Version of Record: https://www.sciencedirect.com/science/article/pii/S0167404818302347
Manuscript_e9b559f708442841cf487ac201d3dd6d

SGX-LEGO: Fine-Grained SGX Controlled-Channel Attack and its Countermeasure

Deokjin Kim[a], Daehee Jang[b,*], Minjoon Park[b], Yunjong Jeong[b], Jonghwan Kim[b], Seokjin Choi[a], Brent Byunghoon Kang[b,**]

[a] *The Affiliated Institute of ETRI, Daejeon, Korea*
[b] *Korea Advanced Institute of Science and Technology, Daejeon, Korea*

Abstract

The introduction of Intel Software Guard eXtension (SGX) prompted security researchers to verify its effectiveness. One of the frequently discussed attacks against SGX is the side-channel attack by gathering page-fault information (controlled-channel attack). Owing to SGX's hardware features, the faulting address of the enclave memory is *page-masked*. Because of this, both the controlled-channel attack and the defenses of SGX are built under the assumption that an attacker observes the memory access attempts of the enclave code with *page-granularity*. However, Van Bluck et al. recently demonstrated a controlled-channel attack technique which negates the prior assumption of *page-granularity*. In this paper, we introduce a new class of attack that stems from the reduced controlled-channel granularity, i.e., the Version IDentification attack (VID). The goal of VID attack is identifying the detailed code information inside SGX enclave by analyzing the fine-grained SGX controlled-channel. To protect enclave memory from such attack, we design and implement SGX-LEGO, an automated system that adopts execution polymorphism to the SGX enclave code. Previous defense approaches against controlled-channel attacks can be broadly categorized into two types: (i) disclosing the fault information and (ii) rendering the fault

*Co-first author
**Corresponding author
Email addresses: deokjin.kim@gmail.com (Deokjin Kim), daehee87@kaist.ac.kr (Daehee Jang), dinggul@kaist.ac.kr (Minjoon Park), yunjong@kaist.ac.kr (Yunjong Jeong), zzoru@kaist.ac.kr (Jonghwan Kim), choisj@nsr.re.kr (Seokjin Choi), brentkang@kaist.ac.kr (Brent Byunghoon Kang)

information useless. SGX-LEGO uses the latter approach by permuting the memory access sequence at the instruction level. In SGX-LEGO design, we leverage the concept of code-reuse-programming to overcome the implementation challenges regarding SGX page management. In the evaluation, we show how VID attacks the cryptographic functions, and demonstrate the efficacy of SGX-LEGO in security perspective and performance.

Keywords: Operating System, Intel SGX, Controlled-Channel, ROP, Page Fault

1. Introduction

Intel Software Guard eXtension (SGX) provides confidentiality and integrity to an application even if the underlying privileged software, such as the operating system (OS) or hypervisor, is untrustworthy. One of the most actively researched topics regarding SGX is the side-channel attack. Side-channel attacks against the SGX environment are usually aimed at extracting memory contents from the enclave memory. So far, various SGX side-channel attacks [1, 2, 3] have been introduced. In particular, Xu et al. [1] introduced an SGX side-channel attack referred to as *controlled-channel attack* that is based on the observation of page-faults.

So far, controlled-channel attack revealed *data* inside SGX enclave assuming the *code* was publicly available. Revealing the code information inside SGX enclave was known to be infeasible once the SGX binary is first encrypted and dynamically decrypted later inside enclave memory. For example, a previous work [4] encrypts the code section (private code which contains the main logic) and decrypts it with a public code fragment (allowed to be exposed) at runtime using the decryption key provided by the trusted remote party over the network. Once the static binary analysis is stopped in this way, extracting the code information at runtime is supposedly prevented because (i) SGX protects enclave memory from direct read attempts, and (ii) the granularity of controlled-channel attack is too big to extract the code information. We point out that (ii) is no

longer true due to the introduction of fine-grained controlled channel. In this paper, we explore the SGX controlled-channel attack in terms of *code disclosure attempt*.

The essence of the controlled-channel attack is extracting memory access patterns using page-fault information. So far, researchers presumed that this memory access pattern can only be observed with page granularity[1] which is big enough to hide the detailed execution trace. Previous SGX side-channel attacks assume that the OS can only observe the faulting sequence of *distinctive pages*. In other words, consecutive page faults against the same page cannot be observed outside the enclave. This is important primitive regarding controlled-channel attack. For example, an adversary can monitor the code page fault and realize that the control flow has reached the page, but due to this primitive, the adversary cannot tell how many instructions were executed inside such page. However, recent work [3] demonstrated a technique that allows malicious OS to observe consecutive memory access attempts against *same page* therefore breaking this primitive.

The previous work [3] discuss the ramifications of their technique in terms of SGX enclave data exposure. In this paper, we extend their discussion and demonstrate that this new development[2] not only advances the efficacy of existing controlled-channel attacks against data but also enables the attacker to reveal the *code* inside enclave memory. According to our experiments, the information inside the SGX enclave such as the code algorithm, SDK library version, and their configuration is no longer safely hidden owing to fine-grained controlled-channel attack. To study the extent of this problem, we define the concept of Version IDentification (VID) attack against SGX and conduct experiments on its issues. The idea of a VID attack is simple and straightforward. Since the page granularity of fault monitoring is broken, an attacker can harvest the code

[1] Code page fault only occurs when the memory address of the program counter jumps to another code page

[2] Referred as instruction-granularity fault monitoring

page access attempts inside the SGX enclave at a fine-grained level. Based on more detailed information of such memory access events, various information of the running code inside the SGX enclave can be inferred regardless of the data they are using.

In general, observing system call information such as their sequence and parameters can be an effective approach in order to fingerprint an application's code identity. However, SGX applications typically consist of a non-enclave part and enclave part. Codes running inside the enclave portion usually process security-sensitive in-memory data without involving system calls (e.g., cryptographic operation, image processing). In addition, system call information cannot distinguish between different build environments (e.g., different compiler optimization level) of the same software. To mount sophisticated exploits (e.g., involving ROP gadgets), the attacker seeks detailed information about the target software, including its detailed software version. In this paper, we assume the side-channel attack and defense focused to the code page access.

As memory access patterns can be observed from the outside world with finer granularity, deterministic memory access patterns can be used as a fingerprint for identifying the exact program version inside the SGX enclave. For example, extracted memory access patterns can be compared to such patterns of previously known programs (enclave application and non-enclave applications both). We later show the experimental results for this *deterministic pattern extraction and comparison*. This attack becomes effective as SGX SDK libraries (and other library codes) are increasingly shared among developers. We provide more details about this attack in section 3 and demonstrate the experimental results in section 6.

To address the threat of fine-grained controlled-channel attack (including the VID attack model), we design and implement SGX-LEGO: a binary conversion framework that adopts execution polymorphism for SGX applications. The goal of SGX-LEGO is to remove discernable memory access patterns (including consecutive access to the same page) while the code is running inside the enclave. Several techniques can be considered for removing discernable memory access

patterns inside the SGX enclave. For instance, heavy obfuscation (e.g., VM-based obfuscation with added randomness) can be considered as a solution. However, the SGX environment lacks dynamic page permission management[3], which is essential for implementing polymorphic binary. Polymorphic execution can also be implemented based on RWX[4] memory without using dynamic page management. However, the use of RWX memory is discouraged for software security [5][5].

One of the main contributions of SGX-LEGO is that it achieves polymorphic execution (thus randomized memory access pattern) without using dynamic page management or RWX memory. To satisfy this requirement, SGX-LEGO leverages the concept of code-reuse programming (CRP[6]) technique. In general, CRP is utilized by attackers to bypass the DEP enforcement where RWX memory is not allowed. Here, we use it as a defensive measure against side-channel attack. SGX-LEGO is composed of 1,700 lines of C/C++ and 1,100 lines of Python. We explain further in section 5. We have implemented and evaluated SGX-LEGO in the following environments: Windows 10 Pro 64-bit, Intel i7 SkyLake. Additional details of the evaluation are provided in section 6.

The contributions of this paper can be summarized as follows.

- **The strong(er) attack model**. From previous literature, it is well established that Intel SGX allows attackers to extract data from the enclave memory using controlled-channel attack. In this paper, we show the extension of this attack model and demonstrate that such side channels can also be used for identifying the code and its detailed version running inside the SGX enclave memory. We evaluate the efficacy of the newly

[3]Recent generation of SGX hardware supports dynamic page permission. This paper is based on SGX spec 1

[4]Readable, writable, and executable

[5]Google Chrome and Microsoft Edge are removing the use of RWX pages from security-sensitive renderer process using various techniques

[6]Also known as return-oriented programming (ROP). We use the term CRP and ROP interchangeably.

proposed VID attack model using the instruction-granularity page-fault monitoring technique, which was discovered from previous work [3].

- **The new defense scheme**. We leverage the traditional concept of CRP (which originated from ROP attack) technique to defend the newly introduced attack model. SGX-LEGO automatically converts the given program into small pieces like LEGO parts (hence, SGX-LEGO), and then generate the polymorphic payload[7] to eliminate discernable page access patterns while preserving the semantics of the original input program. This approach removes deterministic page access patterns without involving the use of dynamic page permission change or RWX memory.

The remainder of this paper is organized as follows. section 2 provides the background on the SGX issues and side channel attacks. section 3 explains the VID attack. section 4 discuss the design and overall architecture of SGX-LEGO system. section 5 then explains the details of SGX-LEGO implementation and section 6 shows evaluations of VID attacks and SGX-LEGO benchmarks regarding their effectiveness and performance. section 7 discusses additional issues of this paper and limitations of our work. In section 8, we enumerate various related works and explain the relationship between such works and this paper. Finally, section 9 concludes the paper with a brief summary.

2. Background and Assumption

2.1. Basic Background of SGX

Intel SGX is one of a promising technology for the safe computing environment. It provides the extended instruction set listed in Table Table 1 to ensure integrity and confidentiality of application that has sensitive code and data. It helps the application to be executed in a secure container, so-called *enclave*. It protects the

[7] In this paper, payload indicates the set of addresses and function parameters for ROP execution

Mnemonic (Opcode)	Leaf-Mnemonic (EAX)	**Description**
Supervisor Mode		
ENCLS (0F 01 CF)	ECREAE (00)	Create an enclave
	EADD (01)	ADD a page
	EINIT (02)	Initialize an enclave
	EEXTEND (06)	Extend EPC page
User Mode		
ENCLU (0F 01 D7)	EGETKEY (01)	Create cryptographic key
	EENTER (02)	Enter an Enclave
	ERESUME (03)	Re-enter an Enclave
	EEXIT (04)	Exit an Enclave

Table 1: Intel SGX Instructions. The EAX register is used to point out each specific leaf-instruction. The leaf-instructions in same mode have the same opcode.

application against various kinds of malicious entities (such as kernel privileged malware, or physical attacks) outside the enclave.

The enclave is set up by the operating system that maps the virtual addresses of the enclave and the physical addresses of specific memory regions, namely Enclave Page Cache (EPC), which is stored in Processor Reserved Memory (PRM). PRM is a contiguous subsection of physical memory that is pre-defined by the BIOS. After initialization, even highest privileged software such as operating system and Virtual Machine Manager (VMM) cannot acquire any information inside the enclave, except the side-channel information like memory mapping and cache delay time. According to ISCA 2015 technical documents [6], the code inside SGX binary exists in the form of plain text before instantiation as it requires runtime measurement via EINIT instruction. Code and data can be first encrypted and later decrypted by a key. However, the key should never exist inside SGX application. Decryption key must be provided from the external network after secure attestation step.

2.2. Page Fault and SGX

The attack model of Intel SGX assumes all the components required by an application other than the application binary code and CPU; are not trusted (including the operating system). This is a strong attack model which enables numerous side channels to attack the system. One of a well-known side channel which enables the attacker to undermine the SGX security is *page-fault based side channel attack*. To run an application, there are several inevitable operations which require assistance from the OS. Page-fault is one of such requirement.

Since the page-table and MMU management is handled by operating system, the OS can deliberately manipulate the memory page access permission referenced by SGX application[8]. The faulting address of page fault potentially leaks information inside the enclave. To prevent this side-channel attack, Intel SGX CPU conceals the exact faulting memory address and only provides the page number to interrupt handler when such event occurs. More precisely, when the execution of SGX enclave is interrupted by an external event, Asynchronous Enclave Exit (AEX) handler is invoked before the OS interrupt handler is invoked. While AEX step is processed, various information, including lower 12 bits of page faulting address, are erased before OS can access. Therefore, the OS cannot know the exact faulting address inside enclave memory other than the page number. However, numerous previous works [2, 4] showed that using only page number still can reveal sensitive information inside the enclave.

