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

TODO thesis in a glance. ◁ My thesis argues that Trusted Execution Environments (TEE), such as SGX, are powerful tools that isolate portion of code against strong adversaries (*i.e.*, the OS itself). However, TEEs are not the silver-bullet of cyber security and they suffer of limitations in terms of scalability and security.

I argue that we can further extend the TEE properties by carefully choosing smarter software design without the need of changing the TEE modules (and thus changing the hardware). Throughout the dissertation, I first investigate the TEE limitations, then, I will propose relative solutions. TODO I have to discuss every point carefully, this is just a memory for me. ◁

Overall, my study covers five TEE aspect:

- TODO Scalability untrusted code protection ◁ First, I will face a scalability issues that affect many modern TEE technologies (Chapter 3).
- TODO New defenses for the untrusted code ◁ Then. I will use TEE to implement runtime protection for untrusted memory (Chapter 4).
- TODO New threats ◁ At this point, I covered static and runtime protection for the untrusted code, now I will investigate new type of threats for the *enclaves* themselves (Chapter 5).
- TODO New defenses for the trusted code ◁ From this point, I will design new defenses for TEE *encalves* (Chapter 6)
- TODO Forensic analysis ◁ Finally, I will investigate the new challenges introduced by TEE technologies in terms of memory-forensic analysis (Chapter 7).

Chapter 2

Security Properties of TEE

This is the background of Trusting Technologies, mainly SGX and TrustZone (?).

Chapter 3

A Practical and Scalable Software Protection enforced by TEE

In this chapter, we propsoe 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 work 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, 2017; Abera et al., 2016a. 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., 2017. 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., 2016; Tsai, Porter, and Vij, 2017.

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

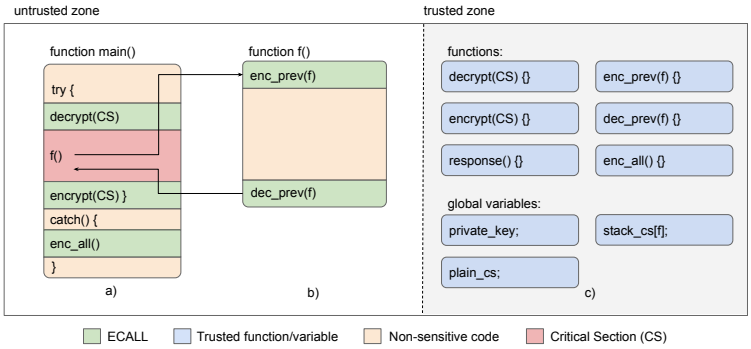


FIGURE 3.1: An overview of our schema for single-thread applications, the memory is split in trusted and untrusted zones. The trusted zone contains all methods required for our technique, while in the untrusted zone we show the interaction of those methods with the CSs.

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 4.4), we use the Intel Software Guard eXtension (SGX) Rozas, 2013 terminology. However, it is possible to use other trusted modules (see Section 3.4.6).

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 Møller and Schwartzbach, 2012 of all possible branches from *CS_Begin* to *CS_End*, and considering all function calls as atomic operations. 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

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].

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

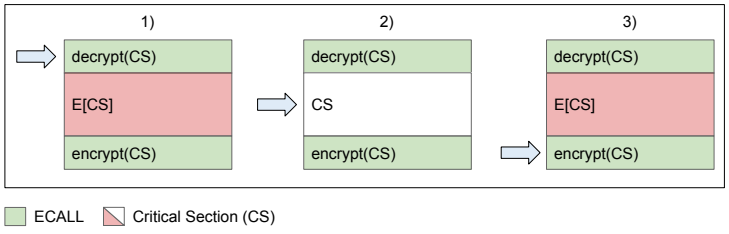


FIGURE 3.2: Packing mechanism of our schema.

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 **??** TODO reference ◁). 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. Details on sealing and attestation are discussed in Section **??** TODO add background ◁.

