are two ways an attacker can steal information from SGX enclaves using side-channels. Firstly, an attacker can abuse observed access-patterns to infer some information about the program and/or its input. Secondly, an attacker can perform timing-based attacks to leak some information. We provide a systematic security analysis of OBFUSCURO against both of these attack avenues.

A. Access Pattern Attacks

As OBFUSCURO is composed of multiple components to realize obfuscated program execution, we start by showing the security properties of the individual components of OBFUSCURO. Then we show how these components interact with each other and show that these interactions are completely oblivious as well. Finally, we present the results of an empirical study showing that OBFUSCURO achieves access pattern obliviousness.

Obliviousness of Individual Components. OBFUSCURO introduces newer components to legacy programs in order to achieve obfuscated execution, as shown in Figure 8. In the figure, we show the four components of OBFUSCURO (labeled as ① ∼ ④). We comment on each component individually in the following.

① ***Code ORAM controller*:** The code ORAM controller takes the virtual address of next required code block as input, and it places the corresponding code block on the C-Pad. An attacker cannot decipher the virtual address because OBFUSCURO performs secure computation based on this address. In particular, the address is first translated to a specific ORAM block using data oblivious right-shift operation (§V-C), which returns the corresponding block number in the ORAM tree. Then, OBFUSCURO finds the corresponding leaf for this block through sequential `CMOV`-based scanning of the position map.

For the stash, OBFUSCURO uses two variants, a `CMOV`-based and a register-based. The `CMOV`-based stash performs `CMOV`-based memory access similar to how OBFUSCURO shields the position map. This includes both (a) while copying the required block from the stash to the C-Pad or D-Pad and (b) while writing back the blocks from the C-Pad or D-Pad to the stash. For the register-based stash, the AVX registers are retrofitted as stash space. Since all operations on the AVX registers are oblivious to the underlying system, we can perform a direct memory access to/from a specific register while ensuring that no information is leaked. Please refer to Figure 9 for detailed operations performed by the code controller.

② ***C-Pad*:** OBFUSCURO ensures that the C-Pad has a fixed location (determined at the program loading) and a fixed size (i.e., 64B), and ensures that all oblivious code execution occurs from this location. Since 64B is the cache-line size (i.e., the finest visible granularity through access pattern-based side-channel attacks), the attacker learns no useful information to infer semantics during the C-Pad execution. In other words, as OBFUSCURO runs the target program, the attacker will keep observing the same memory activity over C-Pad, which is completely independent of the code block being executed.

③ ***Data ORAM controller*:** The data ORAM controller takes the virtual address of data objects as input, and places the corresponding data block to D-Pad. The data controller follows the exact same workflow of the code controller except that it operates on the D-Tree instead of the C-Tree. As previously shown for the code controller, the data controller also does not leak any sensitive information.

④ ***D-Pad*:** The D-Pad is functionally and structurally similar to the C-Pad, except that data access is performed on it and not code execution. Similar to the C-Pad, it has a fixed location and the same size, thereby showing the same memory activity for each data access.

Oblivious Interactions b/w Components. The aforementioned components perform five interactions between them (labeled as ⓐ ∼ ⓔ). We illustrate below how each of these interactions is secure against access pattern-based attacks.

ⓐ ***Jump from Code ORAM controller to C-Pad after fetching code block*:** After obliviously extracting a block from the C-Tree and copying it to C-Pad, the code controller performs a single jump to the start of the C-Pad. This step only reveals that some code block of a target program will now be executed, which entails no semantics behind the code block being executed.

ⓑ ***Jump from C-Pad to Data ORAM controller for fetching data block*:** Each code block (executing within the C-Pad) is strictly enforced to perform a single jump to the data controller, because OBFUSCURO normalizes the number of data access within each code block to be exactly one (refer §V-E). Moreover, this jump is performed at a fixed offset within C-Pad to mitigate the risk of branch prediction attacks. The target address of this jump is also fixed, i.e., the start of the data controller's logic.

ⓒ ***Return from Data ORAM controller to C-Pad*:** There is only a single jump from the data controller to the C-Pad at a fixed offset within the C-Pad, after fetching/updating the required data block on the D-Pad.

ⓓ ***Single D-Pad access*:** There is only a single access to the D-Pad per code block. Since the size of the D-Pad is 64B, this access does not reveal offset information either.