2.3. Page Access Monitoring in SGX

There is a major difference between SGX application and normal application regarding page access monitoring. In general, data memory access and code execution can be monitored by an operating system (using page faults) with *single instruction granularity* whereas no such monitoring granularity is provided in SGX environment. The reason is that OS must leverage breakpoints (or instruction

[8] Read/write/exec permission of enclave page is not affected by the operating system. However, the operating system can cause page-fault against such page.

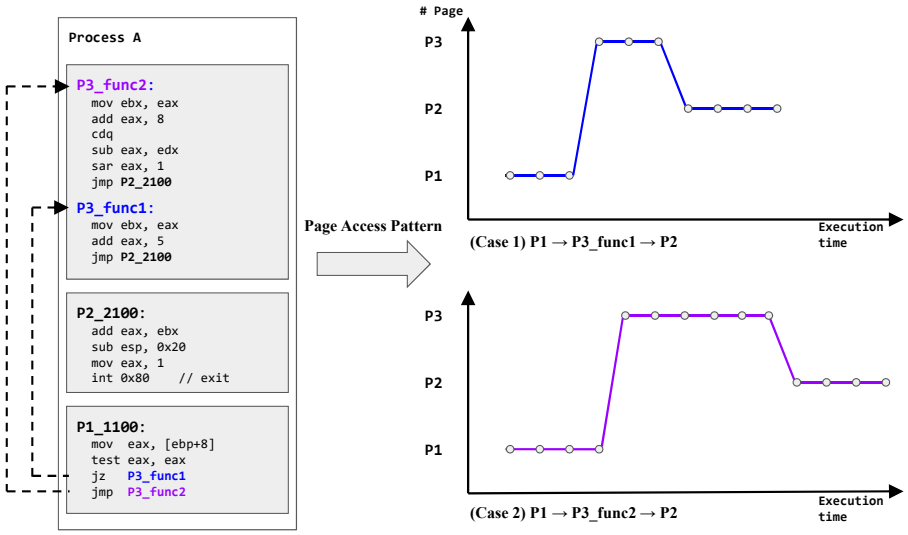


Figure 1: Instruction granularity page fault monitoring.

traps) to monitor program behavior per each instruction. For example, consider the following scenario: (i) OS wants to monitor entire memory access against data page `Page-A`. Thus OS first removes access permission of `Page-A`. (ii) At some point, a memory access instruction of SGX application touches data page `Page-A` and fault occurs. (iii) OS records the first access attempt against data page `Page-A`. (iv) OS gives access permission to `Page-A` before resumes the application (unless application hangs). (v) To catch second memory access attempt against data page `Page-A`, OS must remove the access permission of `Page-A` again before the next instruction executes. To do so, OS places software breakpoint (or set trap flag, etc.) at the next instruction and traps the application again.

From step (v), OS must know the exact faulting instruction address and also requires memory access permission (or register access permission) to place breakpoint so OS can re-adjust page access permission. This procedure is infeasible under SGX environment as enclave memory is protected from OS. Therefore, memory access (or instruction execution) monitoring technique using page fault; significantly differs between general application and SGX enclave. OS is unaware of all events between page faults among different two pages.

2.4. Code Reuse Programming

Code Reuse Programming (or Return Oriented Programming) is a programming method that utilizes the concept of *Code Reuse Attack* (CRA) which is a known technique introduced by various previous works/tools such as *Q* [7], *Mona* [8], *ROP Compiler* [9] and *JIT CRA Compiler* [10]. The major difference between CRP and CRA is that there is no need to find any gadgets (small code fragment) in CRP as the programmer can intentionally place any gadget that is necessary to the application. We refer to the set of gadgets as *gadget corpus* in this paper. The *gadget corpus* is automatically created by taking legacy binary as an input of SGX-LEGO system. Unlikely to existing CRA tools, SGX-LEGO adopts randomness to payload generation to avoid deterministic memory access patterns.

2.5. SGX Remote Attestation

SGX provides an environment such that enables enclave to acquire sensitive information from trusted sources; this is so-called *remote attestation*. To initiate the *remote attestation*, enclave must attest itself to guarantee that every component is intact. The following is the steps for remote attestation. First, the enclave calculates its hash and signs with the attestation key derived from the CPU. The signing information is delivered to *quoting enclave* (QE) which is an architectural enclave included in Intel SGX SDK. QE verifies the message, and if the message is valid, QE uses EPID private key to sign the message and send it, called the *quote*, to the remote verifier. Remote verifier uses EPID public key to verify the *quote*. If the chain of verification succeeds, network channel for communication is encrypted by the session key made from the remote attestation and the remote verifier sends sensitive secret information to the enclave. The cryptographic background of remote attestation is based on SIGMA protocol which is an enhancement of Diffie-Hellman. We use such remote attestation to securely receive sensitive control logic of enclave which is consisted of a ROP payload. Details are explained in section 4.

3. Version IDentification Attack

SGX environment assumes a fully privileged adversary such as OS and hypervisor. OS is capable of capturing the page access event of any application running on top of them. This configuration allows the fully privileged attacker to inspect the memory access pattern of SGX enclave. SGX prevents such information gathering attack by masking off the offset information of page fault event at a hardware level. For example, if a page fault occurs at memory address `0x07654321` where the memory page is 4Kbytes, the hardware signals the OS and gives the address `0x07654000` which is an address that the page offset information is removed. Additionally, section 2 explained page fault monitoring against SGX application is coarse-grained as the per-instruction trap is inapplicable. Therefore the execution of SGX application only results in a sequence of accessed page numbers. Although the same page is consecutively accessed multiple times (by multiple instructions), only the first access can be observed from outside.

However, according to the recent work on SGX controlled-channel attack [3, 11], multiple executions of instructions (or memory access attempts) inside the same page can be distinguished by an attacker. Given that the number of instructions executed inside the same page can be counted due to finer-grained page fault monitoring, it is possible to generate discernible code execution pattern using page access trace. Figure 1 describes the per-instruction counted page access patterns.

Without the capability of instruction counting, page fault information of Figure 1 would be observed as `P1-P3-P2`. However, assuming each instruction inside the same page raises distinguishable faults, the observed fault information would become `P1-P1-P1-P3-P3-P3-P2-P2-P2-P2` which is more detailed than the previous version. Based on such development, we show that it is possible to identify codes and their detailed information inside unknown SGX enclave. According to our experiments, instruction-granularity page fault information (which includes a number of repeated access to the same page) can reveal the identity of code inside SGX enclave. This is done by gathering enclave page

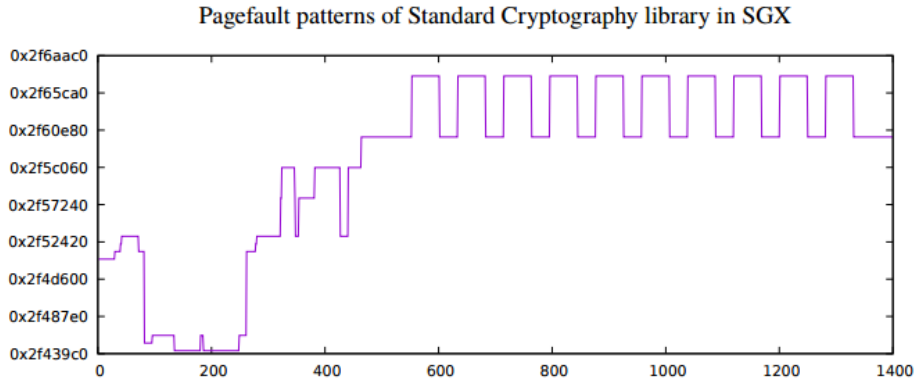


Figure 2: Fine-grained page access pattern of AES-GCM code in Windows SGX SDK. X-axis is number of executed instructions and Y-axis is address of faulting code page while executing the instruction.

access patterns and comparing them with a previously known set of patterns. This becomes particularly effective in case the SGX enclave uses a standard public library such as SGX SDK.

Figure 2 is the result of gathering the exact addresses of 1,400-page faults occurred by each code execution from an ordinary cryptographic application program. The x-axis indicates the number of instructions executed (each execution causes code page fault) and Y-axis indicates the faulting page address. Although we take into account that page-offset information is removed from SGX environment, the (page masked) address information still leaves access pattern based on its execution path. From the page access pattern result, continuous execution mostly occurs within the same page. However, functions are usually composed of multiple subroutines. The codes of each subroutine are likely to have distance more than one page. Also, so far, the page access patterns within same code page yielded only one-page fault no matter how many instructions were executed. However, given that the page faults raised by each instruction within the same page can be counted, we have an additional dimension of page access pattern graph, which is the length of X-axis. Comparing the similarity of such patterns with a known set of pattern database can reveal the detailed

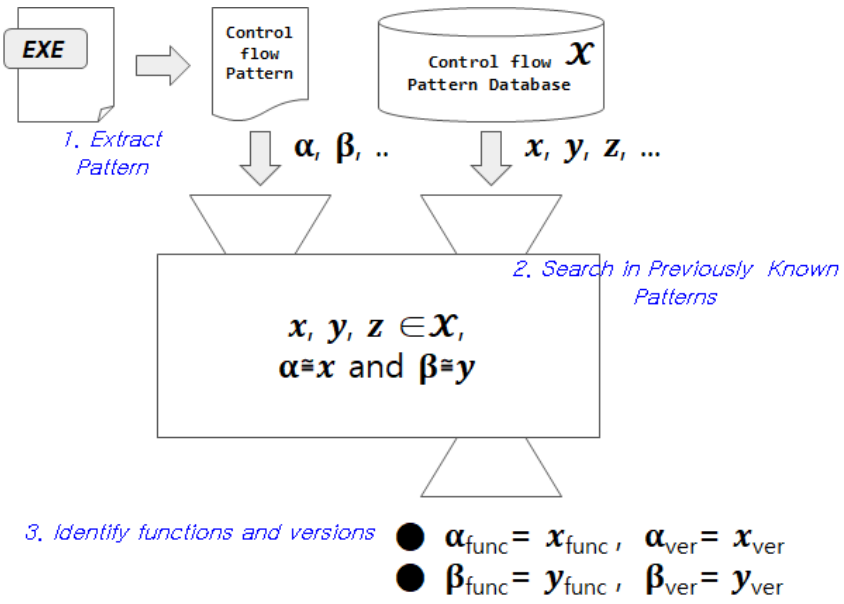


Figure 3: VID Attack Process Overview

identity of code inside SGX enclave. We refer this code disclosure attack against SGX as Version IDentification (VID) attack.

The overall procedure of VID attack can be depicted as Figure 3. First, we setup a database of universally known patterns from public SGX SDK libraries (i.e., SDK versions 1.x). The database should include page access pattern information of various algorithms of libraries. Using page access pattern database of publicly known library functions, we can deduce which algorithms are being used inside unknown SGX enclave by extracting its fine-grained page access pattern and finding the pattern match from a database. We demonstrate actual result in section 6 using cryptographic functions of SGX SDK library.

4. SGX-LEGO

To harden the SGX from fine-grained controlled-channel attack, we propose SGX-LEGO, a binary conversion framework for removing discernable patterns of code execution. The simplest way for eliminating deterministic execution pattern

is using polymorphic code[9]. However, making polymorphic code requires dynamic generation of the executable page at runtime or RWX pages. Unfortunately, dynamic alteration of page permission is disallowed in early SGX hardware; and using RWX page is discouraged[10] regarding security as it violates DEP enforcement.

To achieve polymorphism without involving any dynamic change of page permissions nor the use of RWX memory, SGX-LEGO leverages ROP attack mechanism. In software vulnerability exploitation, ROP attack is leveraged to bypass the DEP enforcement where RWX memory is unavailable. The insight here is that, if we use the concept of ROP, execution polymorphism can be implemented without involving dynamic page permission changes nor RWX memory. To launch ROP attack, attacker stitches a proper sequence of existing gadget codes and ultimately achieves semantically malicious overall execution. The sequence and address information of gadgets are stored as *payload* (information that determines execution order) which is pure data that only requires readable-writable memory. Thus adopting polymorphism against ROP-style execution with *payload* do not involve any requirement of dynamically assigned executable memory. The procedure of converting existing code into ROP-style payload is fully automated, and we add polymorphism to ROP execution for eliminating deterministic patterns that can be used for pattern analysis.

