Available online at www.sciencedirect.com

ScienceDirect

Forensic Science International: Digital Investigation 39 (2021) 1–17

FSI: Digital Investigation

The evidence beyond the wall: memory forensics in SGX environments

Flavio Toffalini[a,*], Andrea Oliveri[b], Mariano Graziano[c], Jianying Zhou[a], Davide Balzarotti[b]

[a] *Singapore University of Technology and Design, Singapore*
[b] *Eurecom, Sophia Antipolis, France*
[c] *Cisco Systems, Inc., USA*

Abstract

Software Guard eXtensions (SGX) is a hardware-based technology that introduces unobservable portions of memory, called enclaves, that physically screens software components from system tampering. Enclaves can be used to run arbitrary programs (including malicious code), but their actual impact on digital forensics and incident response remains unknown. In our work, we propose a methodical study of what information can be retrieved from an SGX machine and how to use this information to infer the enclaves interfaces and structure layout.

We tested our techniques over a dataset of 45 SGX applications and we showed the practicality of our techniques in a real-product use-case and on two malware-enclaves.

Keywords: SGX, TEE, memory forensics

1. Introduction

Software Guard eXtensions (SGX) is a technology developed by Intel that allows developers to define secure memory regions, called *enclaves*, to protect critical pieces of code and data [1]. SGX assumes a zero-trust environment, in which the whole system (including the Operating System, the SMM, and the peripherals) can be compromised and the CPU itself is the only trusted element. Specifically, SGX allows to run code while preventing any other part of the system from accessing the *enclave* memory space.

Developers have used this technology for DRM protections [2] and to enhance security properties of remote systems deployed in untrusted environments [3] (*e.g.,* virtual machines in cloud environments). Several development frameworks have also been proposed, by both industry and academia [4, 5, 6, 7, 8, 9], to simplify communication between the software running in the enclave and the external world.

However, the strong isolation provided by the SGX technology is a double-edged sword — protecting legitimate code from untrusted environments, but also preventing security tools from inspecting the memory and performing forensic investigations. Both researchers and malware developers have also investigated how to exploit SGX for

*Corresponding author

Email addresses: flavio_toffalini@mymail.sutd.edu.sg (Flavio Toffalini), oliveri@eurecom.fr (Andrea Oliveri), magrazia@cisco.com (Mariano Graziano), jianying_zhou@sutd.edu.sg (Jianying Zhou), davide.balzarotti@eurecom.fr (Davide Balzarotti)

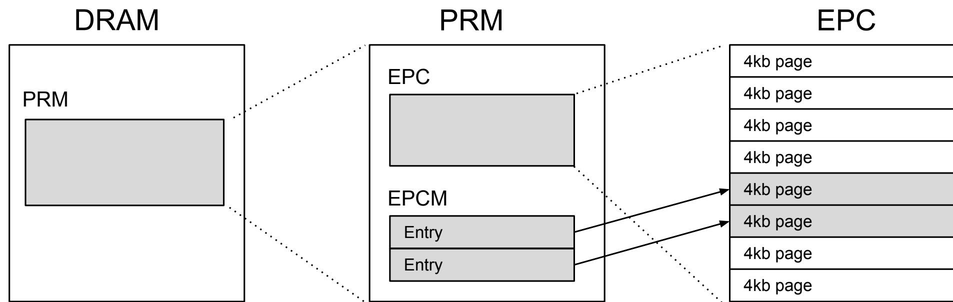

Figure 1: Enclave pages architecture in DRAM.

malicious purposes [10, 11, 12, 13, 14]. At the same time, a dump technique based on Intel DCI interface ables to access to Intel SGX enclaves memory has been recently proposed by *Latzo et al.* [15]. However, this technique requires specialized hardware and an unlocked Intel DCI interface on the machine which, in general, are not available.

Even though code running in an enclave cannot be retrieved and analyzed, other artifacts may be present in unprotected memory, thus allowing an investigator to infer certain properties of what is running in the SGX-protected space. In fact, an enclave cannot be completely self-contained, as it always needs some external support code to interact with the rest of the environment. However, to the best of our knowledge, no study has been performed to date on the consequences of SGX on memory forensics.

In this work, we take the role of a forensic analyst who is analyzing a copy of the physical memory acquired on an SGX-enabled machine. Many different variables can affect the analysis, including the way the memory image was acquired, the kernel drivers used to assist SGX operations, or the use of a certain SGX development framework. In this polyhedric scenario, we want to provide a clear guideline that helps a forensic analyst untangle the possible cases in an SGX-machine inspection. Our study shows which artifacts can be extracted in each case and discusses sound methodologies to extract those artifacts.

We evaluate our techniques in two different environments (a bare-metal machine and a VM in the Azure cloud) by using a set of 45 SGX applications taken from most used open-source projects, accademic works, a commercial SGX development framework [16], and two state-of-the-art examples of malicious enclaves [12, 13]. In all scenarios, our system was able to correctly detect and find the *enclaves*, retrieve information about the internal *enclave* architecture, and identify their system interfaces (*e.g.,* network communications or file-system access). Furthermore, we discuss three practical use cases in which our techniques can support an analyst to dissect a real application and two malicious *enclaves*.

Contribution – In summary, we make the following contributions:

- We discuss the limitation of current forensic analysis in SGX-machines.
- We methodically study which information about SGX is available to an analyst at each level of the forensics process and in each component of the system (*i.e.,* directly from hardware, kernel and processes) by proposing new techniques to support SGX *enclaves* discovery and analysis (§3).
- We evaluate our techniques on both open- and close-source projects, and malware-enclave samples (§4). We release our open-source proof-of-concept tools, along with a guide to replicate our experiments, at `https://github.com/tregua87/sgx-forensic`.

2. Background

In this section, we introduce the Software Guard eXtension technology (§2.1) and the development frameworks for SGX (§2.2).

2.1. Software Guard eXtension

The core concept of Software Guard eXtension (SGX) [1] are *Enclaves*: restricted memory regions allocated in DRAM. Figure 1 shows the SGX memory architecture where a subset of DRAM, called Processor Reserved Memory (PRM), is dedicated to the CPU itself (*i.e.,* the microcode) and is shielded from the other system components, such as the operating system, the SMM code [17], and DMA transfers [18]. The PRM includes an Enclave Page Cache (EPC), which physically contains the *enclave* pages. Moreover, since SGX assumes the OS is untrusted, the PRM also contains a specific structure, called Enclave Page Cache Map (EPCM), that keeps trace of the *enclave* page status. The EPCM assists the microcode in performing further security controls to block cross-*enclave* memory access or multiple *enclave* page allocations.

The SGX specifications define four types of EPC pages:

1. SGX Enclave Control Structure (SECS) contains the *enclave* metadata. The SECS is used by the CPU as a unique *enclave* identifier and is the only page not mapped in user space, as only the CPU microcode is allowed to read and write it.
2. The Thread Control Structure (TCS) contains information about a trusted thread [19] running on the *enclave*, pointers to its stack, and other internal status structures.
3. State Save Area (SSA) handles exceptions generated from within an *enclave*.
4. Regular pages (REG) contain code and data.

An *enclave* is bootstrapped by using code running in kernel mode (*i.e.,* a kernel driver) and a set of SGX specific functionalities, triggered by two dedicated CPU instructions: `ENCLS` (executable only in kernel mode) and `ENCLU` (executable instead in user mode). The kernel handles page allocation and eviction. Any evicted page is encrypted and versioned by the microcode before being stored in non-volatile storage. In addition, the microcode performs a double-check to validate the correctness of the pages against a cryptographic hash [20] stored in SECS to prevent replay attacks from the (untrusted) kernel.

An *enclave* can be bootstrapped in debug mode or in release mode. The debug mode is used during the development of the *enclave* and permits external software (*e.g.,* debuggers) to access the *enclave* memory pages through dedicated opcodes (*i.e.,* `EDBGRD` and `EDBGWR` [21]). Release mode, instead, enforces all the SGX security features and does not permit to read the content of an *enclave* from the system. However, to run an enclave in release mode, a developer must request a key issued by Intel. This fact, as shown in [22], incentives malware developers to release enclaves in debug mode which does not publicly exposing own identity (because does not require signatures as release mode enclaves) and guarantees an undetectability from ordinary anti-malware tools (through the impossibility to access to the enclave memory using ordinary syscalls).

