3-2019

Careful-Packing: A practical and scalable anti-tampering software protection enforced by trusted computing

Flavio TOFFALINI

Martín OCHOA

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Jianying ZHOU

Citation

TOFFALINI, Flavio; OCHOA, Martín; SUN, Jun; and ZHOU, Jianying. Careful-Packing: A practical and scalable anti-tampering software protection enforced by trusted computing. (2019). *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY 2019), Dallas, TX, USA, 2019 March 25-27*. 231-242.

Available at: https://ink.library.smu.edu.sg/sis_research/4965

Careful-Packing: A Practical and Scalable Anti-Tampering Software Protection enforced by Trusted Computing

Flavio Toffalini
Singapore University of Technology and Design
ST Electronics-SUTD Cyber Security Laboratory
flavio_toffalini@myemail.sutd.edu.sg

Martín Ochoa
Cyxtera Technologies
ST Electronics-SUTD Cyber Security Laboratory
martin.ochoa@cyxtera.com

Sun Jun
Singapore University of Technology and Design
ST Electronics-SUTD Cyber Security Laboratory
sunjun@sutd.edu.sg

Jianying Zhou
Singapore University of Technology and Design
jianying_zhou@sutd.edu.sg

ABSTRACT

Ensuring the correct behaviour of an application is a critical security issue. One of the most popular ways to modify the intended behaviour of a program is to tamper its binary. Several solutions have been proposed to solve this problem, including trusted computing and anti-tampering techniques. Both can substantially increase security, and yet both have limitations. In this work, we propose an approach which combines trusted computing technologies and anti-tampering techniques, and that synergistically overcomes some of their inherent limitations. In our approach critical software regions are protected by leveraging on trusted computing technologies and cryptographic packing, without introducing additional software layers. To illustrate our approach we implemented a secure monitor which collects user activities, such as keyboard and mouse events for insider attack detection. We show how our solution provides a strong anti-tampering guarantee with a low overhead: around 10 lines of code added to the entire application, an average execution time overhead of 5.7% and only 300KB of memory allocated for the trusted module.

ACM Reference Format:
Flavio Toffalini, Martín Ochoa, Sun Jun, and Jianying Zhou. 2019. Careful-Packing: A Practical and Scalable Anti-Tampering Software Protection enforced by Trusted Computing. In *Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19), March 25–27, 2019, Richardson, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3292006.3300029

1 INTRODUCTION

The widespread of commercial software and of potential security threats makes it necessary to develop systematic protection mechanisms. For instance, a customer could attempt to use a program without paying the license fee [38], a player might cheat in a videogame [21], or an anti-virus software can be sabotaged [31] by malware. To achieve these goals, a common strategy is to edit the binary code of such software in order to alter its logic. These threats are often referred to as Man-At-The-End attackers (MATE) [3]. Both academic researchers and commercial companies have spent an extensive effort against MATE threats [5, 12, 16, 18, 50, 53]. The goal of the defending mechanisms is to guarantee that an attack cannot change the software logic to some extent. It is possible to achieve this goal in different ways, *e.g.,* through anti-tampering techniques [36] or through trusted computing technologies [20].

Anti-tampering techniques allow a software to inspect itself and check whether its code has been modified. We refer to those techniques as *self-checking*, which literally read the binary code of the protected software by using special functions called *checkers*. The checkers compute a digital fingerprint of the software bytecode and verify whether that fingerprint matches a pre-computed value [36]. On the other hand, trusted computing technologies provide dedicated hardware so that the software can be executed in secure containers which are physically separated from the rest of the system. Those containers are composed of memory regions that cannot be directly read/written by other processes (either from kernel-space or from user-space). Trusted computing technologies are further reinforced against physical attacks such as flashing BIOS/firmware, page swap, or page cache attacks [13].