ⓔ ***Jump from C-Pad to Code ORAM controller*:** Finally, OBFUSCURO enforces that there is only one jump from C-Pad to the code controller at a fixed address located towards the end of the C-Pad. The target address of this jump is also fixed at the start of the code controller logic.

Empirical Study. Lastly, we present the results of our empirical study on obfuscated memory traces exhibited by various applications. The results are depicted in Figure 10.

ORAM operations	Sensitive information	OBFUSCURO defense	Observed traces by adversaries
1. Locating corresponding pos.map element	Offset in pos.map	CMOV-scanning read	Sequential read traces on pos.map
2. Extracting requested ORAM path to stash	No sensitive info.	-	Sequential copy traces from requested ORAM path to stash
3-a. Copying ORAM block in stash to scratchpad (CMOV-based)	Offset in stash	CMOV-scanning copy	Sequential copy traces from stash to scratchpad
3-b. Copying ORAM block in stash to scratchpad (Register-based)	Offset in stash	Register operations	No traces since registers are oblivious to memory
4. Updating pos.map with new leaf number	Offset in pos.map	CMOV-scanning write	Sequential write traces on pos.map
5-a. Writing back scratchpad to stash (CMOV-based)	Offset in stash	CMOV-scanning write	Sequential write traces from scratchpad to stash
5-b. Writing back scratchpad to stash (Register-based)	Offset in stash	Register operations	No traces since registers are oblivious to memory
6. Writing back stash to requested ORAM path	No sensitive info.	-	Sequential write traces from stash to requested ORAM path

Fig. 9: Security analysis of secure ORAM implementation used by the code and data controller.

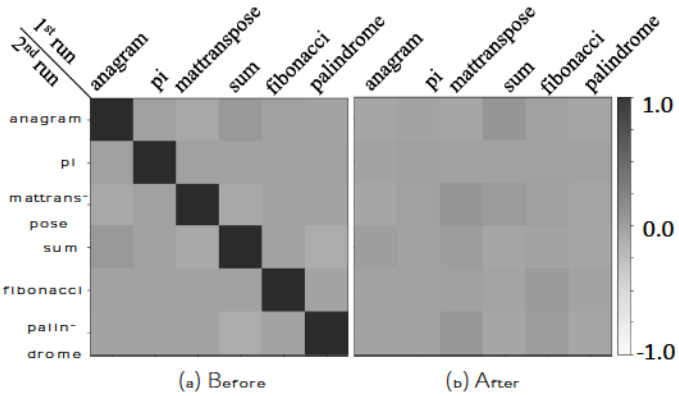

Fig. 10: Confusion matrix for native access patterns vs. obfuscated patterns shown by OBFUSCURO.

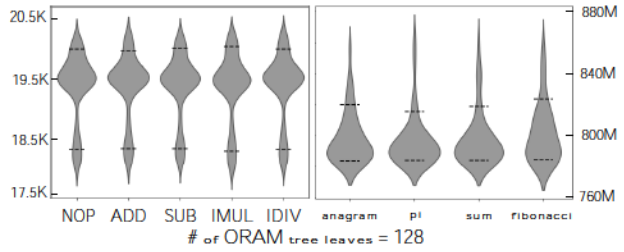

Fig. 11: (a) Distributions of code execution cycles of different types of code blocks (y-axis) with 10%∼90% percentile intervals. (b) Distributions of total execution cycles of various test programs (y-axis) with 10%∼90% percentile intervals.

We choose six target applications for the study, including `anagram`, `pi`, `mattranspose`, `sum`, `fibonacci`, and `palindrome`. These applications were chosen due to the diversity of their computational complexity.

In Figure 10, we attempt to show that there is no correlation between native and obfuscated memory traces of the same program. We measure multiple runs for each aforementioned application, and for each run we accumulate data corresponding a timing sequence to the address accessed by the program. Using the accumulated data, we calculate the Pearson correlation value between the test applications and populate the corresponding cell in the confusion matrix. For example, consider the $(anagram, anagram)$ cell in Figure 10-(a), the Pearson correlation value is very close to 1 because this cell is comparing the memory traces between two runs of the same program. On the other hand, the correlation value in the $(anagram, pi)$ cell is nearly 0 because their access patterns are quite unique to each other.