SGX also introduces an attestation mechanism [23] to issue a secure communication channel between two *enclaves*. The attestation can be local (on the same machine) or remote (involving remote machines). In particular, the SGX remote attestation guarantees a remote third-party to verify the integrity of a local *enclave* and the SGX platform running it. In SGX machines, this process is based on special privileged enclaves, called *quoting* and *provisioning* enclaves, that are issued by Intel. These enclaves produce a cryptographic proof that validates the microcode version, the CPU unique keys, and the content of the enclave to be attested. The result of this cryptographic process is sent to an Intel attestation server which replies with an asymmetric signature. The signature can be then sent over the network to the remote third-party to prove the identity and integrity of the enclave and the platform [23].

The process that contains an *enclave* is called *host process*, and interacts with the *enclave* through the `ENCLU` instruction. The reserved part of the virtual address space of the *host process* designated to its *enclave* is called ELRANGE. The ELRANGE contains all the *enclave* pages and is defined by a base address and its length. An *enclave* assumes all the code located inside its ELRANGE to be trusted, while everything outside is untrusted and possibly under control of an attacker.

To help the developers build more sophisticated SGX-based applications, both Intel, third-party researchers, and companies, have released a number of development frameworks that we discuss in the next section.

Table 1: SGX frameworks used in 84 open-source projects from [24, 25].

Framework	Category	Projects
Intel SGX SDK	API-like	72
Open Enclave SDK	API-like	4
Asylo	API-like	4
RUST-SDK	API-like	–
SGX-LKL	Container-like	–
Graphene	Container-like	4

2.2. SGX Development Frameworks

A host process that desires to interact with an *enclave* needs to include specific portions of code, namely *enclave interface*, deputed to communicating with the OS. In particular, an enclaves interface is mainly composed on two type of functions: *secure functions* (*ecall*) and *outside functions* (*ocall*). The host process uses the former to invoke code inside an enclave, while the enclave uses the latter to communicate with the operating system (*e.g.,* to invoke system calls). Moreover, both functions invoke an `ENCLU` opcode directly or indirectly (*i.e.,* through a utility function). Due to the complexity of this programming pattern, a number of development frameworks have been developed to abstract the technical details and automatically include components both in- and outside the *enclave*.

All development frameworks share some architecture details. In particular, parts of *secure* and *outside functions* are contained in a specific portion of code called *edge*, that is automatically generated and statically linked by the development framework. In our study, we found that the *edge* can either rely on external libraries (*e.g.,* `libsgx_urts.so`) to communicate with the enclave, or it can embed the entire communication logic (including the `ENCLU` invocations).

For our study we analyze the four frameworks that are used by the 84 SGX applications available on two public hubs of SGX open-source projects [24, 25] (results are summarized in Table 1). In addition, we decided to also include two additional frameworks: SGX-LKL, which was used in a recent paper [26], and RUST-SDK, an example of a RUST technology applied over SGX [9].

We divided the resulting six development frameworks into two categories.

API-like – These frameworks organize the *enclave* code as a set of *secure functions*. Application simply invokes any secure function on-demand and retrieves its result. This is the case of the Intel SGX SDK [4], which was the first framework to be published. It allows to define *secure functions* within an *enclave*, as well as to handle *outside functions* and exceptions. Open Enclave SDK [6] has been developed by Microsoft, it is built on top of the Intel SGX SDK and permits to makes the enclave code portable to other secure computing technologies such as TrustZone. Google released Asylo [7] which is built on top the Intel SGX SDK and, similarly to Open Enclave, aims to transparently port the *enclave* code to different technologies. Finally, RUST-SGX [9] expands the Intel SGX SDK to include Rust code within an enclave, thus mitigating possible memory corruption errors inside the enclave itself.

Container-like – These frameworks use the *enclave* as a container to host an entire application, similarly to a Virtual Machine or a Docker Container. In these cases, the *secure function* is mainly used to start the contained application. SGX-LKL [8] is based on the Open Enclave SDK, it can load unmodified binaries inside an SGX *enclave* in Linux environment. Graphene [5] is a research project to load unmodified binary within an *enclave*. It is also the only framework that, besides the driver itself, does not share any other components with the Intel SGX SDK.

3. Memory Forensics in SGX environment

In this section, we tackle the various scenarios an analyst can face while inspecting an SGX-machine. In the first part, we define our threat model (§3.1). In the second part, we focus on our study, that we split in three main phases: memory acquisition (§3.2), kernel space analysis (§3.3), and user space analysis (§3.4). For each phase, we discuss the possible conditions that an analyst can encounter, propose new analysis methodologies, and describe which information can be gathered from the system. To clarify the different possibilities and provide a useful support for the

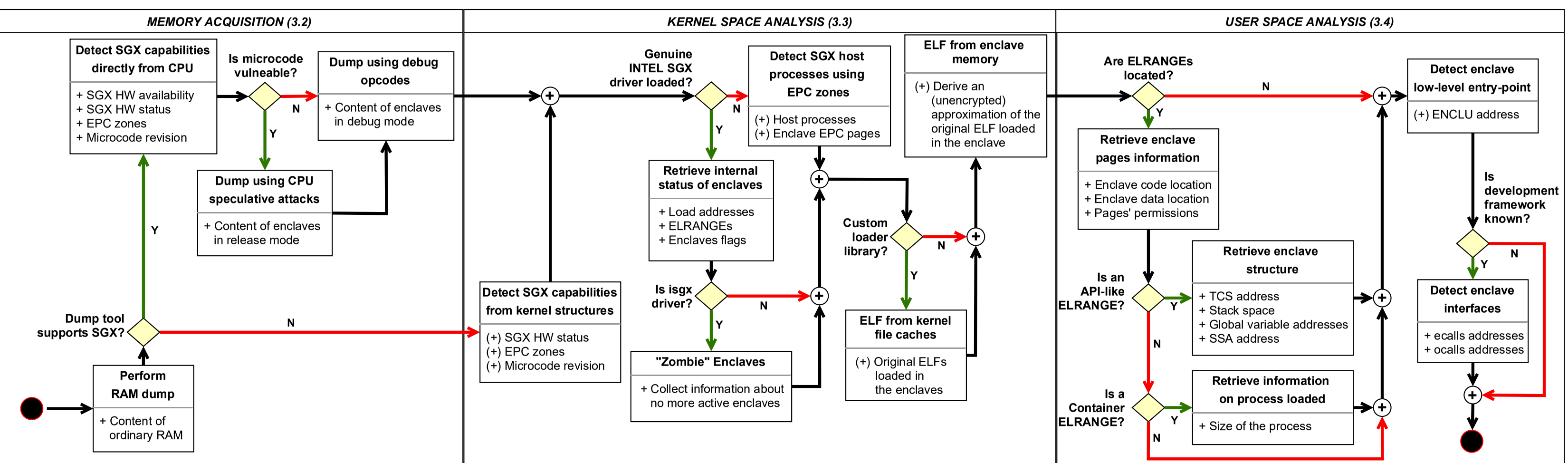

Figure 2: The flowchart describes the workflow of an SGX-machine inspection and depicts the three main phases: memory acquisition, kernel space, and user space analysis. The diamonds represent different conditions that can be encountered (*e.g., Is microcode vulnerable?*), each condition leads to different class of information (*e.g.,* if the microcode is vulnerable, we can *Dump using CPU speculative attacks*). Green-arrows with symbol "Y" indicate the available information if a condition is satisfied, while red-arrows with symbol "N" otherwise. In addition, every box contains detailed pieces of information that we mark with the symbol "+" if they are always retrievable, while we use the symbol "(+)" if their availability depends by the system state (*e.g.,* a system log that might be wiped). The technical details of each phase is detailed in their relative section.

analyst, we summarize the SGX forensics process into the flowchart in Figure 2. The image contains all the scenarios discussed in our study and can serve as a high-level guide for an SGX-machine inspection. The map distinguishes, left to right, the three main phases described above (*i.e.,* memory acquisition, kernel-, and user-space analysis). Each diamond identifies a different condition (*e.g., Dump tool support SGX?*), while boxes synthesize the action that can be performed (*e.g., Detect SGX capabilities from kernel structures*) and the fine-grain information that can be recovered (*e.g., SGX HW availability*). In this case, we use the symbol "+" to refer to a piece of information always present, and the symbol "(+)" if the information might be present (*e.g.,* a system log that might be wiped). It is important to note that each type of information represented in Figure 2 is useful to the human analyst to have a complete view of the machine and gives details about the state of the SGX enclaves, their possible interaction with the system and gives hints to perform more analysis on those (e.g. static code/data analysis of the SGX enclave content). On top of that, some pieces of evidence here presented are prone to be lost if not correctly acquired during the dump phase causing irreparable damage to the forensics survey. The details about each phase are described in the next sections.