In short, SGX-LEGO disassembles the given binary and automatically generates a minimal pool of small code fragments (ROP gadgets) and a data sequence of their execution (ROP payload). Automatic generation of ROP attack payload was demonstrated by previous work Q [7]. However the goal of Q is generating payload against given binary for attack purpose thus there is no issue of (i)

[9]Code fragment that changes its form at runtime while preserving the execution semantic of overall algorithm

[10]To quote Intel SGX manual, "The ideal enclave would also have a defense-in-depth mechanism that ensured that all sections containing executable code would also be non-writable."

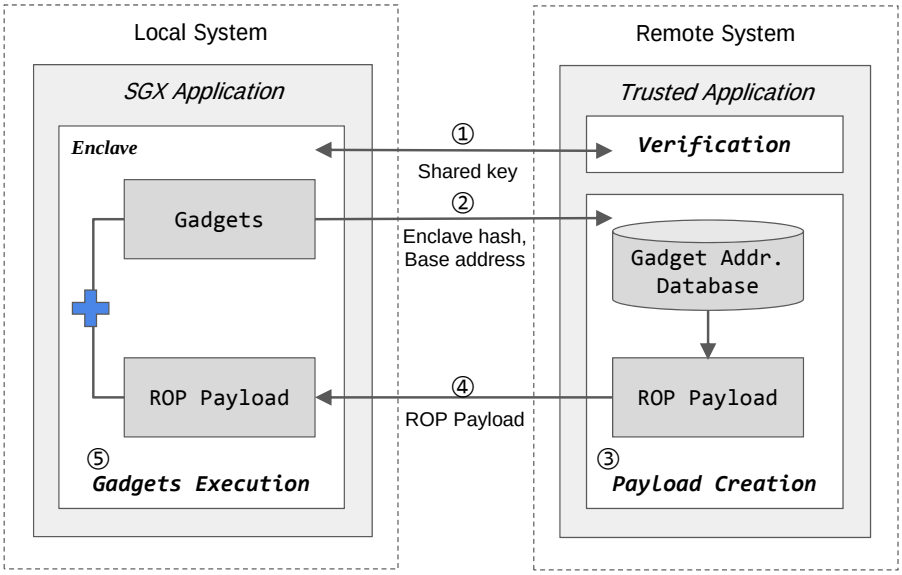


Figure 4: Overall Design of SGX-LEGO System.

minimal gadget pool generation, (ii) adopting execution polymorphism, and (iii) performance. SGX-LEGO performs optimized gadget generation and adopts polymorphism to automatically generated payload for defense purpose. We show a detailed comparison between SGX-LEGO and Q in section 8. Figure 4 illustrates the overall design of SGX-LEGO framework.

SGX-LEGO is composed of a local and remote system. Gadgets inside local application use constant number of *readable-executable but non-writable pages*. However, the payload for execution is generated from the remote system. Once the local system initiates, standard SGX remote-attestation is processed, and the remote system generates polymorphic payload and transfers it to the local system. Payload generation and transfer are divided into multiple times in case the program is too big thus the stack or heap size of the local system is insufficient to hold the entire payload for execution. SGX-LEGO assumes that gadget codes for ROP-style programming is not secret information, but considers the payload as secret.

To achieve execution polymorphism, SGX-LEGO uses following strategies: (i) distribute gadgets depending on their access frequency and make the page access to be evenly distributed, also place multiple gadgets at different locations that are semantically-equivalent with each other and choose it randomly while payload generation. (ii) Apply randomness to stack payload positioning thus change the stack payload layout and randomize its page access patterns. Using such methods, the execution sequence and memory access patterns of gadgets (codes) and payload (stack) are randomized. However, this methodology still leaves memory access to data pages such as BSS segment and heap to be deterministic. The current version of SGX-LEGO implementation does not consider data memory access randomization. Previous works [12, 13, 14] addressed this issue. The main focus of SGX-LEGO is to protect the identity of code information inside SGX enclave.

4.1. Non Deterministic Gadget Access

One advantage of using ROP-style execution is that the amount of code can be reduced compared to the original binary. For example, SGX-LEGO reduced the required amount of code pages of simple crypto application down to 30% by adopting ROP-style execution. If an application is tiny enough, the entire set of gadgets (generated from the original input binary) can fit into a single memory page. In such a case, the discernable pattern cannot exist regardless of the execution path. In such a case, SGX-LEGO does not adopt any randomization as there is no pattern other than constantly accessing the same page. However, in reality, the amount of required gadgets to run an application easily exceeds the boundary of a single page. Therefore depending on the execution path, the page access sequence becomes distinctive and recognizable.

In case the pages required for gadgets are more than one (which is the general case), SGX-LEGO eliminates discernable access patterns by applying *even distribution policy and a duplicate set of gadgets*. Even distribution prevents inferring the existence of frequently used gadgets inside a particular page. For example, if page `A` contains frequently used gadgets and page `B` contains rarely

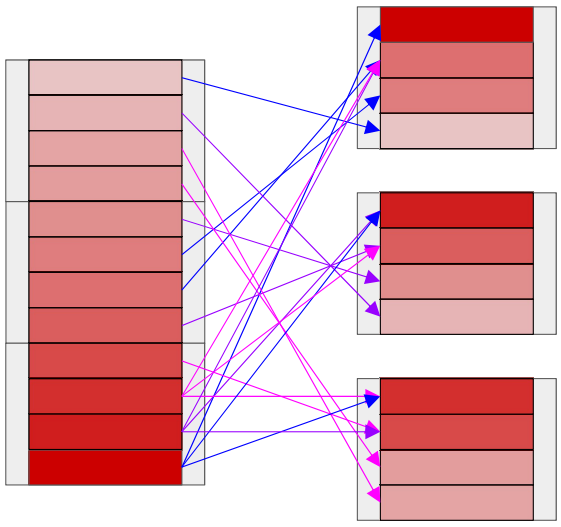


Figure 5: Evenly distributed gadgets in SGX-LEGO. Each block represents a gadget and the brightness of the block implies the access frequency.

used gadgets, page access pattern will be weighted to `A`. To prevent such a result, SGX-LEGO considers the access frequency of each gadget. Even distribution policy pre-defines the access frequency of each gadget (in average) and place them accordingly to distribute the page access attempts evenly. If binary is composed with two frequently used gadgets `mov`/`jmp` and two rarely used gadgets `les`/`halt`, SGX-LEGO will group `mov` and `les` into same page (assuming only two gadgets can fit into single page for simplicity of explanation) then place `jmp` and `halt` into another page.

SGX-LEGO also considers the repeated use of the same gadget. To avoid distinctive access patterns due to excessive use of specific gadget, SGX-LEGO places semantically duplicate gadgets on multiple pages so that payload can randomly select different gadgets even if the execution is repeated. Figure 5 is simplified illustration of SGX-LEGO gadget access randomization. In Figure 5, each block indicates gadget and the color indicate their access frequency (dark color indicates more frequently used gadget). The left side of the figure is

required gadgets for program execution. In case all gadgets cannot fit into the single page, SGX-LEGO evenly distributes such gadgets considering their access frequency; and places duplicate gadgets for frequently accessed gadgets and make the payload to randomly choose the necessary gadgets from multiple candidates of possible pages. By increasing the number of duplicate gadget codes among different pages, the randomness entropy of gadget page access pattern increases, however, overly extensive use of duplicate gadgets will waste memory usage.

4.2. Non Deterministic Payload Access

The *fit-into-single-page* situation from the previous subsection is also applied to payload page (stack for ROP execution) randomization. If the payload for program execution is tiny, the entire payload can fit into single data page thus avoid any discernible page access pattern. However, such an ideal situation is unlikely. SGX-LEGO applies randomness to payload page access patterns by randomizing the payload location with stack pivoting techniques and reordering semantically equivalent gadget execution. The key point of payload page randomization leverages the fact that the content and order of payload are semantically equivalent to executable codes however its memory access property does not require executable permission. Therefore we can adopt execution polymorphism with statically given readable-writable pages only.

5. Implementation

In this section, we discuss various components that implement SGX-LEGO system. SGX-LEGO is implemented using C/C++ and Python language. The CPU architecture is assumed to be 32bit x86. SGX-LEGO is primarily composed of the followings: (i) Disassembly Module, (ii) Gadget Generation Module and (iii) Remote System. Figure 6 summarizes the implementation components and their execution steps.

In order to obtain assembly codes as the input for SGX-LEGO system, we use existing disassemblers such as IDA Pro[15], Hopper[16], Dyninst[17]. The current

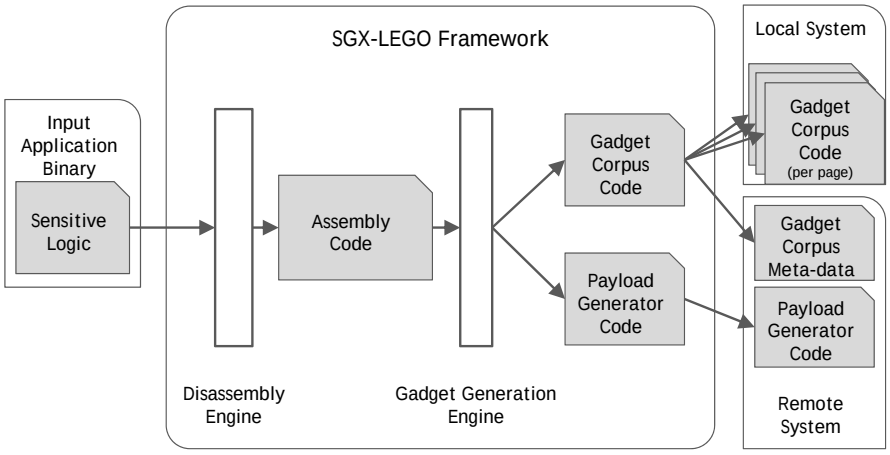


Figure 6: SGX-LEGO System Overview

implementation of SGX-LEGO is based on IDA Pro. The Disassembly Module is a wrapper of IDA Pro disassembly engine that automates the extraction of assembly codes into the proper format used by Gadget Generation Module. While extracting the assembly codes, address map information regarding all necessary binary symbols is also obtained. Therefore SGX-LEGO does not consider symbolization problem as other reassembly works [18].

The Gadget Generation Module receives assembly codes as input and generates executable code segments that will be used by payload which executes semantically equivalent original code. We refer to the set of generated ROP gadgets as *gadget corpus*. The generation of gadget corpus is fully automated.

Gadget Generation Module first separates the code segment into non-control-flow related instructions and control-flow related instructions. Each of the original instructions is converted into the corresponding set of gadgets and payload for controlling its execution flow. To avoid one gadget affecting another gadget's execution context, register context should be saved and restored back each time when a gadget is executed. Also, some control flow instructions are affected by `EFLAGS` register content, and `EIP` register value. These instructions need to be carefully considered while instrumenting them for SGX-LEGO, which

uses ESP register for controlling the execution flow. In addition, side-channel related instructions (e.g., CMOVcc, SETcc) are handled as exceptions and not transformed other sets of semantically equivalent instructions.

The second operation is making the payload generator code for the remote system. The payload generator code will be used after SGX remote attestation step. The payload generator of the remote system dynamically creates the payload using the base address information of loaded gadget corpus as meta-data. After a small set of instructions are categorized and transformed into ROP-gadgets, randomization, and reordering steps are finally applied at the end of gadget generation procedure.

After gadget generation, SGX-LEGO generates proper payload to execute gadgets. This process is depicted in Figure 7. In case of the branch instructions, SGX-LEGO converts the offsets between codes into offset between payloads. For example, payload[137] in the figure stores 3 which is a ROP payload distance between payloads for proper gadgets. In case of the payload[189], the 0xfeedbeef will be replaced with return address at runtime.

Initially, enclave saves the payload into heap memory (which will be later used as a stack) and pivots the stack pointer to point the start address of payload. The payload is composed of gadget addresses to be executed in pre-calculated order and parameters of some gadgets such as CALL and JMP. Once SGX-LEGO reduces the required number of code pages by adopting ROP-style execution, duplicate gadgets (for randomization) are added therefore the overall amount of code page remains similar to the original input program. According to our analysis, an average gadget is composed of approximately 7 bytes, therefore a single page contains 580 gadgets on average.