Figure 10-(b) shows the confusion matrix formed while comparing obfuscated programs (using OBFUSCURO) to their native access patterns. Since OBFUSCURO ensures that all applications proceed in a fixed pattern of execution, the access patterns of these programs are completely different from their counterparts in native execution. Furthermore, all cells in Figure 10-(b) are almost 0 because none of obfuscated programs have any correlation with any of the native programs.

B. Timing-based Attacks

Apart from access pattern attacks, a privileged attacker can also break program obfuscation within Intel SGX by abusing timing channels. In particular, we expect following two ways in which an attacker can abuse timing channels to leak information from OBFUSCURO—(a) observing the time it takes for individual code blocks (in C-Pad) to execute, and (b) observing the total time it takes for an obfuscated program to execute. We individually show the infeasibility of each of these timing channels.

C-Pad Execution Time. Timing differences in executing each code block (i.e., C-Pad) can leak information about the execution semantic of the program. We statistically prove that this side channel is infeasible within OBFUSCURO's execution. The reason for this is that the execution time for the data ORAM access (which is performed exactly once per C-Pad) dominates the entire execution time of the C-Pad, and the time taken to perform the ORAM access is independent to which data block it accesses. We conducted a statistical experiment measuring CPU cycles in executing different classes of code blocks. We constructed five different code blocks, including `NOP`, `ADD`, `SUB`, `IMUL`, `IDIV` code blocks. Each code block initially jumps to the data controller to fetch a data block and the remaining space is filled using one of the instruction type. Furthermore, we impose data dependencies within the instructions to prevent out-of-order execution. We accumulated the execution times for each class over 10,000 repetitions, and the distribution is shown in Figure 11-(a). As illustrated, the 10%∼90% percentile intervals for each type (marked as two broken lines) largely overlap, which is hardly possible for an attacker to distinguish.

Program Execution Time. As mentioned in §V-F, OBFUSCURO ensures that a program continues executing until its number of executed code blocks reaches a fixed user-configured limit. In particular, OBFUSCURO allows the user to define the total number of C-Pad executions a program should perform. If the program's logic terminates before that number is

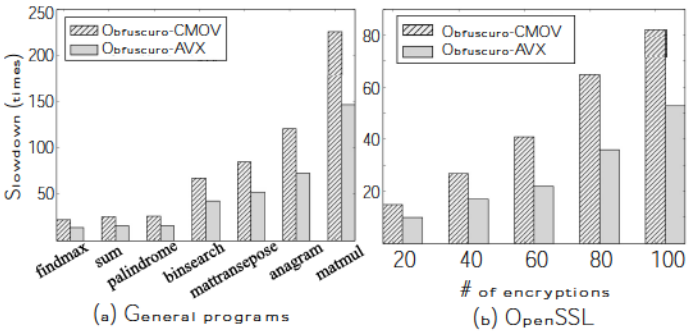

Fig. 12: Performance benchmarks from our test applications. The average performance overhead of OBFUSCURO-CMOV is 83× and for OBFUSCURO-AVX (simulated) is 51×.

reached, OBFUSCURO continues executing dummy code blocks to complete the number of C-Pad executions.

In order to prove that this results in a uniform execution time irrespective of the target program being executed, we performed an experiment on a diverse set of applications as shown in Figure 11-(b). In the experiment, we fixed the total number of C-Pad executions for each of these applications to 30,000 and measured the total execution time. We accumulate 100 executions for each program, and plot the distributions of them. As shown in the figure, the ranges of total execution times for the chosen evaluation set largely overlaps, despite computational diversity of these applications. The reason for this is that each C-Pad execution, as illustrated before, is bounded at very similar execution times irrespective of the underlying CPU instructions. Therefore, it is expected that the program execution time (with same number of C-Pad executions) will also be very similar.

VIII. PERFORMANCE EVALUATION

In this section, we report a detailed performance benchmark through both micro-benchmarking custom applications and macro-benchmarking by running openSSL [19].

Experimental Setup. All our evaluations were performed on Intel(R) Core(TM) i7-6700K CPU @ 3.40GHz (Skylake with 8 MB cache, 8 cache-slices and 16-way set-associativity) with 64 GB RAM (128 MB for EPC). Our system ran Ubuntu 16.04 with Linux 4.4.0.59 64-bit. We performed our experiments using Intel SGX SDK [39] and the Intel SGX drivers [40]. Due to the current unavailability of AVX-512 for SGX-enabled computers, most of our experiments (having large code and data sizes) used `CMOV`-based stash. However, we experimented with AVX2 registers to find the expected benefit of using the *register-based* stash and have accordingly simulated the performance improvement achieved by *register-based* stash on our target applications.