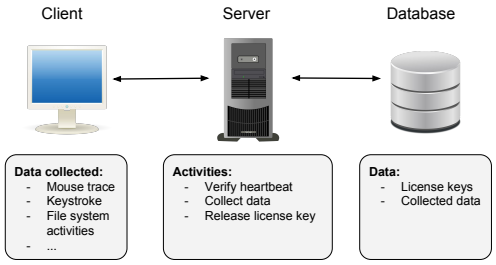
3.3 Implementation

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.

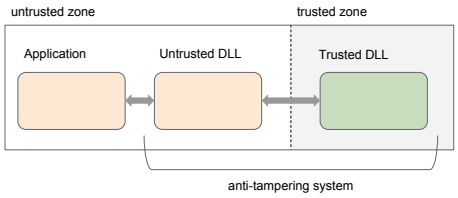
3.3.1 Client

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.

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.



(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.

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 Zhou, Michalik, and Hinsenkamp, 2007 between the assembly code and the license key. SGX natively provides an implementation of this algorithm Intel, 2018.

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 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.

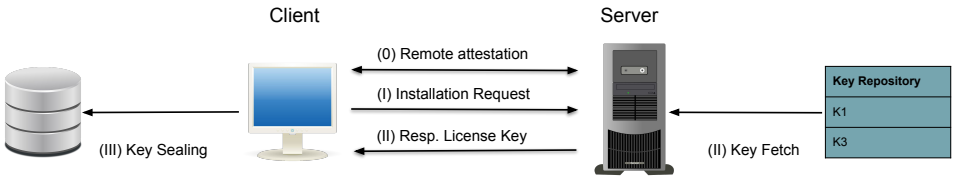


FIGURE 3.4: Secure installation protocol between client and server.

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

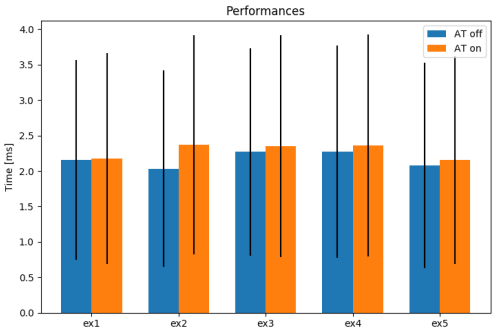
Module	LoC	Perc.
Main program	12175	96,5%
Untrusted DLL	248	2,0%
Trusted DLL	186	1,5%

3.4.2 Microbenchmark Measurements

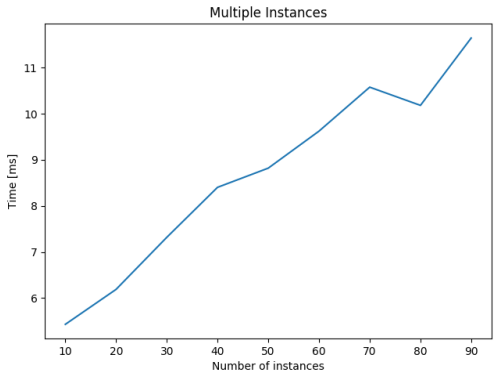
In these experiments, we perform a set of microbenchmark to measure the overhead in time introduced by our technique. As a use case, we measure the execution time of the CSs in our proof-of-concept monitoring agent (see Section 4.4). At first, we briefly introduce the experiment setup. Then, we measure 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 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.



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



(B) Average response time with multiple instances

FIGURE 3.5: Careful-Packing evaluation.

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 **??** TODO reference ◁). 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 4.4, 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.

3.4.6 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

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 (*Amazon Web Services (AWS)* 2006) or Microsoft Azure (*Microsoft Azure* 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$\oplus$X/DEP Pinzari, 2003, Address Space Layout Randomization - ASLR 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 *McSema* 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 Dessouky et al., 2018 or the thousands per second Abera et al., 2019 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*).

4.1 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., 2016b; 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 4.5.4 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 4.6. 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.

4.2 ScaRR Control-Flow 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 4.2.1) and, then, the challenges we faced during its design (Section 4.2.2).

4.2.1 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 **??**. 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).