SGX-LEGO instruments the original instructions in the form of gadgets and corresponding payloads. The instrumentation requires the temporal use of general purpose registers. However, the instrumentation must guarantee all the register information to avoid clobbering; unless the correctness of the program breaks. For example, we need to carefully consider the side effects of arithmetic operation as it implicitly changes the EFLAGS register state. Also, all the register

Figure 7: Example code snippets for SGX-LEGO transformation. Upper side is the case for JMP, Lower part is the case for CALL. Left side of the figure is the original code before SGX-LEGO transformation. Right side of the figure is the automatically generated code for payload generation.

values must be properly saved and restored while the instrumentation to prevent register clobbering.

Traditional compilers handle this issue by using the stack memory for temporal storage for registers. However, we avoid using such methodology as it involves discernable stack page access pattern that we aim to eliminate from the first place. As a solution, we use MM0 - MM7 registers for temporary storage in general circumstances where such registers are unused. Admittedly, this method is only plausible under the assumption that instrumented application does not use any of such registers. In case the original code fragment uses such registers, SGX-LEGO should reluctantly use memory space for saving register context. However, the current version of SGX-LEGO is only applicable to codes that do

not use such registers; which is a limitation at this point.

Simple instructions which are independent of a various execution context are easy to instrument. The challenging part is handling instructions which are highly dependent on context information such as conditional move or conditional jumps and indirect calls. Maintaining the register context (`EFLAGS` and others) to be intact is a matter of performance overhead, however, preserving the semantics of control flow transition requires special techniques. Listing 1 is a code fragment example of indirect jump (for if statement) conversion. SGX-LEGO calculates the distance of payload from the indirect jump relative offset (`ECX`) and holds such information with `EAX`. The `cmp` which affects `EFLAGS` is substituted with `cmovz` and the binary result (1 or 0) is multiplied to pre-calculated payload offset number inside `EAX`. In case the `cmovz` yields zero, the branch will not be taken.

Listing 1: Conditional Branch Gadget Generation Example

```
1  gc_jg:
2    mov eax, 0
3    mov ecx, 1
4    ; Set eax to 1, if 'ZF=0 and SF=OF'
5    cmovg eax, ecx
6
7    ; Get a distance to jump
8    ; from the payload pointed by ESP
9    pop ecx
10   imul ecx
11   imul eax, eax, 4
12   add esp, eax
13   jmp gc_hub_00
```

To adopt polymorphism, we use a special type of gadget so-called *hub gadget*. This gadget is randomly used during the original semantic of code execution. The hub gadget uses `rdrand` instruction which is provided by Intel for obtaining true randomness inside SGX enclaves. With random number provided by `rdrand`, hub gadget imposes various randomization procedure we discussed from the earlier section. To minimize discernable access patterns of hub gadget, we use

`JMP` instruction rather than `RET` (which requires additional stack page access) for deterministic control flow branch.

The frequency of invoking the hub gadget decides the effectiveness of security and performance overhead. If the frequency of hub gadget execution is rare, the performance penalty is minimized yet the effectiveness of pattern elimination is weak. On the contrary, overly frequent execution of hub gadget effectively eliminates discernable page access patterns, yet requires a high-performance penalty. We discuss the details of performance penalties in section 6.

6. Evaluation

Evaluation in this paper is based on the following environments: Windows 10, Visual Studio 2012 (for SGX SDK 1.1), and Visual Studio 2013 (for SGX SDK 1.7), Windows SGX SDK version 1.1, 1.7 and Linux SGX SDK. For generating memory access patterns and its visualization, we use SGX emulation mode provided by Visual Studio and a customized version of open source binary execution visualization tool [19].

6.1. VID Attack

The goal of VID attack is to distinguish and identify the code running inside SGX enclave memory by analyzing its page access patterns in fine-grained level. If the observed execution pattern has high similarity with the previously known pattern, an attacker can infer the code is running inside the enclave. Because cryptographic algorithms are designed to show exactly same execution pattern regardless of data/key contents (length matters), the fine-grained page access pattern will be a strong signature for identifying the code information[11]. It is obvious that page access pattern of different algorithms (e.g., Blowfish and RSA) would be different. However, the question is: How much difference can we tell, given that the granularity of previous SGX controlled-channel is reduced?

[11] During experiments; we noticed that ECDSA yields slightly different execution pattern depending on the input contents.

Can we tell the difference between the same algorithm but using a different library version? In case of a cryptographic algorithm, can we tell the difference of its build configurations, encryption modes by looking into its code page access patterns? In this section, we answer such questions.

The overall VID attack evaluation indicates that it is feasible to identify the code version information inside SGX enclave by analyzing the page access patterns. Figure 8 Shows the execution pattern of two applications that implements the same algorithm (AES-GCM) but using a different version of SGX SDK (Windows and Linux). Each of the execution leaves same code page access pattern repeatedly thus making uniquely discernable pattern even under the presence of current ASLR applied inside the enclave.

Even though an application implements identical functionality, we can observe that the fine-grained page access pattern changes drastically depending on various parameters such as Library types (e.g., OpenSSL, SGX SDK, etc.), minor software version, algorithm modes, build configuration (e.g., optimization level). Based on our experiments, we observed some variation of fine-grained execution pattern between different SDK versions and cryptographic algorithm modes. For changes in minor version and crypto modes, pattern differences were minimal. We conducted various experiments to see the differences in page access patterns of cryptographic algorithm depending on its minor version and operation modes. Figure 9(a) is the result of using Windows SDK version 1.1, and Figure 9(b) shows the pattern result of SDK version 1.7. Figure 9(c) is result of MD5 operation and Figure 9(d) shows the result of SHA1 both in Windows 10 bcrypt library. Overall, the patterns are identical in major portion, however, there are some portions with a subtle difference as well. Figure 10 Shows the difference between AES-GCM and AES-CTR modes. As shown in the figure, changing the mode affects the fine-grained execution pattern. However the modified portion is small. If the attacker can compare the specific clean signatures, it would be enough to distinguish the detailed version. In case the attacker compares the overall similarity considering added noises, it would be difficult to tell such information.

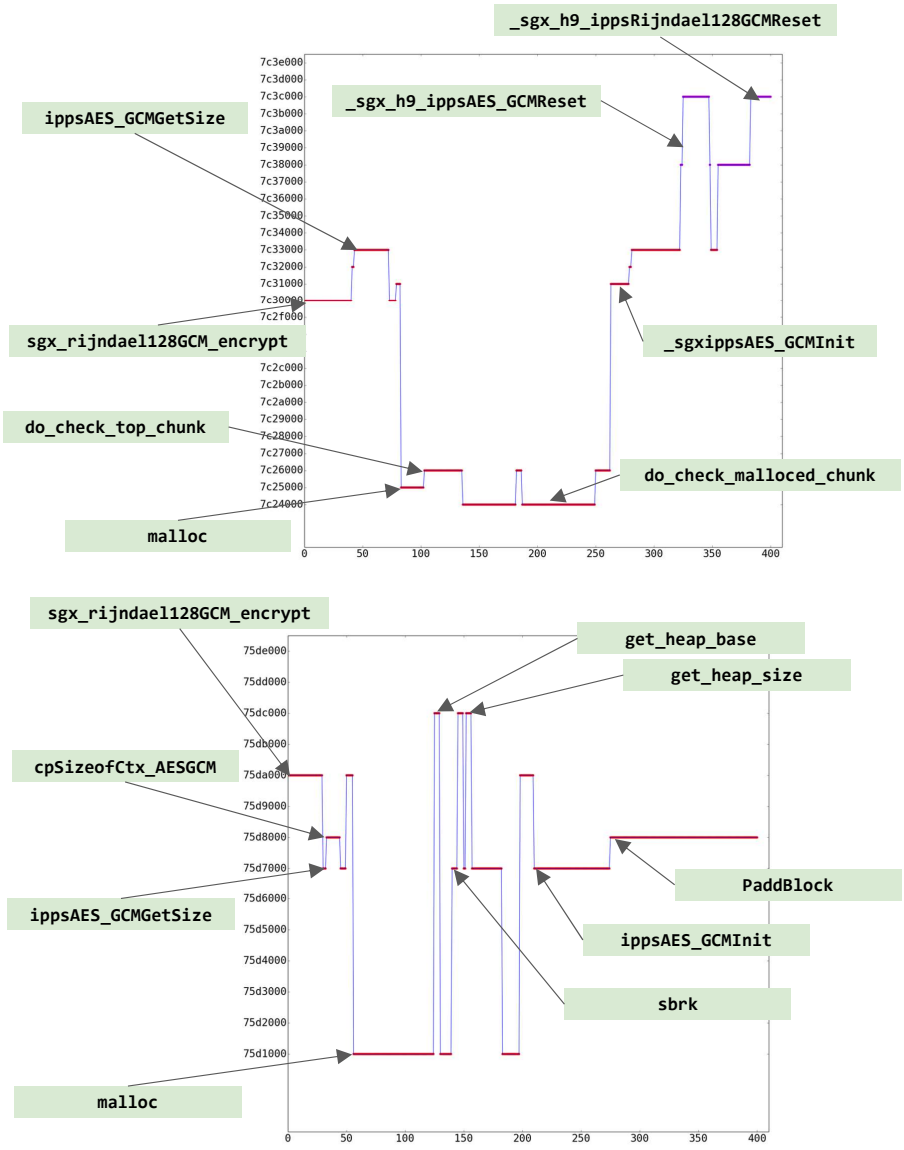
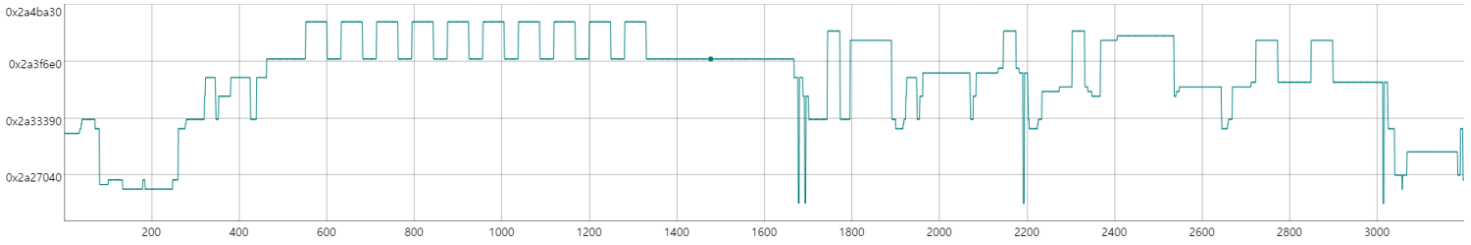
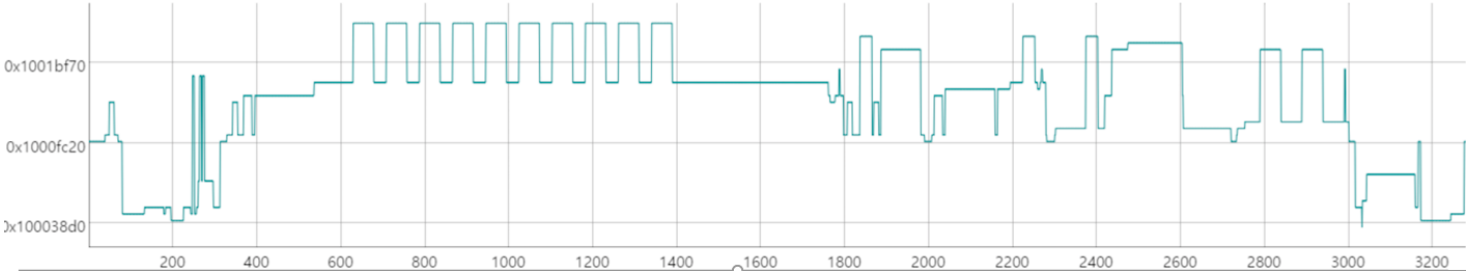


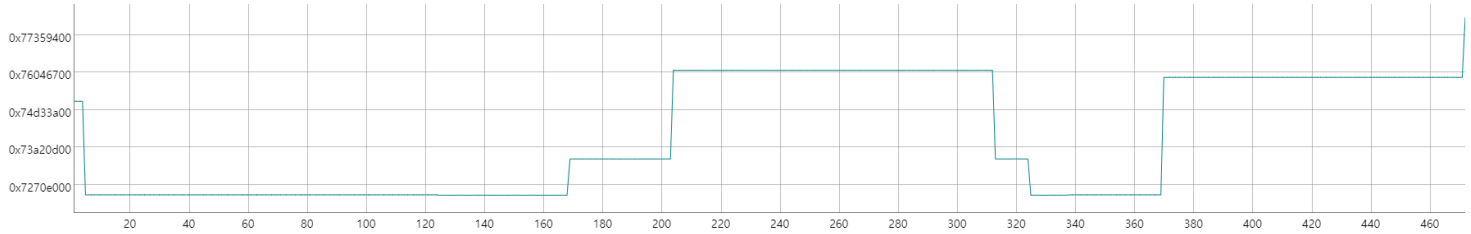
Figure 8: VID attack for distinguishing the Windows SGX Library (Upper side graph) and Linux SGX Library (Bottom side graph) version of AES implementation. The graph shows partial execution (initial 400 instructions). X-axis is number of executed instructions and Y-axis is address of accessed code page while executing the instruction. In the experiments, AESNI instruction was not used.