1) Micro-Evaluation: Firstly, we start by providing a detailed performance evaluation result by running several programs with OBFUSCURO. Next, we show the performance improvement achieved by the novel *register-based* stash designed by OBFUSCURO.

Benchmarks. We ported simple benchmarking applications on OBFUSCURO in order to show the feasibility of obfuscated execution using commodity hardware such as Intel SGX. In particular, we ported a diverse set of applications from simple applications like finding the maximum within a given array to complex binary searching.

Data Size (Bytes)	CMOV (cycles)	AVX (cycles)	Improvement
1,024	272M	206M	32%
2,048	521M	388M	34%
4,096	1,044M	741M	41%
8,192	2,050M	1,481M	38%

Fig. 13: Performance improvement achieved by using the AVX2 register extensions as the ORAM stash compared to `CMOV`-based stash.

Figure 12-(a) shows the performance shown by OBFUSCURO while running the test set of applications described above. We also simulate the performance of OBFUSCURO-AVX (the version of OBFUSCURO which uses register-based stash. These simulated results are based on the experiments we performed on AVX2. In general, the performance overhead of OBFUSCURO-CMOV is on average 83× and OBFUSCURO-AVX is 51× The performance overhead of OBFUSCURO is expected since it has to cater to the plethora of side-channels plaguing Intel SGX. In no particular order, the overhead is attributed to: (a) code access control especially dealing with branch-alignment, (b) data access normalization and (c) side-channel-resistant ORAM-based access inside Intel SGX.

Comparison: CMOV-based vs Register-based Stash. We provide a comparison of the `CMOV`-based stash versus the register-based stash. We attempt to answer the question — what is the performance benefit attained by using register-based stash over the CMOV-based stash? One caveat is that all our experiments are based on the AVX2 registers but we expect the performance benefits to be similar while using the AVX-512 registers. Figure 13 attempts to illustrate the performance benefit achieved by AVX extensions over `CMOV` while accessing data of variable size through ORAM. Compared to the `CMOV`-based stash, since the register-based stash performs just a single oblivious access onto the AVX registers, it outperforms the `CMOV`-based stash. The average improvement is around 30-40%.

2) Macro-Evaluation: In order to show how real-world applications perform with OBFUSCURO, we provide a case-study with OpenSSL [19]. Figure 12-(b) shows the result of our evaluations using OpenSSL with OBFUSCURO and without OBFUSCURO. In this experiment, we perform a variable number of consecutive encryptions and compare the results. As the number of encryptions increase, the difference between the performance of OBFUSCURO and native also increases. The reason for this is that OBFUSCURO has to perform a fixed number of ORAM operations which adds significant overhead per-encryption whereas the per-encryption overhead of native execution is very small.

IX. DISCUSSION

Timing Channels. Based on our statistical analysis, OBFUSCURO provides accurate execution-time-normalization (see §VII-B). But, it is hard to conclusively prove that OBFUSCURO would leak no timing information regardless of the underlying application. However, if even that is the

case, we believe that OBFUSCURO can still be used to defeat all timing channels. For example, OBFUSCURO could profile the execution time of each code block of the target program and, if a discrepancy is encountered, carefully modify the code blocks such that they would have identical execution times. This would, in turn, also provide accurate execution-time-normalization. We believe that, if required, this profiling and subsequent modification would not require help from the program developer either. We leave a thorough exploration of timing channels that could affect OBFUSCURO as part of future work.

AVX-512. As shown in §VIII, the register-based stash can provide a performance improvement over CMOV-based stash. But, our experiments were performed on AVX2 instructions due to the current unavailability of AVX-512 for SGX-enabled processors. Intel states that AVX-512 register instructions result in frequency reduction [41] which could potentially slow down the entire application. However, there are two reasons why this might not be an issue — (a) linux-based systems (in particular) allow control of frequency scaling [42] and (b) users have found out that only heavy (e.g., floating point) instructions cause frequency scaling [43]. Especially for the latter, it has been reported that load/store instructions on AVX-512 registers (which OBFUSCURO is concerned with only) provide similar performance as AVX2 registers.