However, both anti-tampering and trusted computing have limitations. On the one hand, purely software-based anti-tampering techniques are not completely secure, since the defending mechanisms reside in an untrusted memory region and a determined attacker can identify and disarm such defenses. It is possible to harden anti-tampering techniques by using a combination of additional approaches [6, 10, 11, 51] that raise the bar for the attackers but that do not fundamentally address the problem [22]. On the other hand, trusted computing technologies, which provide higher security guarantees than purely software-based solutions, often have practical limitations, e.g., software within a secure container cannot directly interact with the hosting operating system (OS); and the secure container often has size limitations [7]. Previous works studied solutions that move part of the OS functionality inside a trusted region [4, 7, 47], but they introduce further complexity for employing a secure interaction with the rest of the world (*e.g.,* networking, file system). Other authors suggested protecting

CODASPY '19, March 25–27, 2019, Richardson, TX, USA

ACM ISBN 978-1-4503-6099-9/19/03…$15.00
https://doi.org/10.1145/3292006.3300029

only portions of the code [30, 41]. However, these approaches do not address critical limitations such as the interaction with the underlying OS, or the limited amount of memory. Limited memory makes it unsustainable to deploy all processes in dedicated trusted containers. For instance, machines featured with Intel Software Guard eXtension (SGX) [40] provide only a few hundred megabytes that must be shared among all the running trusted containers. If we consider processes such as Skype or Firefox, which require around 100MB each, we need multiple trusted containers for each process to protect. Therefore, this approach does not scale for multiple parallel processes. The introduction of SGX 2.0 allows modifying the size of a single trusted container but it does not modify the maximum memory available for trusted containers.

Our proposal 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 stand-alone 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.

Organization. We provide background on SGX and software protection in Section 2. Then, we discuss the threat model in Section 3 and describe our approach and the technical challenges in Section 4. After that, we explain our implementation in Section 5, while we perform an evaluation of our approach in Section 6. Finally, we provide a discussion about related works in Section 7 and conclude in Section 8.

2 BACKGROUND

In the following, we recap some notions of Intel Software Guard eXtension (SGX) [40] and of anti-tampering techniques.

2.1 Software Guard eXtention Overview

Although our approach is not bounded to one particular trusted computing technology, we have chosen to develop our first proof-of-concept prototype based on Intel Software Guard eXtension (SGX) [40].

At the core of SGX are *enclaves.* An enclave is a memory region located in the user-space which contains trusted functions. A system can host multiple enclaves at the same time, each of which has a dedicated set of pages (4kB each) in DRAM. Each enclave has direct access only to its own page set and it cannot read/write pages belonged to other enclaves. Processes in neither the user-space nor the kernel-space can direct access to an enclave's pages. Figure 1 shows the SGX memory structure. This is achieved by using a *Processor Reserved Memory* layout (PRM), which reserves a subset of DRAM pages only for enclaves instantiations. The system assigns some pages of PRM to a specific enclave, this association is maintained thanks to the *Enclave Page Cache* (EPC). Finally, SGX keeps track of the status of each page assigned by a specific table called the Enclave Page Cache Map (EPCM).

SGX provides a gateway mechanism for calling trusted functions in the enclaves, which is implemented by a dedicated instruction (*ECALL*). All trusted functions are enumerated at the compilation time. Then, a process calling a trusted function will provide a pointer to the enclave along with the number of the function to call. The passage of parameters from the untrusted zone to the trusted one is implemented through a proxy mechanism: SGX provides a set of functions for moving parameters into the enclave, and also for retrieving the return values. SGX also allows pointer parameters that

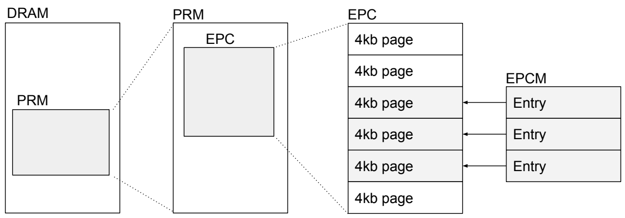

Figure 1: Intel Software Guard eXecution Memory Structure

allow a trusted function to directly read/write untrusted memory regions.

To export data outside an enclave, SGX provides a mechanism called *sealing*. Sealing works by encrypting a buffer of bytes before it leaves the trusted memory region. Through sealing, an enclave can encrypt a secret (*i.e.,* a variable) such that only the enclave itself can read it again. It is also possible to encrypt a secret such that two or more enclaves can read it. To achieve it, SGX provides an attestation process which guarantees the identification of an enclave and the hardware in which it is deployed. Sealing is used when an enclave needs to store its state in a non-volatile storage, or else when an enclave needs to share information with another one inside the same machine or over a network.