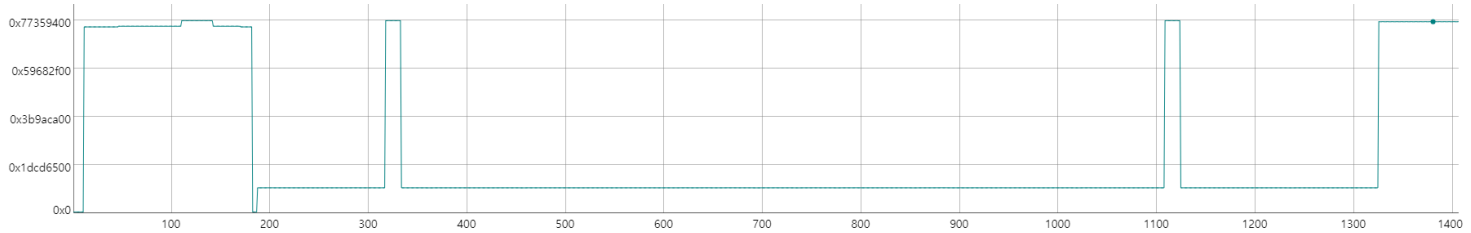
(a) AES-GCM 256-bit encryption and decryption in Windows SGX SDK `version 1.1`.



(b) AES-GCM 256-bit encryption and decryption in Windows SGX SDK `version 1.7`.



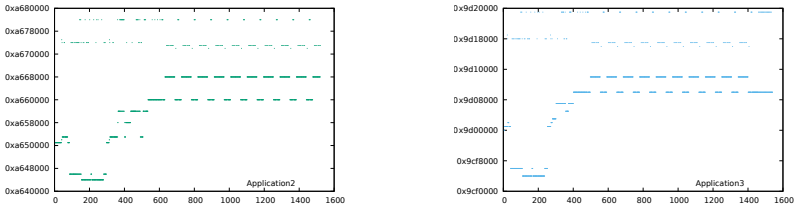
(c) MD5 checksum calculation in Windows bcrypt library.



(d) SHA256 checksum calculation in Windows bcrypt library.

Figure 9: Page access pattern (offset is masked) of various cryptographic operations. X-axis is number of executed instructions and Y-axis is address of accessed code page while executing the instruction[a]. Minor difference between SGX SDK 1.1 and 1.7 is due to trivial code updates

[a]AES encryption keys and input data do not affect execution pattern.



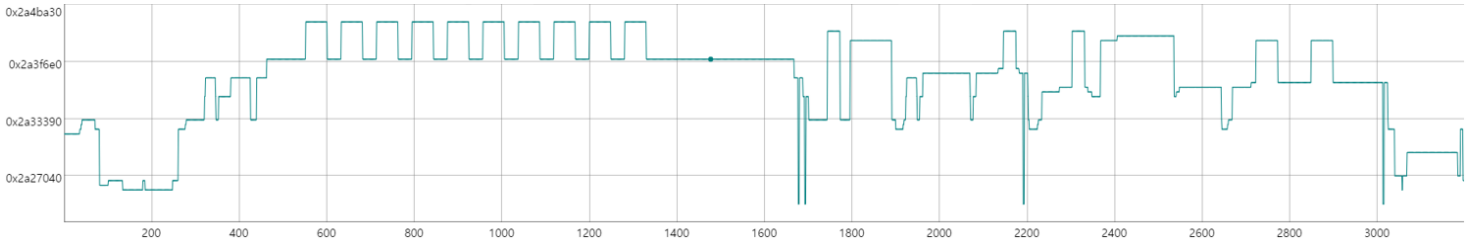
(a) Page fault pattern of Windows `AES-GCM`.

(b) Page fault pattern of Windows `AES-CTR`.

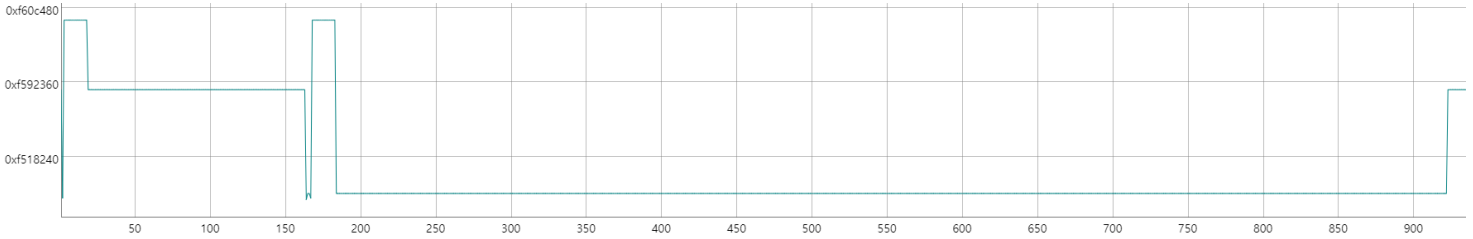
Figure 10: Fine-grained page access pattern of AES implementation that uses the same version of SGX SDK, but different configuration. Two graphs are showing patterns with different subtle changes. The x-axis is number of executed instructions and Y-axis is the address of accessed code page while executing the instruction.

In case the underlying implementation library is different, we can observe a significant difference in execution pattern. Figure 11 shows the page access patterns while executing AES in four different libraries. Figure 11(a) is the result of SGX SDK 1.1, Figure 11(b) is the result of Windows Crypt32 library, Figure 11(c) is the case of Windows OpenSSL 1.1, and Figure 11(d) is the result of TinyAES. In this case, we can observe that execution patterns are vastly different thus can be served as a unique pattern signature for identifying the code. With these pattern information, even considering some noises, an attacker can compare the similarity of the execution pattern and conclude the exact type of AES implementation with high confidence.

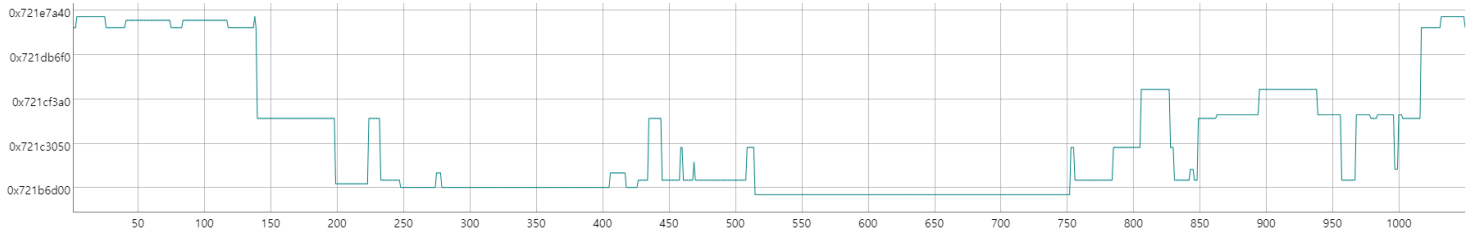
We also tested how the compiler optimization affects the fine-grained execution pattern. Figure 12 are the experiment results of extracting the pattern of TinyAES library code with different compiler optimization flags. Figure 12(a) shows the access pattern of TinyAES build with *Od (no optimization)*. Figure 12(b) and Figure 12(c) is the result of using *O1 (minimum code size), O2 (maximum speed)* flags during the source code compilation. Figure 13 is the optimization test result of Linux SGX SDK AES-GCM library code. We analyzed the output binary and confirmed that optimization reduced the number of codes thus placing some of the algorithm-relevant codes into different pages. In



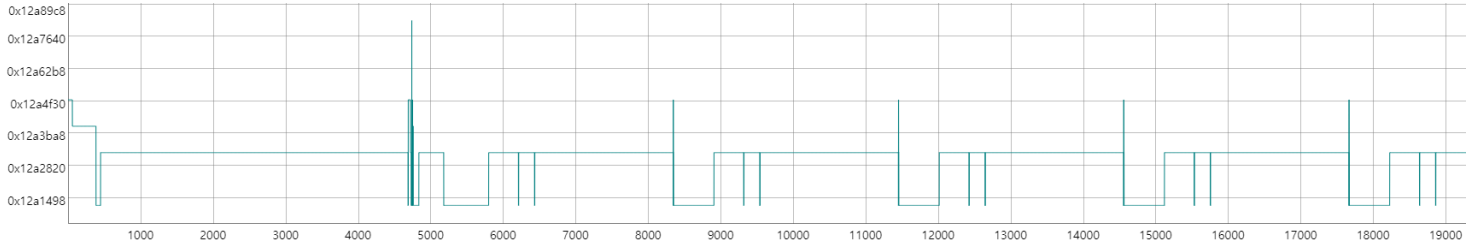
(a) SGX SDK 1.1.



(b) OpenSSL 1.1.



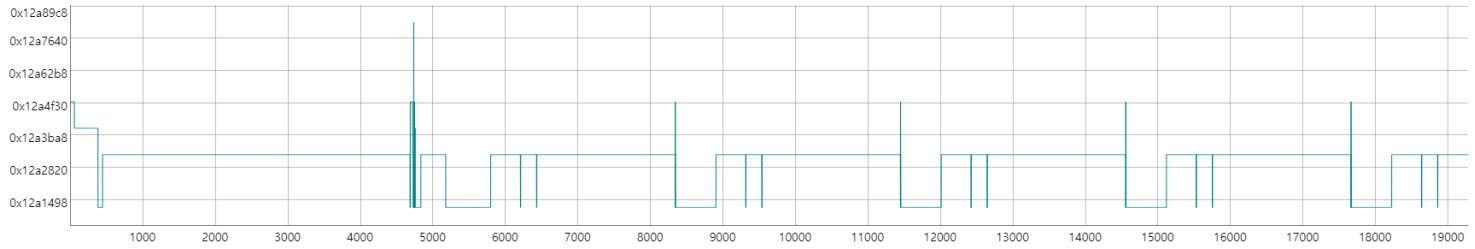
(c) Crypt32.



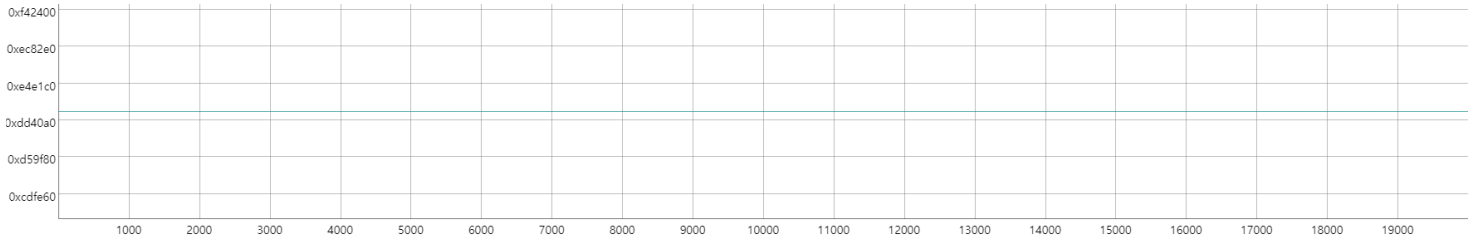
(d) TinyAES.

Figure 11: Page access pattern (offset is masked) of AES encryption and decryption in various libraries. X-axis is number of executed instructions and Y-axis is address of accessed code page while executing the instruction[a].