3.1. Threat Model

In this work, we assume a machine that supports the SGX technology. The systems may contain an indefinite number of enclaves (possibly none). Any running enclave is correctly loaded in memory and isolated from the other parts of the system thanks to SGX. The purpose of the analyst is to gather as much information as possible about the SGX configuration and the running enclaves.

Overall, the system configuration may assume many combinations. Here, we depict the two main coarse grain scenarios:

- In the first case, we assume that the enclave is operated by using a known OS driver to handle the enclave pages and (for the user space analysis) the application was developed by using a known SGX framework.
- In the second more challenging case, we remove these assumptions and consider the case in which the analyst does not know the SGX drivers and has no information about the adopted development framework.

In Figure 2, we detail a fine grain combination of the previous system setting.

3.2. Memory Acquisition

As explained in Section 2, each SGX enclave is composed of a set of encrypted physical pages which reside in an EPC memory zone. The CPU exposes the EPC memory zones as special MMIO devices not as part of the main memory of the system. For this reason, the OS does not include them in data structures that map all the ordinary memory regions available. However, the kernel is able to map EPC pages in the virtual-address space even if a direct read of their content returns a constant value (*i.e.,* `0xFF`) due to the SGX secrecy protections. Only when an enclave is running in debug mode, a kernel thread can dump the plain content of the enclave memory by using the special opcode `EDBGRD` as illustrated in [21]. Ordinary dump tools, which relies on the operating system data-structures to identify available zones of the memory containing data, fail to find and dump the EPC zones. An SGX awared dump tool can overcome this limitation by querying the CPU using the `CPUID` instruction [19], which permits to retrieve various configuration parameters including if SGX is present and enabled, all the EPC zones defined on the system, their physical addresses, and sizes and the microcode revision. It is important to note that this query to the CPU is based only on opcodes and does not affects in any way the content of RAM, its correctness, and coherency. Previous works that investigated malicious applications of SGX argued that, in practice, an adversary is incentivized to use enclaves in debug mode for deploying malware [22]. This is because compiling an enclave in release mode requires to contact Intel, thus publicly exposing own identity (more info in §2.1). Therefore, we have developed a dump tool which localizes the EPC zones, independently by the OS data-structures, marking them in the dump, and, using `EDBGRD` opcode, saves pages related to debug-mode enclaves, without any information on the number and exact location of them. This also permits to dump those pages associated to past debug-enabled enclaves, which are not associated to a host process anymore.

CPU speculative attacks as forensics pinlock – Enclaves that run in release mode continue to be blind spots from the analyst point of view. A possible way to dump the content of non-debug enclaves and gaining a complete vision of the system status is to exploit CPU speculative attacks. These classes of attacks, as summarized in [27], allow an

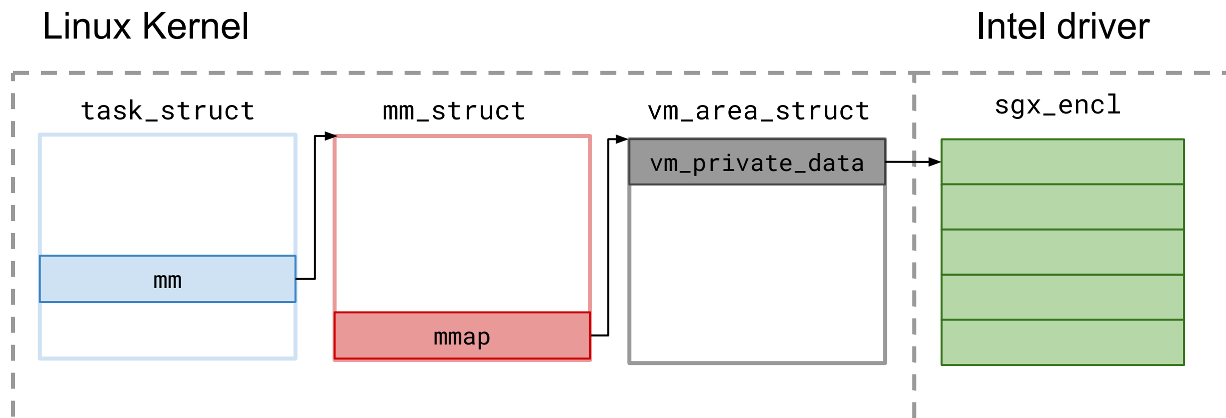

Figure 3: Relation between the kernel and Intel drivers data-structures

attacker to dump the content of an enclave even if it is not in debug mode. In particular, attacks like Foreshadow-SGX [28] and SGAxe [29] permits to recover the content of the enclave with high accuracy in a finite time and without any knowledge of their internals. It is important to note that all these attacks rely on CPU hardware bugs which are quickly fixed by Intel through patched microcode updates. These updates are fundamental to run enclaves which require remote attestation because in this case attestation servers checks the microcode revision and only attests the enclave if it is running on a fully-patched CPU. On the other hand, patched microcode must be applied by the final users or released as BIOS updates by motherboard producers and often require a machine reboot in order to be enabled. System administrators of mission-critical servers, non-expert users and unmanaged systems could delay the installation of patched microcode thus permitting in certain cases to an SGX awared dump tool to exploit them to dump also non-debug enclaves. In this case, the SGX-aware dump tool can retrieve the microcode revision of the CPU by using the `CPUID` opcode, and if it is vulnerable to speculative attacks it can extract the content of enclaves running also in release mode.

3.3. Kernel space analysis

If the dump tool is not SGX-aware, no additional information about the SGX configuration is collected from the machine at the acquisition time. However, as shown in Figure 2, an analyst can deduct these pieces of information by analyzing kernel data structures. For example, it is possible to retrieve EPC zones by looking at the `iomem_resource` data structure, that contains MMIO mapped devices (as shown in `/proc/iomem` on live systems). It is also possible to extract logs lines from kernel logging facility (`dmesg`) in-memory buffer. However, both data sources are not always available: early log entries can be overwritten if the machine has a long uptime and some BIOS vendor does not export EPC zones addresses as MMIO memory regions, making them unavailable in `iomem_resource`.

When the status of the SGX hardware on the machine is not available, any further analysis is more difficult and can result in a larger number of false positives.

As explained in 2.1, SGX enclaves rely on the kernel to manage memory allocations: the kernel traces all pages allocated to the enclaves, assigns new ones when required, and evicts them when it no longer needs them. Furthermore, the kernel traces all the enclaves associated with a process and maps the enclaves' pages into the virtual address space of their host process. Non-monolithic kernels, as Linux and Windows, use specific drivers to manage enclaves. These drivers permit the user space enclave loader to the load the enclave pages inside the EPC memory. In particular, on Linux, there are two open-source drivers, developed by Intel, `isgx` [30] and `DCAP` [31], which share very similar structure and functionalities. From a forensic perspective, the need of keeping track of the enclaves status by the kernel allows to recover information on the enclaves present in a memory dump by analyzing the data-structures allocated by the kernel modules.

Figure 3 shows the Intel drivers structures that handle the enclaves pages. The drivers allocate private data-structures under the `vm_private_data` space of the enclave's host process, which is contained in an `vm_area_struct`.

The vm_private_data contains information about the ELRANGE. The sgx_encl structures maintain: (i) the enclave load address, (ii) a list of the EPC pages allocated, (iii) the status flags of the enclave (*e.g.,* flags which indicate if an enclave is in debug mode, has performed cryptographic operations, the CPU capabilities required etc.) and, (iv) a linked list to the other enclaves instantiated on the system (this only for the isgx driver) which permits also to recover "zombie" enclaves (*i.e.,* enclaves detached by the host process but not yet deallocated).

In the case in which the kernel uses a module unknown to the analyst, it is still possible to rely on heuristics to determine the presence of an enclave in a process. In fact, the enclaves pages have to reside in an EPC memory zone that is exposed by the CPU as a separate hardware device, and the driver must treat it accordingly. For example, on Linux systems, memory pages associated with MMIO devices, like SGX, must be marked as VM_PFNMAP and VM_IO, which allows the kernel to treat them in a special way with respect to ordinary memory operations. Therefore, to detect a process that hosts an enclave, an analyst can check if it has memory pages flagged as part of special memory zones without using any knowledge about the internal of the driver. Furthermore, if the dump was performed by using an SGX-aware tool, it is also possible to check if the candidate pages reside in an EPC zone to reduce false positives (*e.g.,* processes which map common MMIO devices as graphic cards). However, this driver-agnostic approach comes at a cost: it is not possible to determine the enclave base address and its flags. These information are stored in the SECS page which is not accessible, and can be recovered only if a copy is maintained into known driver data structures.