Comparison with Cryptographic Schemes. On par with what OBFUSCURO provides, we discuss following two security properties: 1) computational confidentiality and 2) integrity. Towards these security properties, we focus our discussion on theoretical program obfuscation techniques, which construct a virtual black box (VBB). We note that, unlike OBFUSCURO which performs hardware-assisted secure remote computation, theoretical program obfuscation techniques [44–46] do not rely on specific architectural characteristics and thus are designed to be resistant to memory-based side-channel attacks.

More specifically, two well-known cryptographic primitives, fully-homomorphic encryption (FHE) and garbled circuits are generally used for the program obfuscation, but both of them are limited in terms of either performance and generality. In the case of FHE [47], its performance overhead is in twelve orders of magnitude scale in string search [48] without ensuring integrity. On the other hand, garbled circuits [49] incur a performance overhead of around four orders of magnitude. Moreover, they cannot be used for generic programs (i.e., a loop structure in a program cannot be supported), and the integrity cannot be guaranteed similar to FHE. To ensure integrity, verifiable computing techniques can be adopted but verifiable computing itself imposes huge overheads (i.e., about 10^4 times [50]).

Compared to theoretical solutions, OBFUSCURO efficiently achieves confidentiality and integrity, leveraging memory protection and remote attestation mechanisms of SGX. From the performance perspective, OBFUSCURO is a more practical solution since it imposes two orders of magnitude performance overhead, as opposed to twelve and four orders in the case of FHE and circuit representation, respectively. OBFUSCURO also supports generic programs since it retains the form of the host-architecture instruction.

Protecting Input/Output. Traditional program obfuscation assumes that the attacker has an oracle-like access to the obfuscated program. Therefore, the attacker can provide input and get the corresponding correct output. However, OBFUSCURO can be further leveraged to guarantee that an attacker does not figure out the input/output either. For the input, since it is not controlled by OBFUSCURO, we assume that the user of the enclave will provide us a fixed-length encrypted memory buffer to extract the input from. OBFUSCURO will execute for a fixed time T based on the input and extract a fixed-size output from the D-Tree at the end of T. Then, OBFUSCURO will encrypt this data and send it back to the user.

Potential Applications. There are various potential applications for OBFUSCURO ranging from protection of a intellectual property to securely patching vulnerabilities. Firstly, OBFUSCURO can ensure that machine learning services requiring huge computing resources can safely outsource their computational load to cloud servers. For example, companies like 23andMe [3] want to outsource genomic analysis but also want to stay ahead of the competition by preventing the theft of their algorithm. Secondly, developers can securely patch vulnerabilities without disclosing the vulnerabilities through the patches, rendering their exploitation highly unlikely.

Generic Side-channel Defense. OBFUSCURO can be utilized as a general-purpose side-channel defense, whose main objective is to protect the input of a known program from attackers. The attackers usually exploit unique memory access patterns leaked from side channels consisted of caches, page fault, and branch predictor [10, 12–15]. Since OBFUSCURO is specifically designed to protect all these channels, OBFUSCURO can protect the target program. Furthermore, we could utilize OBFUSCURO to constrain its protection scope to a small, sensitive portion of the code, which would result in performance gains as well.

Other Use-cases. Our current design for an oblivious execution framework is SGX-specific. However, we believe its design characteristics and optimization techniques are general, which can be applied to other trusted platforms such as AEGIS [51], Ascend [52], XOM [53], Bastion [54], Sanctum [55]. For example, our register-based stash (§V-B1) can be considered as a generic optimization for ORAM, if the underlying trust architecture shares any of memory-related subsystem such as cache, TLB, MMU, and DRAM.

X. RELATED WORK

SGX-based Systems. Haven [56], Graphene [57, 58] and Panoply [59] provide LibOS for SGX, which enable easier application porting and prevent Iago attacks [60]. OpenSGX [61] provides an open research framework for running SGX applications. VC3 [62] provides oblivious data analytical algorithms such as MapReduce [63]. SGX-Shield [24] performs fine-grained ASLR within SGX environments. Some of OBFUSCURO's design schemes, particularly how OBFUSCURO breaks a program into smaller ORAM-compatible blocks, have been inspired by SGX-shield. Ryoan [64] provides a secure framework to port Native Client (NaCl) [65] in Intel SGX. SCONE [66] provides performance optimizations and ports containers within SGX. Eleos [67] provides a framework to use non-enclave space to improve enclave performance. Glamdring [68] provides automatic partitioning within enclave programs. Other works [69–71] consider how to efficiently deliver cryptographic primitives such as multi-party computation and functional encryption

using Intel SGX. These systems do not consider side-channel issues within SGX and can be used together with OBFUSCURO.