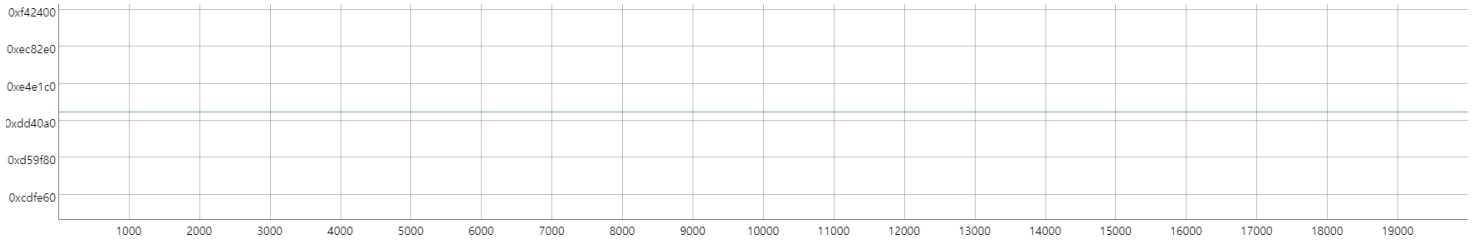
[a]Unlike Y-axis numbers, the graph representation is based on page-masked addresses



(a) Od (no optimization)



(b) O1 (minimize size)



(c) O2 (maximize speed)

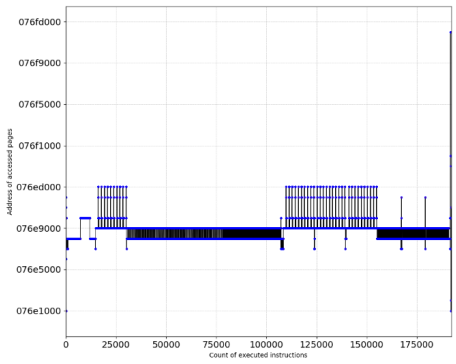
Figure 12: Page access pattern (offset is masked) of TinyAES code depending on various compiler optimization flags. X-axis is number of executed instructions and Y-axis is address of accessed code page while executing the instruction.

the case of TinyAES, because the binary was small, optimization reduced the entire code base to fit into a single page thus eliminating any observable page access pattern. In addition to the compiler optimization for code generation, we also investigated linker optimization and measured the difference in page access pattern. However, linker optimization did not result in any measurable difference.

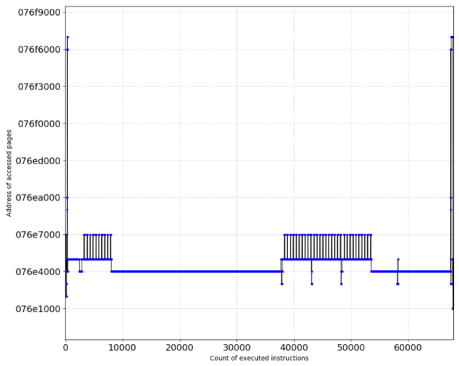
In summary, fine-grained page access patterns can be served as a signature for identifying the code inside an enclave. The similarity of patterns were significantly changed in case (i) the type of implementation library (e.g., OpenSSL and Crypt32) are different, and (ii) compiler optimization level is different. The change in patterns was small in case (i) minor library version is different (e.g., SDK 1.1, SDK 1.7), and (ii) cryptographic algorithm uses different internal modes (e.g., CBC, ECB, GCM).

6.2. SGX-LEGO

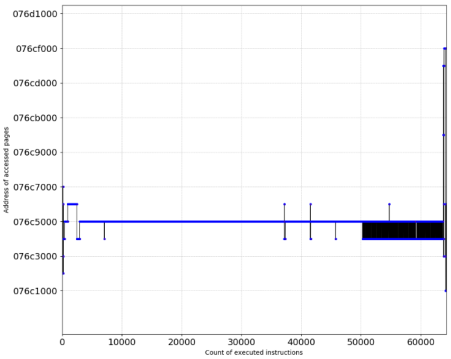
In this subsection, we demonstrate the fine-grained page access patterns before and after we adopt SGX-LEGO. To evaluate the performance of SGX-LEGO transformation, we use four applications that implement AES-GCM, AES-CTR, SHA256, and ECDSA. Figure 14a is the evaluation result of AES-GCM. The most left side of the figure Figure 14a is the original page access pattern, the center figure is after transforming the program with SGX-LEGO without running even-distribution process, and the right side of the figure is the result of SGX-LEGO transformation with gadget relocation process for even-distribution. Additional figures Figure 14b Figure 14c Figure 14d are the same evaluation result using AES-CTR, SHA256, and ECDSA applications. All of the original page access patterns are measured by running Linux SGX SDK codes compiled with default build options. In the case of AES-GCM and AES-CTR, SGX-LEGO places all the necessary gadgets into three pages. Because SGX-LEGO evenly-distributes the gadgets into multiple pages, an attacker cannot easily identify any signature patterns. In case the transformed code size is small enough (ECDSA, SHA256 case), SGX-LEGO puts entire gadgets inside a single



(a) Od (no optimization)



(b) O1 (minimize size)



(c) O2 (maximize speed)

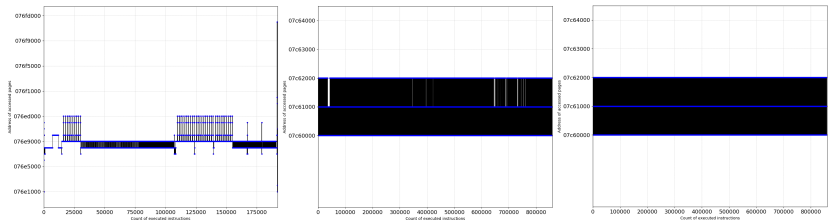
Figure 13: Fine-grained page fault pattern of Linux SGX SDK AES-GCM code with various build optimization.

Algorithm	# code page (R-X)	# data page (RW-)	program version
AES-GCM	13	6	Original
AES-CTR	7	6	Original
SHA256	2	2	Original
ECDSA	12	3	Original
AES-GCM	3	11	SGX-LEGO
AES-CTR	3	9	SGX-LEGO
SHA256	1	2	SGX-LEGO
ECDSA	1	6	SGX-LEGO

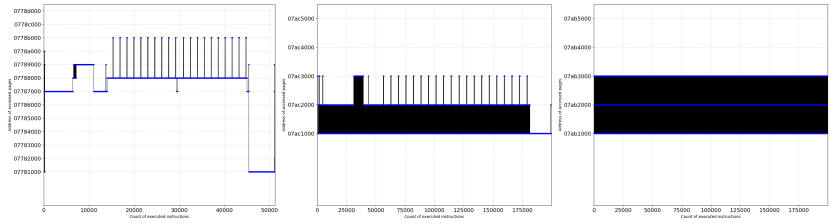
Table 2: Number of code/data pages before/after SGX-LEGO transformation. Only execution-related pages are counted. SGX-LEGO transformation reduced code pages and increased data pages.

page, thus eliminating any discernible fine-grained page access patterns. Our evaluation results do not reflect executions outside the enclave (e.g., EEXIT and EENTER via an interrupt).

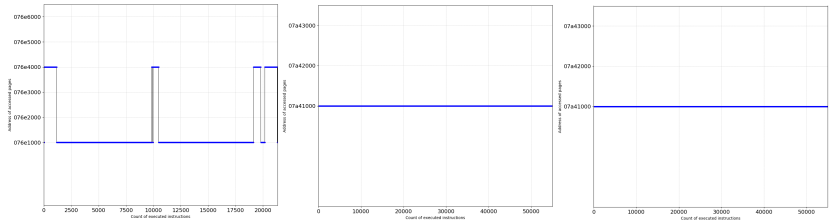
SGX-LEGO also evenly-distributes payload contents to hide access patterns in payload pages. Unlike the gadget distribution process, payload access distribution is challenging to distribute perfectly evenly. The current solution for hiding the payload access pattern is relatively simple compared to handling code pages. Therefore, the payload access normalization in ROP payload still leaves some patterns depending on the application. To overcome this limitation, we implemented a feature that makes dummy access between standard payload accesses to normalize the payload page access frequency. In this manner, we can thoroughly eliminate ROP payload access pattern; however it increases performance overhead up to x600 in the worst case. SGX-LEGO currently implements this feature; however we do not enable it by default due to high cost. From attacker's perspective, it would be straightforward to observe the code page access pattern than ROP payload pages which would be mixed with other data pages of heap. Table 2 summarized the number of code and data pages before and after applying SGX-LEGO. SGX-LEGO reduces the overall number of pages required to execute the application.



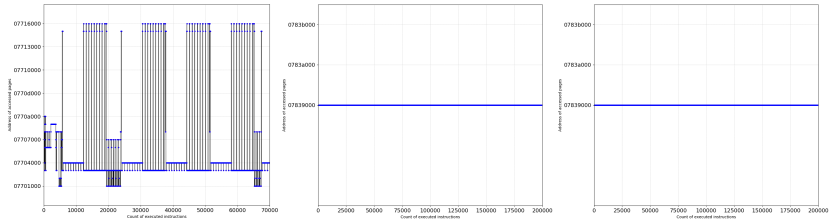
(a) AES-GCM



(b) AES-CTR



(c) SHA-256



(d) ECDSA (partial execution result is shown due to excessive amount of code execution)

Figure 14: SGX-LEGO evaluation results. The x-axis is the number of executed instructions and Y-axis is the address of accessed code page while executing the instruction. The left side image is the original pattern. The center image is after applying SGX-LEGO w/o gadget relocation process. The right side image is after applying SGX-LEGO w/ gadget relocation process. The codes are from Linux SGX SDK library.

After evaluating the security effectiveness of SGX-LEGO, we measured the performance overhead. We used Intel PIN tool [20] to accurately count the number of executed instructions before/after transforming the binary with SGX-LEGO. In average, SGX-LEGO increased the number of instructions approximately three times more than the original code. Table 3 is the result that shows increased numbers of AES-GCM, AES-CTR, SHA256, and ECDSA codes in Linux SGX SDK library[12] due to SGX-LEGO transformation. However, the overall performance overhead of SGX-LEGO showed approximately x16 overhead. Table 5 is the summary of SGX-LEGO performance overhead evaluation. The increased number of executed instructions were about x3. However the actual performance of SGX-LEGO shows about x16 increase execution in time. We further inspected the reason for such high overhead and find out that SGX-LEGO transformation reduced the CPU cache and pipeline utilization, thus decreased the Instruction Per Cycle (IPC) throughput.

Due to SGX-LEGO transformation, code cache hit rate significantly drops as the code turned into gadgets spread over multiple cache line. This is one of a major factor that affects SGX-LEGO performance [21]. Also, due to the massive use of branches (return), CPU pipeline cannot be fully utilized. In modern CPU pipeline architecture, out-of-order execution and parallelism maximize the IPC throughput thus `"IPC bigger than 1"` is commonly observed (Skylake shows IPC=5 in optimal condition). However, massive use of branch instructions inserted by SGX-LEGO hinders the out-of-order execution thus lowers the IPC throughput. As a result, SGX-LEGO imposes x10 to x20 execution slowdown in average Figure 15. Although there is high-performance overhead, we argue that SGX-LEGO is only required for a small portion of the entire application. SGX enclave is designed to place a security-critical portion of code such as cryptographic processing or security credential checks.

Overall, the evaluation suggests that newly emerging SGX side channel attack technique allows attackers to investigate enclave page access patterns

[12]The input message used for each algorithm is `"this is the plain text"`

Algorithm	w/o SGX-LEGO	w/ SGX-LEGO	Overhead
AES-GCM	192,042	858,627	x 4.471
AES-CTR	51,261	199,645	x 3.894
SHA256	21,316	54,960	x 2.570
ECDSA	99,967,248	328,119,294	x 3.282

Table 3: Number of instructions before/after transforming the code with SGX-LEGO

Algorithm	w/o SGX-LEGO	w/ SGX-LEGO	Overhead
AES-GCM	110,019	1,756,419	x 15.96
AES-CTR	25,714	430,495	x 16.74
SHA256	9,316	192,364	x 20.88
ECDSA	56,770,592	806,265,302	x 14.20

Table 4: Performance impact of SGX-LEGO transformation. Numbers are the elapsed clock cycle count measured with RDTSC.