It is possible to implement multi-threading computation in SGX, *i.e.,* trusted functions, which belong to the same enclave, can be called simultaneously by threads. We refer to parallel trusted function as *trusted threads*, while the normal threads are called *untrusted threads*. The number of secure threads is statically defined at compilation time, and each (trusted) thread status is maintained by structures (TCS) within the enclave, SGX also provides secure synchronization mechanisms (*i.e.,* locks). Moreover, it is possible to use policies that guarantee an *untrusted* thread to be always bounded to the same *trusted thread*. This feature allows us to extend our approach to multi-threading programs.

As introduced before, enclaves suffer from *denial-of-service*, *e.g.,* a malicious OS might avoid execution of the trusted functions. In this work, we show how our approach can mitigate these limitations.

2.2 Anti-Tampering Techniques

We say that a program P is tamper-resistant if P is designed such that an attacker would have difficulties to modify P's code. There are several strategies for achieving this goal [36]. In this work, we focus on *self-checking*. These techniques work at bytecode level, and they are structured such that the software can read its own bytecode in order to find anomalies and then reacts accordingly. We call *checkers* those sections of the software which check the software status, and *responses* those which react to the checkers' requests.

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

To prevent the checkers from being disabled by an attacker, they typically spread over the code and/or triggered randomly during the execution. Checkers, hash functions, and hash values can be prone to attack; therefore, an anti-tampering protection must be designed for protecting itself. This is achievable by using different techniques, *e.g.,* through obfuscation techniques [6], or a network of checkers (which communicate with each other so that if one checker is disabled/tampered, other checkers become aware of the attack).

3 THREAT MODEL

In a tampering attack, the goal of an attacker is to edit the code of a victim program [12]. 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 [2, 33]. 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 [49]. Moreover, we assume the task scheduler is trusted, this is crucial to avoid a perfect synchronization of two processes (see Section 6.5).

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.

4 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.

4.1 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 4 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 [29], which is guaranteed neither by anti-tampering techniques [36] nor by trusted computing [28]. But it is tackled by tools such as [32, 46] and discussed in previous works [1, 14, 37, 52, 54].

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. [30] 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 [4, 7, 47].

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 [9, 43, 44], 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 $i \in \text{CS}$ 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 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 6.5, 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 [8] 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 [51]. 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.

4.2 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 5), we use the Intel Software Guard eXtension (SGX) [40] terminology. However, it is possible to use other trusted modules (see Section 6.6).

Unlike previous solutions that simple "hide" checking functions by adopting obfuscation or anti-reversing techniques [6, 10, 11, 51], 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 2, 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 [34] 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 2-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* [48], a technique which is often used by malware to hide its functionality, combined with a heartbeat [18]. Our intuition is to force the untrusted zone to call trusted functions in order to execute application logic. This configuration is depicted in Figure 2-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 2-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. In the beginning, the CS is encrypted (*i.e.,* $E[CS]$) while the `decrypt()` function is executed (Figure 3-1). After the decryption phase, the CS is plain (white color) and it is normally executed (Figure 3-2). Finally, the `encrypt()` function is executed and the CS gets encrypted again (Figure 3-3).

Together with the packing mechanism already explained, we employ a parallel heartbeat as a response, which is depicted in Figure 2-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).