The code of an SGX enclave is distributed as dynamic library file (.dll file under Windows and .so file under Linux). However, in contrast to ordinary dynamic libraries, the enclave code is not loaded by the binary loader of the system but it is loaded inside the memory enclave by an auxiliary dynamic library. This process could leave, in theory, various pieces of evidence in the kernel data structures. For instance, caches associated to the file descriptor of the loading library could contain traces of the enclave code file. However, the default library deployed by Intel does not leave any trace of the original ELF loaded in the kernel. Furthermore, the security implementation of SGX offers a "solution" to the possible information leaked due to the enclave loading (reducing the evidence from a forensics point of view): the code file containing an enclave in release mode can be encrypted during the build, loaded inside the enclave and decrypted only inside the protected memory [32].

However, if we have performed a dump with an SGX-aware tool, we can reconstruct the ELF file loaded into the enclave starting from the content of the enclave itself. This permits, if the dump also contains pages of enclaves in release mode, to bypass the encryption technique used by the author to hide the content of the ELF file loaded since we extract the decrypted version of the ELF directly from the enclave memory.

3.4. User Space Analysis

Since enclaves need a user-space component to interact with the system, an analyst can gather information by analyzing structures that reside in the untrusted memory of user-space processes. In particular, we discuss the recovery of the enclave memory layout (§3.4.1) and the enclave interface (§3.4.2).

3.4.1. Enclave Memory Layout Analysis

It is possible to infer the enclave memory layout by inspecting the page permissions of its ELRANGE. The Intel documentation proposes a well-defined internal layout that helps defining multi-thread enclaves and is thus adopted by all the API-like frameworks we analyzed. However, the enclave memory layout is quite flexible, as in the case of the Container-like frameworks. We implemented an algorithm that is able to analyze the enclave memory layout and automatically infers if the framework is an API-like, a Container-like, or none of them. In Figure 4, we exemplify an API-like and a Container-like framework. Below, we describe what information can be gathered in the two scenarios and we discuss what can be done when the enclave does not follow any known structure.

API Like – All the API-like frameworks that we studied follow the standard Intel SGX SDK layout, which is exemplified in Figure 4a. This layout does not have rwx pages by default and it expects as first pages respectively the enclave header, the code, constants, and the global objects (*e.g.,* BSS or heap data). The rest of the memory is dedicated to the trusted threads [19]. In particular, a trusted thread is composed of four parts: (i) the stack space, (ii) the Thread Control Structure (TCS), (iii) a fixed number of State State Area (SSA), and (iv) the Thread-Local Storage (TLS). The same structure is repeated for each trusted thread.

Label	Start add.	End add.	Size	Perm.
HEADER	0x7ffff6a00000 -	0x7ffff6a01000	(0x1000)	r--
CODE	0x7ffff6a01000 -	0x7ffff6a1f000	(0x1e000)	r-x
CONSTANT	0x7ffff6a1f000 -	0x7ffff6a26000	(0x7000)	r--
	0x7ffff6a26000 -	0x7ffff6a27000	(0x1000)	---
GLOBAL OBJECTS	0x7ffff6a27000 -	0x7ffff6b38000	(0x111000)	rw-
	0x7ffff6b38000 -	0x7ffff6b48000	(0x10000)	---
STACK	0x7ffff6b48000 -	0x7ffff6b88000	(0x40000)	rw-
	0x7ffff6b88000 -	0x7ffff6b98000	(0x10000)	---
repeat TCS + SSA	0x7ffff6b98000 -	0x7ffff6b9b000	(0x3000)	rw-
	0x7ffff6b9b000 -	0x7ffff6bab000	(0x10000)	---
TLS	0x7ffff6bab000 -	0x7ffff6bac000	(0x1000)	rw-
	0x7ffff6bac000 -	0x7ffff6c00000	(0x54000)	---

(a) Example of Memory layout of Intel SGX SDK.

Label	Start add.	End add.	Size	Perm.
APP. LOADED	0x000040000000 -	0x00007b139000	(0x3b139000)	rwx
	0x00007b139000 -	00000x7b306000	(0x1cd000)	rwx
	0x00007b306000 -	0x00007b506000	(0x200000)	---
	0x00007b506000 -	0x00007b516000	(0x10000)	rw-
	0x00007b516000 -	0x00007b742000	(0x22c000)	rw-
MAIN LOADER	0x00007b742000 -	0x00007b786000	(0x44000)	r-x
	0x00007b786000 -	0x00007b985000	(0x1ff000)	---
	0x00007b985000 -	0x00007b988000	(0x3000)	rw-
	0x00007b988000 -	0x00007fd5d000	(0x43d5000)	rw-
	....			

(b) Example of Memory layout of Graphene.

Figure 4: Memory layout example for API-like and Container-like frameworks.

TCSs are particularly important because they are used as input for ENCLU. Moreover, modern SGX threats use TCS addresses for ROP-chains that interacts with the enclave [13]. Therefore, inferring the TCSs is crucial for an analyst to identify such threats in memory.

Container-like – This second type of enclaves usually adopts custom layouts that are designed to host whole applications, as shown in Figure 4b. These frameworks also contain libraries to simulate an operating system [5] and allocate one or more `rwx` blocks for the loaded application. Due to the custom nature of these frameworks, we cannot infer important structures, such as TCSs. However, an analyst can estimate the size of the loaded application by measuring the size of the `rwx` area.

Unknown – If the enclave does not fall in the previous two classes, the information that can be extracted in an automated fashion is very limited. However, it is still possible to gather few clues by observing the page permissions. For instance, an analyst can locate and measure the size of executable pages, which reveals the amount of code loaded in the enclave. Moreover, it is possible to enumerate writable pages (*i.e.,* `rw*`), that likely point to data location.

3.4.2. Enclave Interface Analysis

The *enclave* interface (§2.2) reveals the type of interaction with the underlying OS. However, the *enclave* interface strictly depends from the adopted development framework. To this end, we compiled a list of framework fingerprints composed by specific tokens, list of loaded libraries, and name patterns of the main ELF file used. By using these signatures, it is possible to identify the framework and retrieve the interface information.

In case the framework is unknown, such as for a custom SGX malware, there is no automatic way to infer the enclave interface. However, the host process still requires to invoke an `ENCLU` opcode to interact with the enclave (§2). This reveals the low-level entry-point of the enclave, which serves as starting location for the analyst to manually reverse engineer the enclave interface.

In case of a known development framework, instead, we propose an automatic interface extraction that is based on a combination of memory forensics and reverse engineering techniques, with a similar approach of what Graziano et al. implemented for the analysis of hypervisors [33]. Overall, our analysis is composed of two steps. First, we locate meaningful structures inside a memory dump (*e.g., outside functions* are organized as an array of function pointers). Then, we apply a lightweight static analysis phase to identify the usage of such structures (*e.g.,* the function `sgx_ecall` accepts an array of function pointers as third parameter). This approach is sound, because the code is automatically generated by the framework and thus follows predictable patterns.

It would also be possible to employ more sophisticated semantic software similarity techniques [34]. However, due the nature of the *edge* component, we argue that static approaches best suit our scenario. And in fact, our static analysis solution resulted in zero false positive and zero false negative in our experiments (§4.1). In the following, we provide some examples on how we recognize *outside* and *secure functions*, respectively.

Our approach is based upon two observations. First, important functions (*e.g., outside functions*) are always organized in an array of function pointers (*e.g., ocall_tables*). Second, the location of these structures is hard-coded in crucial utilities. This allows us to automatically locate the usage of this structure in the code by checking the signature of known functions. For instance, in case of the Intel SGX SDK, `sgx_ecall` functions expects a pointer to an *ocall_table* as third parameter (*i.e.,* the register `rdx` for Linux 64). Moreover, the *secure functions* (in Intel SGX SDK) always invoke `sgx_ecall` from an external library (`libsgx_urt.so`), thus importing the symbol. Therefore, locating the invocations points to `sgx_ecall` reveals both *outside* and *secure functions*. In Appendix A, we detail the algorithm used.