Algorithm	# of Instructions	# of Clocks	CPI	IPC
AES-GCM (original)	192,042	110,019	0.57	1.7
AES-CTR (original)	51,261	25,714	0.5	2.0
SHA256 (original)	21,316	9,213	0.43	2.3
ECDSA (original)	99,967,248	56,770,692	0.6	1.8
AES-GCM (SGX-LEGO)	858,627	1,756,419	2.0	0.5
AES-CTR (SGX-LEGO)	199,645	430,495	2.2	0.5
SHA256 (SGX-LEGO)	54,960	192,364	3.5	0.3
ECDSA (SGX-LEGO)	328,119,294	806,265,302	2.5	0.4

Table 5: Performance Overhead Analysis of SGX-LEGO. CPI stands for Cycles Per Instruction. IPC stands for Instructions Per Cycle.

with per-instruction granularity, and this makes our VID attack feasible and practical. Our evaluation regarding VID attack reports that attacker can reliably distinguish the running algorithm and its major library versions with code optimization level. In case the attacker can precisely collect the page access patterns without any noises, she can further speculate the minor library version and detailed cryptographic parameters used by the algorithm. Although with high-performance overhead, SGX-LEGO effectively eliminates such side-channel

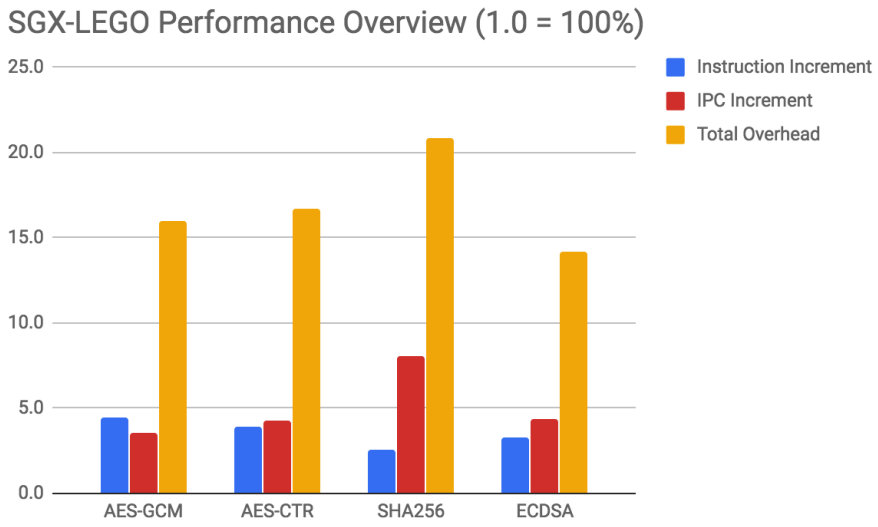


Figure 15: Performance Overview of SGX-LEGO

attacks under SGX spec1 environment.

7. Discussion and Limitation

7.1. Automatic ROP Payload Generation

ROP is a popular exploit technique used to bypass DEP. Automation of ROP is an old topic discussed by many previous works [7, 8, 10, 22, 9]. Schwartz et al. [7] presented *Q*, an automatic ROP payload generator for binary exploit. Their system takes in an attack target binary, and an exploit program written in a high-level language called QooL. Q then finds appropriate gadgets and makes ROP chain out of them. Since ASLR moves most of the code to randomized locations, their solution focuses on semantic analysis to extract gadgets from small amounts of unrandomized code. Similar tools [8, 10, 9] also try to find ROP exploits from given binary and attack specification. Some approaches [7, 22] aim to harden existing exploit code using discovered ROP gadgets.

Previous works for ROP automation is focused on attack. Thus ROP automation is aimed at executing a simple piece of attack code. The attack code is

typically written in either custom high-level language or real exploit, both of which are difficult to convert from existing program. In this paper, we made an advanced framework which is suited for complete execution of the compiled program in ROP style. Our framework does not require gadget discovery as we generate both gadgets and ROP payload. Rather, SGX-LEGO is more focused on generating minimal gadgets which can execute complicated logic such as recursions, and randomize their distributions to hide access patterns.

7.2. Branch Information Leakage Attack

According to other SGX side channel attack researches [23], *branch information leakage* is used as another side channel to retrieve the branch event history of the enclave. SGX-LEGO could be more resistant to such side channel attack as indirect branch instructions are transformed into stack pivoting operation which does not involve any branch instructions. For example, the indirect jump will be ultimately transformed into stack addition (or subtraction) operation; therefore, SGX-LEGO transformed binaries will reduce the original branch events.

7.3. Gadget Reusability

While converting the input binary into ROP-style gadgets, we found that the number of executed gadgets are heavily weighted towards a particular set of frequently used instructions. In detail, top 5% of gadgets have more than 50% portion of executed instructions. The most frequently used instructions are branch instructions such as `CALL` or `JMP`. Second frequently used instructions are `register to memory, memory to register` data transfer instructions. Third frequently used operations are arithmetic operations such as `ADD`, or `SUB`. Other types of instructions were rarely used. SGX-LEGO leverages such statistical information to optimize the randomization of code page access. For example, placing duplicate instruction into multiple pages can be selectively adopted against frequently used instructions for reducing overall code page number.

7.4. Accuracy of VID

The discussion of this paper is based on the assumption in which the side channel attacker can count the number of executed instructions within the same page. This assumption stems from the technique is shown by [3, 11]. However, the accuracy of such instruction counting could be unreliable depending on the instruction types. Therefore the VID attack discussed in this paper does not guarantee the identification with absolute certainty. The comparison between patterns and inference provides probabilistic information based on the pattern similarity, which is a limitation of the VID attack.

7.5. Optimization

Compiler optimization affects the code emission process thus can change the code page access pattern. In our evaluation, we also considered various optimizations for different builds. We observed significant changes in code page access patterns in case the application was small. In case the application code base was big, the impact of code optimization to page access pattern was relatively small. We also considered the link-time optimization affecting the code page access patterns. To evaluated this by changing the Visual Studio build options (`/LTCG, /GL` flags) of Windows SGX SDK. However, we could not measure any difference from the execution pattern.

7.6. VM Obfuscation

Virtual Machine based obfuscation (VM-obfuscation) is one of an effective obfuscation technique that diverts static and dynamic analysis. To obfuscate the binary, VM-obfuscation splits the program logic into small pieces and implement them as a virtual function. With such virtual functions, VM-obfuscation creates an additional software execution layer with bytecodes which will tell the VM how to stitch the functions and execute the binary. This concept can be easily observed in the commercial software market (e.g., Java, ActionScript, Python, etc.) for providing better compatibility with underlying platforms. Such VM-based programming languages provide the bytecode specification thus make it

easy to analyze the binary. Defining a custom bytecode instruction set and hiding its binding information with virtual functions makes the binary difficult to analyze; which is the purpose of VM-obfuscation. From a high-level view, the basic concept of SGX-LEGO is similar to VM-obfuscation technique. However, the difference is that SGX-LEGO mainly focuses its design on achieving side-channel resilient primitive against instruction granularity fault monitoring rather than dynamic analysis with debuggers.

8. Related Works

Since the introduction of Intel SGX in 2013 [24, 25, 26], various defensive researches have been proposed based on its security feature [27, 4, 28, 29, 30, 31, 32, 33]. Haven [27] uses SGX technology to secure the applications from an untrusted cloud environment. Recently, Microsoft utilizes SGX for enterprise blockchain service in cloud environment [34]. On the other hand, offensive researches against SGX as also been explored. According to the attack model of SGX [6, 35], an attacker has highest software privilege such as operating system. Due to such a powerful attack model, various attacks have been studied based on memory mapping manipulation and cache timing based side-channel attacks and so forth. In this section, we discuss various related works regarding SGX attack and defense.

8.1. Attacks against SGX

Memory Mapping Attacks. Controlled-channel attack [1] (introduced in 2015) is a typical example of memory mapping manipulation attack. Controlled-channel attack leverages intentional page-faults caused by an attacker to get page fault sequence information. Later the information is analyzed with code information, and ultimately, attacker infers the data inside enclave memory which caused such faulting sequence.

Xu al el. Demonstrated that controlled-channel attack can partially reveal information of *Font library (FreeType), Spell checker (Hunspell) and JPEG*

encoder (libjpeg) inside the enclave. Shinde al el. [2] demonstrated *pigeonhole* attack which makes OS allocate maximum three pages for SGX based on memory mapping manipulation. Using such attack, the paper various cryptographic keys for AES, EdDSA, RSA from Libgcrtpy and OpenSSL.

Van Bluck et al. [3, 11] proved that page access information of enclave could be inferred even without causing page-fault. The attack is based on the observation of page table entry (PTE) status such as Access or Dirty bits and cache flushing techniques. Once the victim enclave enters (EENTER) to the enclave, page read/write attempt can be inferred by monitoring the change of PTE. Using Inter-Processor Interrupt (IPI), the memory access attempts can be identified instantly. Van Bluck et al. [3, 11] also shows cache timing attack. To handle the repetitive access to same enclave page, attack uses IPI signal with a high frequency and checks the page table and its cache contents to capture the memory access using TLB. When AEX event occurs, TLB entries for the enclave addresses are cleared. However, the data cache of the corresponding PTE remains. Leveraging this, attacker uses Flush+Reload [36] or Flush+Flush [37] attack. Based on the flushing execution time, an attacker can distinguish the specific PTE that is used for page access (high execution time signifies the page is accessed). VID attack discussed in this paper is based on this technique, and SGX-LEGO is proposed to reduce the number of code pages and eliminate discernable access patterns using ROP-style execution and randomness.

Cache Timing Attacks. Cache timing attack is based on timing difference of main memory access and CPU cache access. SGX prevents such attack by restricting EPC memory from non-enclave code, yet some collision of cache set due to memory encryption engine (MEE) [38] remains. However, due to Anti Side-Channel Interference (ASCI), Performance Monitoring Counter (PMC) does not record any hardware events occurred inside enclave such as cache hit/miss. Since there is no shared memory between enclaves, and `clflush` is not supported to enclave memory, Flush+Reload [36] attack is supposedly infeasible. However, due to some inevitable conflict between attacker's enclave and victim's enclave cache, various cache timing attacks are still being proposed by researchers.

Brasser et al. [39] proposed Prime+Probe [40] attack which makes victim enclave process and attacker's process to use dedicated core and share probes cache line. To check cache line eviction of victim enclave, the attack uses PMC. As a result, private key information was leaked during RSA decryption (Chinese Remainder Theorem implementation). Similarly, sensitive information of genome analysis was exposed due to this attack.

Schwarz et al. [41] demonstrated that side channel attack is possible even enclave process and attacker's process uses different CPU core. Prime+Probe [40] attack exploits that RSA signing operation uses fixed buffer while multiplication. Attack reveals RSA private key from Square-and-multiply exponentiation (constant-time Montgomery implementation) of the mbedTLS cryptographic library.

Götzfried al el. [42] combined Prime+Probe [40] attack and PMC to extract key of AES algorithm (Gladman implementation). The attacker, in this case, shares the same process as victim enclave yet uses a different thread.

Branch Prediction Attack. Lee at el. [23] introduced a side-channel attack based on SGX enclave using branch history obtained from Last Branch Record (LBR) register. Due to ASCI, enclave does not keep any information of LBR. However, Branch Target Buffer (BTB) address information is allowed to be shared between the enclave and outside enclave which allows branch shadowing. Ultimately attacker obtains branch address history from outside of enclave.

Attacker places shadow code outside enclave and executes target enclave using the leaked branch address. Based on branch history, the execution speed of branch code is affected. Thus an attacker can tell if the branch was taken or not. Using local advanced programmable interrupt controller (APIC) timer, an attacker monitors the precise events. As BTB is part of the processor, fixing this issue is theoretically possible but not trivial. Since SGX-LEGO uses ROP-style execution, every branch instructions are substituted with arithmetic stack operation (adding/subtracting stack pointer) and conditional moves. Therefore side-channel attack based on branch history leakage is infeasible in case SGX-LEGO is applied.

ROP Attacks. Lee at el. [43] showed that in case enclave code has software vulnerability (such as buffer overflow), ROP attack becomes feasible even attacker has zero knowledge of code information. Using brute-forcing, the attacker first locates some `POP` ROP gadgets which can manipulate specific registers and the location of `ENCLU` instruction. The attack uses different behavior of OS for handling illegal instruction and segmentation fault. Attacker determines if `EEXIT` instruction was executed by using the previously identified `POP` ROP gadgets and what register is affected by such gadgets. Lee at el. shows that using such gadgets, it is feasible to inject malicious code into RWX pages and leak the sensitive information inside enclave to the outside world. In this paper, we use the concept of ROP for defense (not attack), and we assume RWX page allocation is disallowed for the sake of security.