Attacks on SGX. SGX is vulnerable to both page fault [10] and page table [11] attacks. Recent works [12–14] have shown that cache-based attacks are possible with an SGX enclave. SGX has also been found to be vulnerable against the branch-prediction attack [15, 72]. Wang et. al [73] provide an overview of the attack vectors against SGX and the limitations of current defense solutions.

SGX-compatible Defenses. There have been various defenses [74, 75] proposed against the page table attacks. T-SGX [74] uses Transactional Memory (TSX) to run a program. However, T-SGX is vulnerable to the improved controlled channel attack [11]. Cloak [76] also utilizes TSX as a defense primitive, but it only considers cache side channel attacks. Another work [75], provides a way to prevent page faults from the OS-level attacker by periodically modifying the program's memory access patterns. Ohrimenko et al. [77] show how to re-adapt ML-algorithms to exhibit data-oblivious memory access patterns. For cache-based attacks, the previous solutions [78–80], for non-SGX environments, are not directly applicable since most of them require OS support. Compared to these defenses, OBFUSCURO proposes a generic security framework against all memory-based side-channel attacks. Obliviate [16] and ZeroTrace [17] provide access to files and data structures respectively using secure ORAM implementations. Compared to OBFUSCURO, their scope of protection is limited, i.e., files and data arrays respectively.

Hardware and Software-based Oblivious Systems. Previous work has alluded to the concept of creating oblivious systems based on custom hardware [8, 29, 31], software-level defenses [18, 32] or hybrid [30]. All aforementioned systems use variants of ORAM [9] to achieve oblivious execution. Out of all these works, HOP [8] and Phantom [29] are the most similar. However, both Phantom and HOP use RISC-V processors to implement secure ORAM controllers while OBFUSCURO runs on commodity trusted hardware.

XI. CONCLUSION

This paper presents OBFUSCURO, the first system which provides program obfuscation using commodity trusted hardware. OBFUSCURO systematically protects the SGX enclave against information leakage through all side-channels, thereby neutralizing all memory and timing footprints to create a virtual black box for obfuscated program execution. Our evaluation shows that OBFUSCURO can provide strong obfuscation guarantees within Intel SGX while performing much faster than existing cryptographic schemes and being more deployment-friendly than existing system-based solutions.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments on this work. This work is partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2018R1C1B5086364), by National Science Foundation (NSF) under grant No. 1750809, and Samsung Research Funding & Incubation Center (SRFC-IT1701-05).

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Annual International Cryptology Conference*. Springer, 2001.

[2] S. Hada, "Zero-knowledge and code obfuscation," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2000.

[3] 23andme, "23andme," 2018. [Online]. Available: https://www.23andme.com

[4] N. Bitansky, R. Canetti, S. Goldwasser, S. Halevi, Y. T. Kalai, and G. N. Rothblum, "Program obfuscation with leaky hardware," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2011.

[5] K.-M. Chung, J. Katz, and H.-S. Zhou, "Functional encryption from (small) hardware tokens," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2013.

[6] N. Döttling, T. Mie, J. Müller-Quade, and T. Nilges, "Basing obfuscation on simple tamper-proof hardware assumptions." *IACR Cryptology ePrint Archive*, 2011.

[7] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia, "Founding cryptography on tamper-proof hardware tokens," in *Theory of Cryptography Conference*. Springer, 2010.

[8] 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.

[9] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, 1996.

[10] 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.

[11] J. Van 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] 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.

[13] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[14] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx." in *10th European Workshop on System Security (EUROSEC)*, 2017.

[15] 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.

[16] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious file system for intel sgx," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[17] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[18] 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.

[19] O. S. Foundation, "OpenSSL: Cryptography and SSL/TLS Toolkit," https://www.openssl.org/, 2017.

[20] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

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

[22] 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.

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

[24] 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.