Recovering the enclave interface is important when the enclave content is inaccessible (*e.g.,* the enclave is in release mode). For instance, an analyst might find an undocumented enclave in the system (*e.g.,* a malware-enclave) and he is interesting in studying its behavior. In this case, the *secure functions* indicate the data accepted by the enclave, *e.g.,* buffers, pointers to the untrusted memory. The *outside functions*, instead, represent the interaction with the system, *e.g.,* writing files, network communication. The combination of *outside* and *secure functions*, therefore, can give an intuition of the internal enclave logic.

4. Evaluation

We implemented our techniques as a set of plugins for Volatility [35] version 2.6.1. We built the static analysis component on top of radare2 [36] (commit `..C4B4BF52`), and for the acquisition we extended LiME [37] (commit `..3864A39F`). All components are released as open source.[1]

Experiments Setup – To test our solution both with and without a virtualization layer, we repeated each experiment in two different environments: a bare metal machine and a virtual machine running in the Microsoft Azure cloud [38]. The bare metal machine was a 64bit machine equipped with an Intel i7-8565 CPU and 16GB of RAM. Its operating system was an Ubuntu 18.04 with kernel version 5.4.0-42-generic using the legacy Intel `isgx` driver version 2.11.0. For the virtual machine, we used a Standard D4 v3 64bit Gen2 machine equipped with 4 vCPUs and 16GB of RAM. Its operating system was an Ubuntu 18.04 with kernel version 5.4.0-1034-azure and it mounted the Intel `DCAP` driver version 1.33.

Dataset – To evaluate the techniques presented in Section 3, we collected a dataset of 45 enclaves. In particular, we included 40 samples taken from the repository of the respective development frameworks. These samples contain different types of applications that range from simple features demonstrations to complex projects such as databases and Web servers. We also included three enclaves distributed by Conclave [16], which is a commercial development framework for SGX. Finally, we included two malware-enclaves, SGX-ROP [12] and SnakeGX [13]. For sample labeling, we followed this convention: the sample name is the concatenation of the development framework, the

[1] The proof-of-concept tools are available at `https://github.com/tregua87/sgx-forensic`.

Sample[1]	Flags[2]	Ecall	Ocall	Time (s)
Intel SDK-isgx				
S-I-1	-	25	6	61
S-I-2	-	4	7	83
S-I-3	-	32	11	87
S-I-4	-	3	4	40
S-I-5	-	2	5	46
S-I-6	-	3	4	41
S-I-7	-	32	11	88
Open Enclave-isgx				
O-I-1	-	6	82	237
O-I-2	-	9	83	261
O-I-3	-	7	81	341
O-I-4	-	7	81	344
O-I-5	-	8	81	228
O-I-6	-	6	82	201
O-I-7	-	7	81	346
O-I-8	-	6	82	238
Asylo-isgx				
A-I-1	-	1	0	324
A-I-2	-	1	0	459
A-I-3	-	1	0	339
A-I-4	-	1	0	451
A-I-5	-	1	0	297
Rust SDK-isgx				
R-I-1	-	6	62	685
R-I-2	-	3	60	369
R-I-3	-	9	85	509
R-I-4	-	8	85	468
R-I-5	-	3	60	400
R-I-6	-	3	70	66
R-I-7[3]	-	3	60	59
Graphene-isgx				
G-I-1	E	1	40	64
G-I-2	-	1	40	65
G-I-3	-	1	40	64
G-I-4	-	1	40	82
G-I-5	-	1	40	108
G-I-6	-	1	40	64
G-I-7	E	1	40	75
G-I-8	E	1	40	67
G-I-9	E	1	40	68
SGX LKL-isgx				
L-I-1	-	1	17	217
L-I-2	-	1	17	531
L-I-3	-	1	17	250
L-I-4	-	1	17	390
Samples from Conclave [16]				
S-D-1	-	4	13	1800
S-D-2	K	0	0	8
S-D-3	K	0	0	9
Malware-enclave samples				
SGX-ROP	-	1	1	67
SnakeGX	-	4	0	66

[1] Format `framework-driver-sample number`. Frameworks: `A`: Asylo, `G`: Graphene, `L`: SGX-LKL, `O`: Open Enclave SDK, `R`: Rust-SGX, `S`: Intel SGX SDK. Drivers: `I`: Intel `isgx` `D`: Intel `DCAP`
[2] E: Evicted, K: Provisioned Keys
[3] See text for more details about the classification of this sample.

Table 2: Analyzed samples – for each enclave, we indicate the flags from the driver, the enclave interface, and the analysis time. Our enclave interface analysis faithfully located both *ocalls* and *ecalls* for each sample, thus producing *zero* false positives and *zero* false negatives in locating the enclave interfaces.

driver, and an incremental number, for instance, **S-I-1** indicates the first enclave among those using the Intel **S**GX SDK and the ***i**sgx* driver.

4.1. Results

We designed a set of experiments to validate the ability of our tools to correctly acquire enclave information from the system memory and analyze it in all the conditions depicted in Figure 2.

All results are reported in Table 2, where for each enclave we report the recovered flags, the number of *ecall* and *ocall* extracted by our tool, and the total analysis time. Since all enclaves expose the flags `INIT` (which indicates that the enclave is correctly loaded and ready to interact with its host process) and `MODE_64BIT` (which specifies the architecture), we omit them from the table. Furthermore, the table does not show the *quoting* and the *provisioning enclaves*, that however were correctly found as well (§2). Without loss of generality, we performed our experiments over enclaves in debug mode, while we simulated the enclaves in release mode by ignoring the enclave content from the acquired image. This setting is reasonable since enclaves in debug and release mode only differ from the debug flag, while kernel and user space structures do not change, a similar evaluation strategy was adopted in [15].

In terms of performances, our analysis took on average 246.32s (around 4 minutes) per memory dump, with a minimum of 8.59s and a maximum of 1800.89s (around 30min) for the commercial Conclave enclaves (for which the majority of the analysis time was spent to identify the list of possible structures for *ocall* and *ecall_table*).

Acquisition - To begin with, we validated the memory acquisition phase (§3.2) by using our modified version of LiME to extract the SGX capabilities of the machine (*e.g.,* EPC pages, microcode version) and to content of enclaves in debug mode. For the enclaves in debug mode, we verified that the code and data extracted from the enclaves appeared in the system image and could be inspected by Volatility plugins and static analysis tools. The memory acquisition always worked correctly on both bare metal and Azure VM. Finally, we observed the microcode was not vulnerable in both machines.

Kernel Drivers Analysis - For the kernel analysis (§3.3), we simulated a system image obtained from standard memory acquisition tools, thus lacking SGX information. In this case, we deduced the presence of SGX from the system logs extracted from the memory. We also verified that the information inferred from the system logs matched with the one extracted from our acquisition tool and the original machine. Then, we tested our capabilities to inspect both Intel drivers (`isgx` and `DCAP`), thus showing the correct discovery of *enclaves*, flags, and ELRANGEs. In addition, for the `isgx`, we tested the detection of *zombie* enclaves in memory.
Unknown Drivers - We simulated a machine containing unknown drivers to test the heuristic described in (§3.3). As a result, we managed to find all the *enclaves* and the relative process. However, we also erroneously classified three processes as containing *enclaves* (*i.e., xorg*, *insmod*, and *lxcfs*). This result is expected because our heuristics assumes zero knowledge of SGX drivers and EPC zones. In particular, false positives can appear because a process uses special memory pages (*e.g.,* associated to MMIO devices) that are similar to the SGX ones (more details in Section 3.3). Other false positives, as in case of *insmod*, are caused by non-atomic memory acquisition issues, as already discussed by Pagani et al. [39].
User-Space Analysis - For what concerns the user space analysis (§3.4), we inspected the *enclaves*' memory layouts (§3.4.1) and correctly inferred the type of adopted framework (*i.e.,* API-like or Container-like). Moreover, our tool was able to extract meaningful structure addresses in case of API-likes (*e.g.,* TCS, SSA) or the loaded process size for Container-like ones. Then, we applied the interface analysis (§3.4.2) and located all the *ecall* and *ocall* locations in user space. We manually verified the identified functions by reversing the applications and matching them with the respective source files, when possible. We thus observed that all the functions inferred faithfully matched the ones defined in the original enclave applications. Therefore, our analysis produced *zero* false positives and *zero* false negatives in locating the enclave interfaces.
Framework Fingerprinting - All frameworks were correctly identified, with the exception of R-I-7, which was wrongly classified by our tool as Intel SGX SDK instead of RUST-SGX. This is due to the fact that the two frameworks share many similarities. Nonetheless, the plugin managed to correctly retrieve the correct interface information. Finally, we simulated unknown frameworks and successfully locate the `ENCLU` opcodes in the dumps as well.