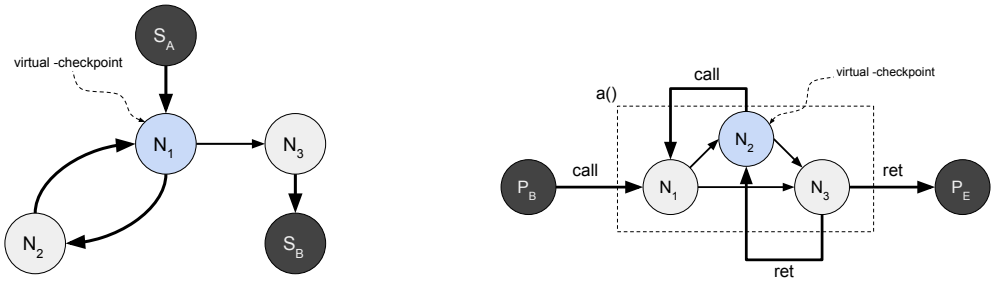
4.2.2 Challenges

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 4.1a, 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 4.1a are as follows:

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

Recursions. In Figure 4.1b, 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



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

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

FIGURE 4.1: ScaRR model challenges.

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 4.1b 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.

4.3 System 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 Abera et al., 2016b; Zeitouni et al., 2017; Dessouky et al., 2017 is based on the association of all the valid execution paths of the *Prover* with a single hash value. Intuitively, this is not a scalable approach because it does not allow to handle complex CFG/execution paths. On the contrary, a second approach Dessouky et al., 2018 relies on the transmission of all the control-flow events to the *Verifier*, which then applies a symbolic execution to validate their correctness. 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 4.3.1) together with the details of its workflow (Section 4.3.2), explicitly motivating how we address both the requirements needed to apply runtime RA on complex systems.

4.3.1 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 4.2 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.

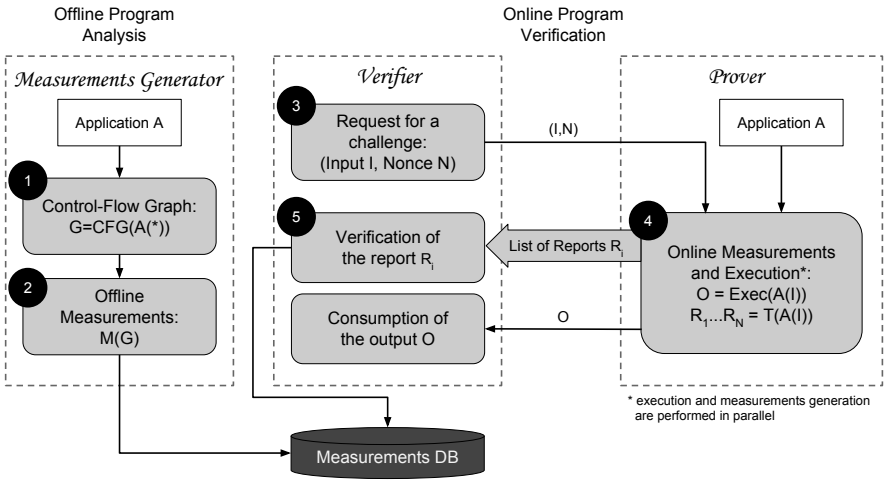


FIGURE 4.2: ScaRR system overview.

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 4.2) and, after generating all the *Application A* valid *measurements*, it saves them in the *Measurements DB* (Step 2 in Figure 4.2). During the second phase, the *Verifier* sends a challenge to the *Prover* (Step 3 in Figure 4.2). Thus, the *Prover* starts executing the *Application A* and sending partial reports to the *Verifier* (Step 4 in Figure 4.2). 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.

4.3.2 Details

As shown in Figure 4.2, 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 Costan and Devadas, 2016; Winter, 2008, can be achieved through a static RA applied on the *Prover* software stack. 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.