[25] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "Sgxbounds: Memory safety for shielded execution," in *Proceedings of the Twelfth ACM European Conference on Computer Systems (EUROSYS)*, 2017.

[26] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (Security)*, 2018.

[28] Intel, "Intel analysis of speculative execution side channels," 2018. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf

[29] 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 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[30] 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 Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2015.

[31] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing (STC)*, 2012.

[32] C. Liu, M. Hicks, and E. Shi, "Memory trace oblivious program execution," in *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, 2013.

[33] R. Sinha, S. Rajamani, and S. A. Seshia, "A compiler and verifier for page access oblivious computation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. ACM, 2017.

[34] Intel, "Overview: Intrinsics for intel (r) advanced vector extensions 2 (intel(r) avx2) instructions," 2014. [Online]. Available: https://software.intel.com/en-us/node/523876

[35] J. Reinders, "Avx-512 instructions," *Intel Corporation*, 2013.

[36] Intel, "Intel (r) core (tm) i9-7980xe extreme edition processor," 2017. [Online]. Available: https://ark.intel.com/products/126699/Intel-Core-i9-7980XE-Extreme-Edition-Processor-24_75M-Cache-up-to-4_20-GHz

[37] ——, "Intel (r) core (tm) i9-7970x processor," 2017. [Online]. Available: https://ark.intel.com/products/126697/Intel-Core-i9-7960X-X-series-Processor-22M-Cache-up-to-4_20-GHz

[38] S. Gueron, "A memory encryption engine suitable for general purpose processors." 2016.

[39] 01org, "Intel(r) software guard extensions for linux* os (source code)," 2016. [Online]. Available: https://github.com/01org/linux-sgx

[40] Intel, "Intel sgx linux* driver," 2017. [Online]. Available: https://github.com/01org/linux-sgx-driver

[41] ——, "Optimizing Performance with Intel(r) Advanced Vector Extensions," https://computing.llnl.gov/tutorials/linux_clusters/intelAVXperformanceWhitePaper.pdf, 2017.

[42] A. Linux, "CPU frequency scaling." https://wiki.archlinux.org/index.php/CPU_frequency_scaling, 2017.

[43] D. Lemire, "By how much does AVX-512 slow down your CPU? A first experiment." https://lemire.me/blog/2018/04/19/by-how-much-does-avx-512-slow-down-your-cpu-a-first-experiment/, 2017.

[44] K.-M. Chung, J. Katz, and H.-S. Zhou, "Functional encryption from (small) hardware tokens," in *Advances in Cryptology - ASIACRYPT*. Berlin, Heidelberg: Springer, 2013.

[45] N. Döttling, T. Mie, J. Müller-quade, and T. Nilges, "Basing obfuscation on simple tamper-proof hardware assumptions." in *IACR cryptology eprint archive*, 2011.

[46] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia, "Founding cryptography on tamper-proof hardware tokens," in *Theory of Cryptography*, D. Micciancio, Ed., 2010, pp. 08–326.

[47] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC)*. New York, NY, USA: ACM, 2009.

[48] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[49] O. Goldreich, "Cryptography and cryptographic protocols," *Distributed Computing*, 2003.

[50] J. Thaler, "Verifiable computing: Between theory and practice," 2017.

[51] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *Information Security Technical Report*, 2005.

[52] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proceedings of the seventh ACM workshop on Scalable trusted computing (STC)*. ACM, 2012.

[53] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, 2000.

[54] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[55] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation." in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[56] 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.

[57] 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.

[58] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, 2008.

[64] 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.

[65] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2009.

[66] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. 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.

[67] 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. 2016.

[68] J. Lind, C. Priebe, D. Muthukumaran, D. OâĂŹKeeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche *et al.*, "Glamdring: Automatic application partitioning for intel sgx," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jun. 2017.

[69] D. Gupta, B. Mood, J. Feigenbaum, K. Butler, and P. Traynor, "Using intel software guard extensions for efficient two-party secure function evaluation," in *International Conference on Financial Cryptography and Data Security*, 2016.

[70] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi, "Secure multiparty computation from sgx," in *International Conference on Financial Cryptography and Data Security*, 2017.

[71] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: functional encryption using intel sgx," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2016.

[72] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2018.

[73] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2016.

[74] 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.

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

[76] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug. 2017.

[77] 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.

[78] 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.

[79] 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.

[80] 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.