5. Case Studies

We now describe how our SGX forensic analysis guidelines and techniques can be applied to more complex real-world scenarios. In particular, we chose three examples: a commercial application based on a proprietary development framework, and two custom malware-enclaves.

5.1. Commercial SGX application

In this first scenario, we try to analyze an image containing an enclave developed with Conclave [16], a commercial SGX development framework distributed by R3 [40]. Conclave is designed to abstract the SGX programming pattern for standard Java applications, *e.g.,* by including utilities to perform remote attestation. Moreover, Conclave embeds the native libraries for the interaction between application and enclave, while the operating system requires only the drivers installed. At the time of writing, Conclave works only with the new `DCAP` driver.

Analysis – We loaded the default application delivered with Conclave in an Azure VM and dumped the system memory with our modified acquisition tool. Our kernel analysis plugins reported a process containing *three* enclaves (samples S-D-1, S-D-2, and S-D-3 in Table 2). The samples S-D-2 and S-D-3 refer to the *quoting* (QE) and the *provisioning* enclaves (PE), which can be easily recognized because our system identifies the presence of the `PROV_KEY` flag (K), which is only used for system enclaves. In fact, Conclave loads its own copy of QE and PE that work in parallel with the ones handled by the `aesmd` system service. The third enclave (S-D-1) contains the main application logic.

We then executed our user-space analysis plugin with its default configuration. As a result, the plugin automatically classified the application as based on the Intel SGX SDK. This is expected because Conclave is built on top of the standard Intel SGX SDK and our system did not have specific signature for the commercial framework. However, out-of-the-box our system failed to recover the enclave interfaces.

This is due to the fact that Conclave handles Java applications, and therefore the main ELF of the process is the JVM itself — while the actual application is mapped as an external library. To overcame this problem, we identified the

shared library containing the main application by inspecting the imported symbols (as it needs to export the symbol `sgx_ecall`). Then, we configured our user-space plugin to analyze that library instead of the JVM binary. This allowed the plugin to correctly locate and report all the interfaces. We also analyzed the outside functions and verified their correctness. In particular, our plugin allowed us to identify that the enclave performed network operations, and to isolate other *outside functions* for file system interaction and remote attestation. Like in all previous experiments, our plugins successfully identified all the *ecall* and *ocall* without false positive nor false negatives.

As mentioned in the previous section, the analysis of the Conclave enclave required a longer time (around 30min) because the applications contained a large amount of `r-x` and `r--` pages. Therefore, our plugin required more time to perform the static analysis and extract the possible *ocall_table*.

5.2. SGX-ROP malware

In this second scenario, we analyze the first example of malware-enclave proposed by researchers (SGX-ROP in Table 2). SGX-ROP exploits TSX to break the ASRL of the host application and build ROP chains to attack the system. The ROP chains are appended to the stack in the untrusted zone and they are activated when the *secure function* returns. Once the payload ends, the ROP chains resume the normal host application execution thus limiting possible side-effects. In the available PoC, the malicious payload simulates a ransomware behavior by creating new file on the host machine.

Analysis – To analyze this sample, we load the application, acquire the memory, and execute our analysis with default configuration. First, our plugin automatically listed an unexpected process containing an enclave. Since the PoC is written on top of the standard Intel SGX SDK, our analysis correctly inferred the development framework, the memory layout, and all the interfaces.

At first inspection, the enclave does not seem able to write files to the file system as it has only one *outside function* that simply invokes `printf()` for debug purposes. This is interesting, as it is unusual for an enclave to have no *outside function* for interacting with the OS.

This is all the information that can be retrieved from the enclave. However, an enclave can transfer data into the host process memory space, and can use this to manipulate the host process execution. The easiest way to achieve this is through code-reuse attacks (*e.g.,* ROP-chains), whose presence can be identified by using an existing Volatility plugin [41]. In our memory image, the plugin indeed reports the presence of a ROP chain in the enclave process, that contains calls to write files. Moreover, stack analysis techniques [42] showed a corrupted call stack in which a return address pointed to the ROP chain extracted.

This example shows how our SGX analysis plugins can be combined with other existing memory forensics techniques to gain a more complete picture of this advanced threat.

5.3. SnakeGX malware

In our final case study we analyze an infector that converts a benign enclave into a malware container [13] (SnakeGX in Table 2). SnakeGX does not add unexpected enclaves that may attract the analyst's attention, like in the case of SGX-ROP. In fact, the PoC delivered with SnakeGX infects a specific enclave, called StealthDB [43], which is an SGX plugin for PostgreSQL [44]. Internally, SnakeGX's payload uses a set of ROP chain in the untrusted memory, called *outside-chains*, to interact with the OS and exfiltrate cryptographic keys on-demand. The *outside-chains* end with an `ENCLU` opcode and a TCS address to resume the enclave execution. In our scenario, we assume the machine owner suspects an information leak due to an unusual outgoing network traffic, thus requiring an inspection.

Analysis – Like in all previous experiments, we load the application, acquire the memory, and execute our analysis plugins with their default configuration. Since SnakeGX infects StealthDB, our system only reports the presence of this enclave. StealthDB works on top of the standard Intel SGX SDK, therefore, our analysis correctly extracted the development framework, the memory layout, and the interfaces. From a first inspection, we noticed that the enclave interface matches StealthDB specifications, and it contains 10 trusted threads. Moreover, our analysis did not spot new enclaves (active or zombies), thus ruling out the presence of unexpected enclaves in the host process. Finally, the PostgreSQL was reported to work correctly, which is coherent because SnakeGX does not affect the infected enclave's behavior.

As in the previous case study, an attacker could have used a code-reuse attack to manipulate the external process execution. To this end, when investigating possibly malicious enclaves, the analyst should always check for the

presence of ROP chains. In this case, Volatility reports a chain in the `.bss` segment that, at a closer analysis, contains gadgets to interact with the system, a TCS reference, and a gadget pointing to an `ENCLU` opcode. A manual analysis confirms that this is indeed the *outside-chain* of SnakeGX.

6. Discussion

In this section, we discuss the limitations of our techniques and indicate future works.

Other Platforms – Our study focuses on Linux machines. This choice came from the observation that the majority of the development frameworks were designed for Linux. However, our consideration can be easily extended on Windows environments. In fact, the memory acquisition is independent from the operating system and the Intel drivers use similar structures also its Windows counterpart. Moreover, our tool also includes heuristic to detect enclaves in case of unknown drivers. For what concerns the development frameworks, our consideration about the memory layout does not changes, while the interface identification would require small changes to be adapted to the Windows OS. This would mainly affect our static analysis part, to consider a different calling convention when matching specific structures (*e.g., ocall_tables*) and target functions (*e.g.,* `sgx_ecall`).

Similar consideration applies to 32bit systems. In this case, we can easily adapt our interface analysis by considering typical 32bit calling conventions and the fact that data structures (*e.g., ocall_tables*) contain pointers of 4 bytes.

Side Channels – In our work, we do not consider enclaves that perform pure side-channel attacks against the system [14]. In this scenario, the enclave gathers private information without using *outside functions* or ROP-chain in the untrusted zone, thus minimizing the traces left in the system. However, even if an enclave uses side-channel attacks, it still has to rely on either *outside functions* or ROP-chains to actually exfiltrate the information outside the system. Therefore, we can rely on forensic techniques similar to the one described in Section 5.2) and 5.3. In addition, we can use orthogonal techniques (*i.e.,* SGX-Bouncer [45]) to detect enclave side-channel attacks at runtime. However we consider the use of these orthogonal approaches outside the scope of our study.

7. Related works

To the best of our knowledge, no academic works have methodically studied the SGX technology from a memory forensics perspective. In the literature, a few works use sparse memory information for crafting attacks, and others propose mitigation for malware-enclave threats.