4.3.3 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 4.3, 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

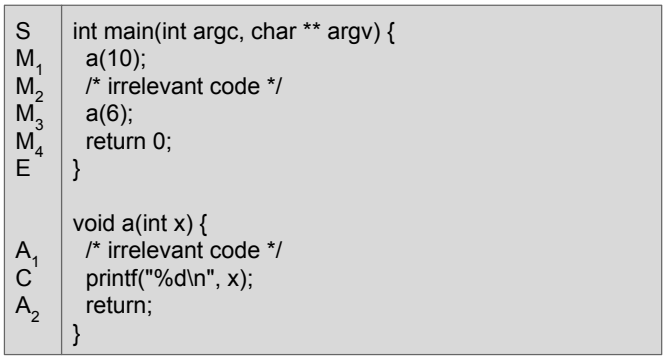


FIGURE 4.3: Illustrative example to explain the shadow stack on the ScaRR *Verifier*.

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*:

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

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 4.4 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

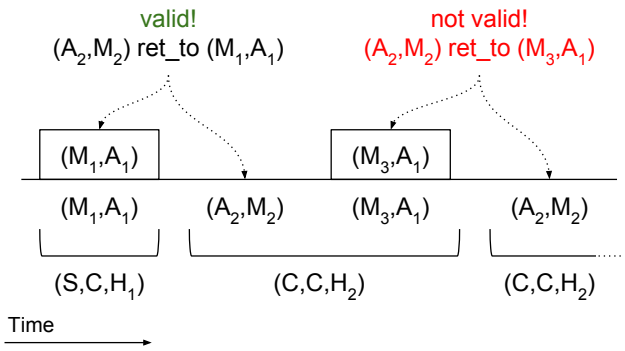


FIGURE 4.4: Implementation of the shadow stack on the ScaRR *Verifier*.

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

4.4 Implementation

Here, we provide the technical details of the ScaRR schema and, in particular, of the *Measurements Generator* (Section 4.4.1) and of the *Prover* (Section 4.4.2).

4.4.1 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) *McSema* 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 5.0.

4.4.2 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 4.5, 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 4.6). 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; *BLAKE2* 2013, which is faster and provides high cryptographic security guarantees for calculating the hash of the *LoAs*.

4.5 Evaluation

We evaluate ScaRR from two perspectives. First, we measure its performance focusing on: attestation speed (Section 4.5.1), verification speed (Section 4.5.2) and network impact (Section 4.5.3). Then, we discuss ScaRR security guarantees (Section 4.5.4).

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.

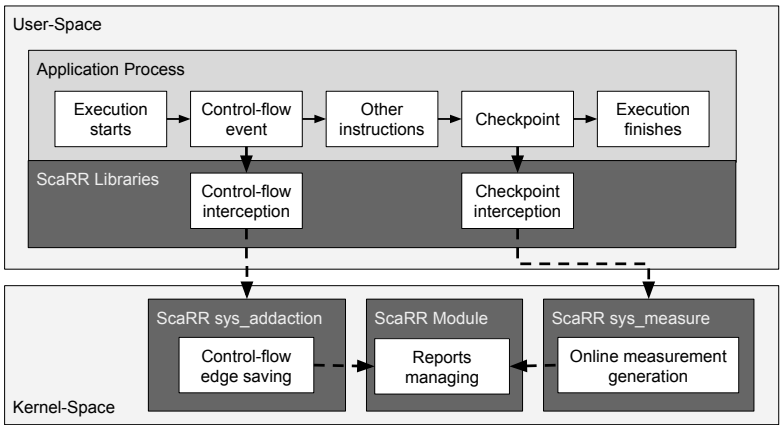


FIGURE 4.5: Internal architecture of the *Prover*.

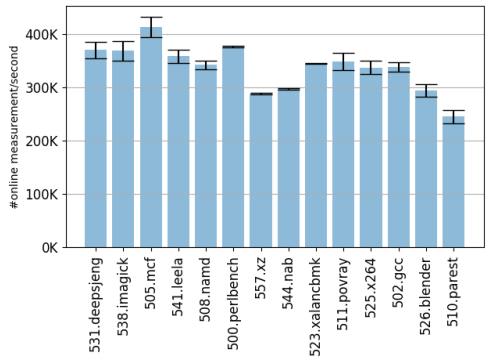
events, we then extracted the *offline measurements* and we ran each experiment to analyze a specific performance metrics.