Scheduling Attack. Nico at el. [44] introduced a method of exploiting thread synchronization between SGX enclaves. Removing the page access permission of specific enclave page allows interrupting the execution flow of enclave code. Once such interrupt happens, another thread uses the object inside critical section thus causing race condition and ultimately hijack the execution flow. For example, if the enclave code has a use-after-free vulnerability, an attacker can remove the page of `free()` function and switch the scheduling to a different process and allow an attack.

8.2. Defenses against attacks

Page Multiplexing. Shinde al el. Introduced pigeonhole attack [2] and suggested defenses using the compiler-based technique. The key idea is to make the codes to have a balanced tree structure for processing data thus make page access pattern to be deterministic and be oblivious to page-fault. However, it is difficult to adopt balanced execution tree for the entire algorithm, and there are 4,000 times additional execution time. To reduce such huge overhead, the paper suggests H/W based method as well.

Oblivious Access Patterns. Oblivious RAM (ORAM) [12, 13, 14] is a general technique that encrypts and shuffles the memory access patterns to

eliminate access patterns inside untrusted storage. Ascend [45], Phantom [46], Raccoon [47] uses ORAM technique to hide memory address trace. Especially, Raccoon [47] and ZeroTrace [48] leverages ORAM to secure the SGX from side-channel attacks. On the other hand, Ohrimenko at el. [49] introduced a method using oblivious primitives and array accesses to prevent input data affecting the control flow. Using such method, machine learning process inside cloud environment is protected from side-channel attacks.

Transactional Synchronization Extensions (TSX). T-SGX [50] and Déjá Vu [51] uses Intel Transactional Synchronization Extensions (TSX) which uses hardware transactional memory to defend SGX side-channel attacks specifically Controlled-channel attack. Using TSX, all faults are handled as transaction abort, and OS cannot observe page fault events inside TSX transaction code. The problem is that if entire enclave code is wrapped with single TSX segment, the program is likely to hang due to normal fault events. Therefore T-SGX uses the concept of `spring board` to separate pages that have sensitive information and not. Proper segmentation optimizes the performance and effectiveness of T-SGX mechanism. T-SGX mitigates early version of controlled-channel attacks. However, newly proposed attacks [3, 52] are not prevented as they use different attack mechanism (access/dirty flag) to observe memory access.

Address Space Layout Randomization (ASLR). SGX-Shield [53] implements additional program loader inside SGX enclave. The newly added loader randomizes the binary with Randomization Units (RU) granularity which is smaller than pages. Each RU is loaded on the random address, and inter RU jumps are handled with additional branch instructions. Ultimately SGX-Shield adopts fine-grained ASLR inside SGX enclave. While implementing fine-grained ASLR, SGX-Shield uses writable code pages. To avoid opening another attack channels, SGX-Shield uses software DEP and Software-Fault Isolation (SFI). To distinguish code and data pages, Non-Readable and Writable (NRW) boundary are used. Using `r15` register, the boundary address is checked and sensitive memory such as State Save Area (SSA), and Thread Control Structure (TCS) is prevented from being accessed. In-enclave loader requires 586KB of additional

memory space to maintain metadata. Also, 64MB memory is pre-allocated to implement the SGX-Shield. SGX-LEGO does not require writable code page and uses default SGX loader.

Runtime Code Decryption. Verifiable Confidential Cloud Computing (VC3) [4] introduced a MapReduce framework which safely executes MapReduce operation from an untrusted cloud environment. Public enclave code inside cloud environment exchange key with a remote user and decrypts user data. However, to decrypt code at runtime, dynamic page permission change or RWX memory is required. The defense also performs runtime integrity check against memory permissions inside the enclave. SGX-LEGO does not require such page permission integrity checking as it does not need dynamic page permission feature from the first place.

9. Conclusion

In this paper, we discuss the security ramification of recently introduced technique [3, 11] that reduces the granularity of SGX controlled-channel attack. Based on the advanced attackers capability, we introduce *VID attack* which leaks the identity of code running inside SGX enclave. While evaluating the VID attack, we explored the feasibility of inferring exact version and the detailed configuration of running codes inside SGX enclave. After discussing *VID attack*, we introduce SGX-LEGO: binary conversion framework which eliminates deterministic page access patterns. The goal of SGX-LEGO is nurturing the VID attack and any pattern analysis based on the fine-grained SGX controlled-channel by applying polymorphism to its execution. SGX-LEGO leverages the concept of code-reuse-programming to overcome some challenges regarding SGX page management environment. SGX-LEGO is composed of 1,700 lines of C/C++ code and 1,100 lines of Python code. We evaluated the feasibility of VID attack and the effectiveness of SGX-LEGO with various cryptographic algorithms and libraries. Performance overhead induced by SGX-LEGO is x10 to x20 on average. However, it effectively eliminates discernable memory access patterns of code execution.

References

[1] Y. Xu, W. Cui, M. Peinado, Controlled-channel attacks: Deterministic side channels for untrusted operating systems, in: Security and Privacy (SP), 2015 IEEE Symposium on, IEEE, 2015, pp. 640–656.

[2] S. Shinde, Z. L. Chua, V. Narayanan, P. Saxena, Preventing page faults from telling your secrets, in: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ACM, 2016, pp. 317–328.

[3] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, R. Strackx, Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution, in: Proceedings of the 26th USENIX Security Symposium, USENIX Association, 2017.

[4] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, M. Russinovich, Vc3: Trustworthy data analytics in the cloud using sgx, in: Security and Privacy (SP), 2015 IEEE Symposium on, IEEE, 2015, pp. 38–54.

[5] T. Frassetto, D. Gens, C. Liebchen, A.-R. Sadeghi, Jitguard: Hardening just-in-time compilers with sgx, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 2405–2419.

[6] Intel, Isca 2015 tutorial slides for intel(r) sgx, revision 1.1 (June 2015).
URL `https://software.intel.com/sites/default/files/332680-002.pdf`

[7] E. J. Schwartz, T. Avgerinos, D. Brumley, Q: Exploit hardening made easy., in: USENIX Security Symposium, 2011, pp. 25–41.

[8] C. Team, Mona (2012).

[9] J. Stewart, V. Dedhia, Rop compiler.
URL `http://css.csail.mit.edu/6.858/2015/projects/je25365-ve25411.pdf`

[10] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, A.-R. Sadeghi, Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization, in: Security and Privacy (SP), 2013 IEEE Symposium on, IEEE, 2013, pp. 574–588.

[11] J. Van Bulck, F. Piessens, R. Strackx, Sgx-step: A practical attack framework for precise enclave execution control, in: Proceedings of the 2nd Workshop on System Software for Trusted Execution, ACM, 2017, p. 4.

[12] O. Goldreich, R. Ostrovsky, Software protection and simulation on oblivious rams, Journal of the ACM (JACM) 43 (3) (1996) 431–473.

[13] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, S. Devadas, Path oram: an extremely simple oblivious ram protocol, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM, 2013, pp. 299–310.

[14] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, S. Devadas, Constants count: Practical improvements to oblivious ram., in: USENIX Security Symposium, 2015, pp. 415–430.

[15] C. Eagle, The IDA pro book: the unofficial guide to the world's most popular disassembler, No Starch Press, 2011.

[16] V. Benony, Hopper disassembler (2017).
URL `https://www.hopperapp.com/`

[17] A. R. Bernat, B. P. Miller, Anywhere, any-time binary instrumentation, in: Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, ACM, 2011, pp. 9–16.

[18] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, G. Vigna, Ramblr: Making reassembly great again, in: Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS17), 2017.

[19] CODEMAP: Semantic run-trace visualization for binary analysis, `http://codemap.kr`.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: Acm sigplan notices, Vol. 40, ACM, 2005, pp. 190–200.

[21] M. Elsabagh, D. Barbará, D. Fleck, A. Stavrou, Detecting rop with statistical learning of program characteristics, in: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, ACM, 2017, pp. 219–226.

[22] L. Davi, A.-R. Sadeghi, D. Lehmann, F. Monrose, Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection., in: USENIX Security Symposium, Vol. 2014, 2014.

[23] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, M. Peinado, Inferring fine-grained control flow inside sgx enclaves with branch shadowing, USENIX Security Symposium.

[24] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, J. Del Cuvillo, Using innovative instructions to create trustworthy software solutions., in: HASP@ ISCA, 2013, p. 11.

[25] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, U. R. Savagaonkar, Innovative instructions and software model for isolated execution., in: HASP@ ISCA, 2013, p. 10.

[26] I. Anati, S. Gueron, S. Johnson, V. Scarlata, Innovative technology for cpu based attestation and sealing, in: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy, Vol. 13, 2013.

[27] A. Baumann, M. Peinado, G. Hunt, Shielding applications from an untrusted cloud with haven, ACM Transactions on Computer Systems (TOCS) 33 (3) (2015) 8.

[28] S. Kim, Y. Shin, J. Ha, T. Kim, D. Han, A first step towards leveraging commodity trusted execution environments for network applications, in: Proceedings of the 14th ACM Workshop on Hot Topics in Networks, ACM, 2015, p. 7.

[29] E. Bauman, Z. Lin, A case for protecting computer games with sgx, in: Proceedings of the 1st Workshop on System Software for Trusted Execution, ACM, 2016, p. 4.

[30] K. A. Küçük, A. Paverd, A. Martin, N. Asokan, A. Simpson, R. Ankele, Exploring the use of intel sgx for secure many-party applications, in: Proceedings of the 1st Workshop on System Software for Trusted Execution, ACM, 2016, p. 5.

[31] M.-W. Shih, M. Kumar, T. Kim, A. Gavrilovska, S-nfv: Securing nfv states by using sgx, in: Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, ACM, 2016, pp. 45–48.

[32] B. A. Fisch, D. Vinayagamurthy, D. Boneh, S. Gorbunov, Iron: Functional encryption using intel sgx.

[33] S. Eskandarian, M. Zaharia, An oblivious general-purpose sql database for the cloud, arXiv preprint arXiv:1710.00458.

[34] M. Moyen, Why intel will benefit from microsoft's blockchain-as-a-service. URL https://seekingalpha.com/article/4111178-intel-will-benefit-microsofts-blockchain-service

[35] V. Costan, S. Devadas, Intel sgx explained., IACR Cryptology ePrint Archive 2016 (2016) 86.

[36] Y. Yarom, K. Falkner, Flush+ reload: A high resolution, low noise, l3 cache side-channel attack., in: USENIX Security Symposium, 2014, pp. 719–732.

[37] D. Gruss, C. Maurice, K. Wagner, S. Mangard, Flush+ flush: a fast and stealthy cache attack, in: Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2016, pp. 279–299.

[38] S. Gueron, A memory encryption engine suitable for general purpose processors., IACR Cryptology ePrint Archive 2016 (2016) 204.

[39] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, A.-R. Sadeghi, Software grand exposure: Sgx cache attacks are practical, WOOT.

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

[41] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, S. Mangard, Malware guard extension: Using sgx to conceal cache attacks, DIMVA.

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

[43] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, B. B. Kang, Hacking in darkness: Return-oriented programming against secure enclaves, in: USENIX Security, 2017.

[44] N. Weichbrodt, A. Kurmus, P. Pietzuch, R. Kapitza, Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves, in: European Symposium on Research in Computer Security, Springer, 2016, pp. 440–457.

[45] C. W. Fletcher, M. v. Dijk, S. Devadas, A secure processor architecture for encrypted computation on untrusted programs, in: Proceedings of the seventh ACM workshop on Scalable trusted computing, ACM, 2012, pp. 3–8.

[46] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, D. Song, Phantom: Practical oblivious computation in a secure processor, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM, 2013, pp. 311–324.

[47] A. Rane, C. Lin, M. Tiwari, Raccoon: Closing digital side-channels through obfuscated execution., in: USENIX Security Symposium, 2015, pp. 431–446.

[48] S. Sasy, S. Gorbunov, C. Fletcher, Zerotrace: Oblivious memory primitives from intel sgx, IACR Cryptology Archive Report 549 (2017) 2017.

[49] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, M. Costa, Oblivious multi-party machine learning on trusted processors., in: USENIX Security Symposium, 2016, pp. 619–636.

[50] M.-W. Shih, S. Lee, T. Kim, M. Peinado, T-sgx: Eradicating controlled-channel attacks against enclave programs, in: ISOC Network and Distributed System Security Symposium, 2017.

[51] S. Chen, X. Zhang, M. K. Reiter, Y. Zhang, Detecting privileged side-channel attacks in shielded execution with déjá vu, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ACM, 2017, pp. 7–18.

[52] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, C. A. Gunter, Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 2421–2434.

[53] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, 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, 2017.