Forensic in SGX environment – Zhang et al. [22] in "Memory forensic challenges under misused architectural features" demonstrated how important and lacking are SGX forensics tools. In their work, they show how a malicious enclave can encrypt files, as classical ransomware, without the possibility for a forensic analyst to recover the encryption key stored in the enclave also with complete access to the system. Furthermore, an analyst who wants to perform any type of analysis on SGX capable systems has, at the moment, few valid dump tools to gain access to enclave memory space. An example is [15] in which the authors use Intel DCI interface to stop the CPU and dump enclaves at least in debug mode. The exposed technique leverage the fact that the SGX threat model excludes hardware attacks and require the use od specific debug hardware to take the control of the CPU and access to the memory. However, the requirement of hardware access to the machine and specific firmware bugs or misconfigurations undermine its usefulness in generic forensics surveys on SGX capable machines. For instance, even advanced software tools, which leverage high privilege CPU modes as SMM mode, like [46] become ineffective against SGX systems. To fill this gap, in this paper we have performed a deep study of the traces left in the system memory by the enclaves, and developed practical tools to recover a large amount of enclave-related information.

Attacks that rely on enclave memory information – To perform code-reuse attacks against an enclave, an adversary has to infer the enclave base address to swift the gadgets accordingly. Previous works from Biondo et al. [47] and Toffalini et al. [13] achieved this goal by reading the ELRANGE from user space, looking for pages belonging to the `isgx` device. However, their analysis was limited to the enclave base address. Moreover, both papers assumed the attacker already knows the enclave code to build the malicious payload. Van Bulck et al. [26] and Alder et al. [48]

analyzed the Application Binary Interface (ABI) of popular SGX development frameworks to find vulnerabilities. On the contrary, the goal of our study is to collect evidence about the enclave content from difference parts of the system.

Malware-enclave mitigation – A number of countermeasures have been proposed to deal with the threat of malware-enclaves. A first solution was proposed by Weiser et al. [49] with SGXJail, a sandbox for malware-enclave. The authors goal was to contain the actions of a (possible) malware by reducing the syscalls available from the host process. In fact, SGXJail does not inspect the enclave content, and thus does not provide any forensic-relevant information.

Zhang et al. [45] proposed SGX-Bouncer, a technique to infer an enclave behavior without breaking its isolation. SGX-Bouncer scans the cache access in an attempt to detect illegal memory accesses, such as side-channels or unauthorized enclave exits. The project works with runtime information, and its purpose is to detect running malware-enclave. On the contrary, we deal with a static image of the system. Therefore, SGX-Bouncer can be seen as a complementary tool for a comprehensive runtime analysis, as discussed in (§6).

8. Conclusion

In this paper, we presented a set of practical guidelines to inspect the physical memory acquired from an SGX-enabled machine. Our work covers many aspects of memory forensics, such as retrieving EPC zones and *enclaves* information, rebuilding kernel structures, and analyzing the *enclave* interfaces and layout. This information can help human analysts to discover the presence of enclaves, their corresponding user-space processes, and kick-start manual reverse engineering by pointing out the functions responsible for the communication to, and from, the enclave code.

We implemented these techniques in an acquisition tool and in a set of Volatility plugins and show how they can be applied to the analysis of 45 samples and three real use-case scenarios.

9. Acknowledgments

This research was supported by the European Research Council (ERC) under the Horizon 2020 research and innovation program (grant agreement No 771844 – BitCrumbs).

References

[1] V. Costan, S. Devadas, Intel sgx explained., IACR Cryptology ePrint Archive 2016 (2016) 86.
[2] F. Toffalini, M. Ochoa, J. Sun, J. Zhou, Careful-packing: A practical and scalable anti-tampering software protection enforced by trusted computing, in: Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 231–242. doi:10.1145/3292006.3300029.
URL https://doi.org/10.1145/3292006.3300029
[3] P.-L. Aublin, F. Kelbert, D. O'keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, P. Pietzuch, Talos: Secure and transparent tls termination inside sgx enclaves, Imperial College London, Tech. Rep 5 (2017).
[4] Intel, Intel sgx sdk, https://github.com/intel/linux-sgx, last access February 2021 (2016).
[5] C. che Tsai, D. E. Porter, M. Vij, Graphene-sgx: A practical library OS for unmodified applications on SGX, in: 2017 USENIX Annual Technical Conference (USENIX ATC 17), USENIX Association, Santa Clara, CA, 2017, pp. 645–658.
URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai
[6] Microsoft, Open enclave sdk, https://github.com/openenclave/openenclave, last access February 2021 (2019).
[7] Google, Asylo, https://github.com/google/asylo, last access February 2021 (2019).
[8] L.-S. D. . S. L. Group, Sgx-lkl, https://github.com/lsds/sgx-lkl, last access February 2021 (2020).
[9] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, Z. Lin, Towards memory safe enclave programming with rust-sgx, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 2333–2350. doi:10.1145/3319535.3354241.
URL https://doi.org/10.1145/3319535.3354241
[10] J. Rutkowska, Thoughts on intel's upcoming software guard extensions (part 1), http://theinvisiblethings.blogspot.com/2013/08/thoughts-on-intels-upcoming-software.html, last access March 2021 (2013).
[11] J. Rutkowska, Thoughts on intel's upcoming software guard extensions (part 2), http://theinvisiblethings.blogspot.com/2013/09/thoughts-on-intels-upcoming-software.html, last access March 2021 (2013).
[12] M. Schwarz, S. Weiser, D. Gruss, Practical enclave malware with intel sgx, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2019, pp. 177–196.
[13] F. Toffalini, M. Graziano, M. Conti, J. Zhou, Snakegx: A sneaky attack against sgx enclaves, in: K. Sako, N. O. Tippenhauer (Eds.), Applied Cryptography and Network Security, Springer International Publishing, Cham, 2021, pp. 333–362.

[14] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, S. Mangard, Malware guard extension: Using sgx to conceal cache attacks, in: M. Polychronakis, M. Meier (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Springer International Publishing, Cham, 2017, pp. 3–24.
[15] T. Latzo, M. S. FAU, F. F. FAU, Leveraging intel dci for memory forensics, Digital Investigation.
[16] R3, Conclave, `https://docs.conclave.net/index.html`, last access March 2021 (2020).
[17] J. Yao, V. J. Zimmer, Q. Long, System management mode isolation in firmware, uS Patent App. 12/317,446 (May 7 2009).
[18] J. S. Coke, A. V. Bhatt, S. Graham, D. Lent, Implementing scatter/gather operations in a direct memory access device on a personal computer, uS Patent 5,708,849 (Jan. 13 1998).
[19] Intel, Intel® software guard extensions (intel®sgx) - developer guide, `https://download.01.org/intel-sgx/linux-2.1.3/docs/Intel_SGX_Developer_Guide.pdf`, last access June 2020 (2013).
[20] C. Rozas, Intel® software guard extensions (intel® sgx).
[21] P. Guide, Intel® 64 and ia-32 architectures software developer's manual, Volume 3D: System programming Guide, Part 4 (41).
[22] N. Zhang, R. Zhang, K. Sun, W. Lou, Y. T. Hou, S. Jajodia, Memory forensic challenges under misused architectural features, IEEE Transactions on Information Forensics and Security 13 (9) (2018) 2345–2358.
[23] 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, ACM New York, NY, USA, 2013, p. 7.
[24] M. Lee, Awesome sgx open source projects, `https://github.com/Maxul/Awesome-SGX-Open-Source`, last access Feb 2021 (2019).
[25] J. Liao, Awesome sgx, `https://github.com/Liaojinghui/awesome-sgx`, last access Feb 2021 (2019).
[26] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, F. Piessens, A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1741–1758. `doi:10.1145/3319535.3363206`.
URL `https://doi.org/10.1145/3319535.3363206`
[27] A. Nilsson, P. N. Bideh, J. Brorsson, A survey of published attacks on intel sgx (2020). `arXiv:2006.13598`.
[28] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, R. Strackx, Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution, in: Proceedings of the 27th USENIX Security Symposium, USENIX Association, 2018.
[29] S. van Schaik, A. Kwong, D. Genkin, Y. Yarom, SGAxe: How SGX fails in practice, `https://sgaxeattack.com/` (2020).
[30] Intel, Linux-sgx-driver, `https://github.com/intel/linux-sgx-driver`, last access March 2021 (2016).
[31] Intel, Sgx data center attestation primitives (dcap), `https://github.com/intel/SGXDataCenterAttestationPrimitives/tree/master/driver`, last access March 2021 (2020).
[32] Intel, Intel software guard extensions (intel sgx) protected code loader (pcl) for linux, `https://github.com/intel/linux-sgx-pcl/blob/master/Intel(R)_SGX_Protected_Code_Loader_for_Linux_User_Guide.pdf`, last access June 2020 (2018).
[33] M. Graziano, A. Lanzi, D. Balzarotti, Hypervisor memory forensics, in: S. J. Stolfo, A. Stavrou, C. V. Wright (Eds.), Research in Attacks, Intrusions, and Defenses, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 21–40.
[34] K. Pei, Z. Xuan, J. Yang, S. Jana, B. Ray, Trex: Learning execution semantics from micro-traces for binary similarity, arXiv preprint arXiv:2012.08680.
[35] V. Foundation, Volatility framework - volatile memory extraction utility framework, `https://github.com/volatilityfoundation/volatility`, last access February 2021 (2017).
[36] R. Team, Radare2 github repository, `https://github.com/radare/radare2` (2017).
[37] S. Joe, Lime linux memory extractor, `https://github.com/504ensicsLabs/LiME`, last access February 2021 (2012).
[38] Microsoft, Introducing azure confidential computing, `https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/`, last access November 2018 (2017).
[39] F. Pagani, O. Fedorov, D. Balzarotti, Introducing the temporal dimension to memory forensics, ACM Transactions on Privacy and Security (TOPS) 22 (2) (2019) 1–21.
[40] R3, R3, `https://www.r3.com/`, last access March 2021 (2014).
[41] C. @and Inon Weber, ropfind repository, `https://github.com/orchechik/ropfind` (2019).
[42] E. Smulders, User space memory analysis, Master's thesis, University of Twente (2013).
[43] D. Vinayagamurthy, A. Gribov, S. Gorbunov, Stealthdb: a scalable encrypted database with full sql query support, Proceedings on Privacy Enhancing Technologies 2019 (3) (2019) 370 – 388.
URL `https://content.sciendo.com/view/journals/popets/2019/3/article-p370.xml`
[44] J. D. Drake, J. C. Worsley, Practical PostgreSQL, " O'Reilly Media, Inc.", 2002.
[45] Z. Zhang, X. Zhang, Q. Li, K. Sun, Y. Zhang, S. Liu, Y. Liu, X. Li, See through walls: Detecting malware in sgx enclaves with sgx-bouncer.
[46] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, D. Bruschi, When hardware meets software: A bulletproof solution to forensic memory acquisition, in: Proceedings of the 28th annual computer security applications conference, 2012, pp. 79–88.
[47] A. Biondo, M. Conti, L. Davi, T. Frassetto, A.-R. Sadeghi, The guard's dilemma: Efficient code-reuse attacks against intel SGX, in: 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, Baltimore, MD, 2018, pp. 1213–1227.
URL `https://www.usenix.org/conference/usenixsecurity18/presentation/biondo`
[48] F. Alder, J. Van Bulck, D. Oswald, F. Piessens, Faulty point unit: Abi poisoning attacks on intel sgx, in: Annual Computer Security Applications Conference, ACSAC '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 415–427. `doi:10.1145/3427228.3427270`.
URL `https://doi.org/10.1145/3427228.3427270`
[49] S. Weiser, L. Mayr, M. Schwarz, D. Gruss, Sgxjail: Defeating enclave malware via confinement, in: 22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019), 2019, pp. 353–366.