4.5.1 Attestation Speed

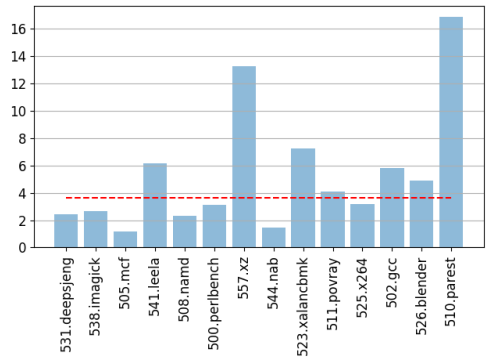
We measure the attestation speed as the number of *online measurements* per second generated by the *Prover*. Figure 4.6a shows the average attestation speed and the standard deviation for each experiment of the SPEC CPU 2017. More specifically, we run each experiment 10 times, calculate the number of *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., 2016b. 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.

4.5.2 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 Stallings, 2002 to simulate an ideal scenario in which the *Prover* is fast. 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 4.6: 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 4.6b 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., 2016b; Abera et al., 2019. Overall, Figure 4.6b 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.

4.5.3 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 4.5.2, 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

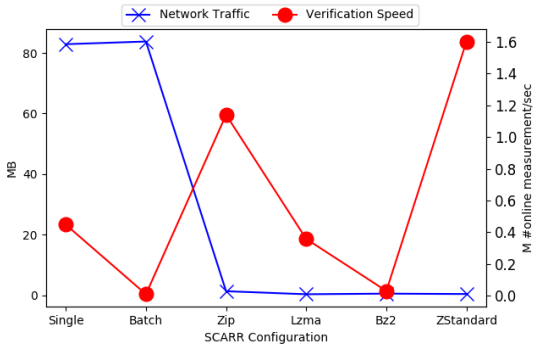


FIGURE 4.7: Comparison of different approaches for generating partial repors in terms of network traffic and verification speed.

send 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* *Zstandard* 2016. The results of this experiment are shown in Figure 4.7. 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.

4.5.4 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, 2007 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., 2015, 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*.

4.6 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) Stravers and Waerdt, 2013 at each enclave exit. This caused frequent page walks after each enclave call. A similar problem was related to the Page-Table Isolation (PTI) Watson et al., 2018 mechanism in the Linux kernel, which protects against the Meltdown vulnerability. 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 Ge, Cui, and Jaeger, 2017; Hu et al., 2018 experienced information loss.

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., 2018; Liu, Zhang, and Wang, 2018.

Chapter 5

Advanced attacks against SGX Enclaves

The solutions in 3 and 4 enhance static and runtime protection of untrusted code in memory, respectively. What happens when the attacker focuses on the TEE itself?

The answer to this question is addressed in the paper:

- SnakeGX: a sneaky attack against SGX Enclaves (ACNS 2021).

Chapter 6

A Novel Runtime Remote Attestation Schema for SGX Enclaves

The attack described in 5 requires a study of new defenses and analyses of *enclaves*. The answer to this question is addressed in the papers:

- SgxMonitor: A Novel Runtime Remote Attestation Schema for SGX Enclaves (under review).

Chapter 7

Memory forensics in SGX environment

After discussing the attacks in 5, and see the defences in 6. 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 8

Conclusion

These are the conclusions.

Appendix A

Appendix Title Here

Write your Appendix content here.

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.

Abera, Tigist et al. (2016a). "C-FLAT: Control-Flow Attestation for Embedded Systems Software". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, pp. 743–754. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978358. URL: http://doi.acm.org/10.1145/2976749.2978358.

Abera, Tigist et al. (2016b). "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/.

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

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

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.

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.

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.

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.

BLAKE2 (2013). Last access March 2019. URL: https://github.com/BLAKE2/BLAKE2.

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.

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

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

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

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

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.

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.

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.

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.

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.

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.

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. (2018). "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.

Intel (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.

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

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.

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

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.

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

Microsoft (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.

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

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

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

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

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.

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

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.

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.

Schuster, Felix et al. (2015). "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.

Shacham, Hovav (2007). "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.

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.

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.

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

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

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

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.

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, 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.

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.

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.

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.

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

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