Appendix A. Interface analysis algorithms

These are the pseudo-code used to extract either *secure* and *outside functions* from a process memory dump. The algorithm 1 shows our approach for the Intel SGX SDK and RUST-SGK, the algorithm 2 show our approach for Open Enclave SDK, and finally the algorithm 3 show our approach for Graphene and SGX-LKL.

Appendix A.1. Intel SGX SDK and RUST-SGX

Alg. 1 describes the approach for these two frameworks. In particular, we exploit the fact that the host application imports the symbols `sgx_ecall` from `libsgx_urts.so`.

In the algorithm, we first create a list of *ecall* (line 2) and confirmed *ocall_table*s (line 3); that will contain the enclave interfaces. Then, we extract a list of possible *ocall_tables* from the `r--` and `r-x` pages of the ELF extracted from memory (line 4). We consider possible *ocall_tables* any list of addresses (*i.e.,* sequence of 8 bytes) that point to `r-x` pages of the main ELF. In the rest of the algorithm, we apply a static analysis pass on the code scanning all functions (line 5). Our goal is to identify those functions *f* that invoke `sgx_ecall` by using one of the possible *ocall_table* as parameter (line 7). Once we locate a function that satisfies our requirements, we save the function *f* in *ecall_list* and its *ocall_table* in *ocall_table_conf* (lines 8 and 9). Finally, we return *ecall_list* and *ocall_table_conf* (line 11).

Appendix A.2. Open Enclave SDK

Alg. 2 describes the approach for Open Enclave SDK. In this case, the framework identifies the *secure functions* through string tokens that are grouped in an array of strings, called *ecall_tokens*.

In the algorithm, we first create a list of *ecall* (line 2) and confirmed *ocall_table*s (line 3); that will contain the enclave interfaces. Moreover, we create a list of *ecall_tokens_conf* (line 4). At this point, we extract a list of possible *ocall_tables* (line 5) and a list of possible *ecall_tokens* (line 6) from the `r--` and `r-x` pages of the main ELF. We consider possible *ocall_tables* any list of addresses (*i.e.,* sequence of 8 bytes) that point to `r-x` pages of the main ELF. Moreover, we consider possible *ecall_tokens* any array of null-terminated ascii strings. In the rest of the algorithm, we statically analyse all functions (line 7). to identify those functions *f* that invokes `oe_create_enclave`, the function used to create an enclave. In particular, `oe_create_enclave` expects an *ocall_table* as 8th parameter (pushed into the stack) and `ecall_token` as third parameter (line 8). Once we locate a function that satisfies our requirements, we save the *ocall_table* in *ocall_table_conf* and the *ecall_tokens* in *ecall_tokens_conf* (lines 10 and 11). The *ecall_tokens* are also used in the actual *secure function* invocation. We thus scan the functions and select those that have a pointer to one of the *ecall_tokens_conf* elements (line 13 to 15). Finally, we return *ecall_list* and *ocall_table_conf* (line 17).

Appendix A.3. Graphene and SGX-LKL

Alg. 3 describes the approach for these two frameworks. In this case, the enclave exposes *one* secure function that boots the application loaded into the enclave.

In the algorithm, we list the functions that contain the `ENCLU` opcode in the main ELF memory space (line 2). The `ENCLU` plays the role of *ecall* for the enclave. Then, we create a list of confirmed *ocall_table*s (line 3) and extract a list of possible *ocall_tables* from the `r--` and `r-x` pages of the main ELF (line 4). In the rest of the algorithm, we scan all functions (line 5). to identify those functions *f* that refers to *ocall_table* (line 6). Here, Graphene and SGX-LKL refers to *ocall_table* in a slightly different way, but for sake of simplicity we consider both cases equivalent. Once we locate a function that satisfy our requirements, we consider the pointed *ocall_table* as confirmed an we save its reference to *ocall_table_conf* (lines 7). Finally, we return *ecall_list* and *ocall_table_conf* (line 9).

Algorithm 1: Interface pseudo-code for Intel SGX SDK and RUST-SGX.

```
1  get_interface(process)
2      ecall_list ← ∅
3      ocall_table_conf ← ∅
4      ocall_tables ← find_ocall_table()
5      for f ∈ process.functions do
6          if f contains call to sgx_ecall ∧
7          f has rdx pointing to ocall_tables then
8              ecall_list ← f
9              ocall_table_conf ← rdx
10     end
11     return (ecall_list, ocall_table_conf)
```

Algorithm 2: Interface pseudo-code for Open Enclave SDK.

```
1  get_interface(process)
2      ecall_list ← ∅
3      ocall_table_conf ← ∅
4      ecall_tokens_conf ← ∅
5      ocall_tables ← find_ocall_table()
6      ecall_tokens ← find_ecall_tokens()
7      for f ∈ process.functions do
8          if f has X pointing to ocall_tables ∧
9          f has Y pointing to ecall_tokens then
10             ocall_table_conf ← X
11             ecall_tokens_conf ← Y
12     end
13     for f ∈ process.functions do
14         if f has a pointer to ecall_tokens_conf then
15             ecall_list ← f
16     end
17     return (ecall_list, ocall_table_conf)
```

Algorithm 3: Interface pseudo-code for Graphene and SGX-LKL.

```
1  get_interface(process)
2      ecall_list ← find_enclu(process)
3      ocall_table_conf ← ∅
4      ocall_tables ← find_ocall_table()
5      for f ∈ process.functions do
6          if f has X pointing to ocall_tables then
7              ocall_table_conf ← X
8      end
9      return (ecall_list, ocall_table_conf)
```