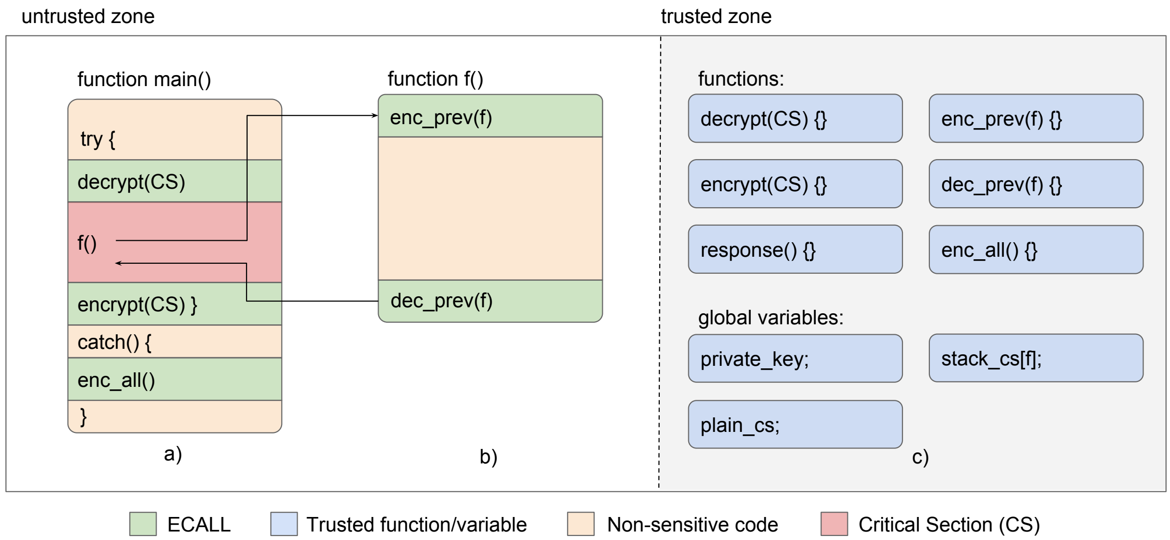

Figure 2: An overview of our schema for single-thread applications, the memory is split in trusted and untrusted zones. The trusted zone contains all methods required for our technique, while in the untrusted zone we show the interaction of those methods with the CSs.

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 2-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 2-c. These stacks are global variable inside the trusted module, we identify the stack for the function `f` as follows:

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

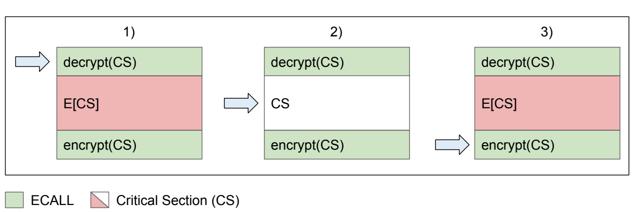

Figure 3: Packing mechanism of our schema.

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 2-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 2-a. The exception block is made such that (i) to catch all exceptions, (ii) to run `enc_all()`, (iii) to throw the exception again. In this way, we restore the anti-tampering mechanism as soon as an exception appears. Thus, after an exception, we encrypt all the plain CSs and the application can continue normally. Note that the *response* function has to be extended in order to protect the *catch* block, or else, an attacker might raise an exception in order force a CS to be plain[1].

Multi-threading programs. We can extend the previous techniques in order to handle parallel computation, this is possible because some trusted computing technologies allow multi-threading programming, like SGX (see Section 2.1). To achieve multi-threading, we maintain a *plain_cs* and a `stack_cs[f]` for each thread. Moreover, we introduce a counter for each CS. These global variables represent the number of threads which are executing a CS in a specific moment. In the beginning, the CSs' counters are set to *zero*. Then, they are increased and decreased by `decrypt()` and `encrypt()` functions respectively.

Ensuring a Secure Booting Phase. Our approach requires that the program has a secure booting phase, which means having the following assumptions for the *encrypt*, *decrypt* and *response*: the key for crypto algorithms must be loaded in a secure way together with a table which describes where the CSs are located (*i.e.,* their address and length) with their hash values. We refer to this table as *block table*. We assume a trusted loading of this information by adopting SGX sealing and attestation mechanisms. Those mechanisms ensure to store information on a disk or to establish a secure channel with other enclaves within the same machine (*i.e.,* local) or with a remote one (*i.e.,* remote) in a trusted way. Details on sealing and attestation are discussed in Section 2.1.

5 IMPLEMENTATION

In this section, we describe a proof-of-concept implementation of our anti-tampering technique, whose architecture is depicted in Figure 4. 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) [40], however, it is possible to use other solutions that involves the kernel (*e.g.,* TPM [26]). 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.

5.1 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) [45], however, it is possible to deduct the right address offset by employing a Drawbridge system [39].

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 5, the application communicates with the untrusted DLL to call the functions described in Section 4. The untrusted DLL works together with the trusted DLL (*i.e.,* the enclave) to handle the whole anti-tampering technique. We choose this architecture to simplify the integration of our anti-tampering system. In this way, the developer can focus on the main program and integrate the anti-tampering system later. Each component of the architecture is described as follows:

- **Application:** this is the client that we aim to enforce. Natively, it contains all the functionalities for collecting information from the underline OS and ship them to the server.
- **Untrusted DLL:** this contains the untrusted functions for interacting with the enclave. Also, it keeps track of the status of the enclave (*i.e.,* enclave pointer) and exposes routines procedures.
- **Trusted DLL (enclave):** this is the enclave. It contains the trusted functions described in Section 4 (*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

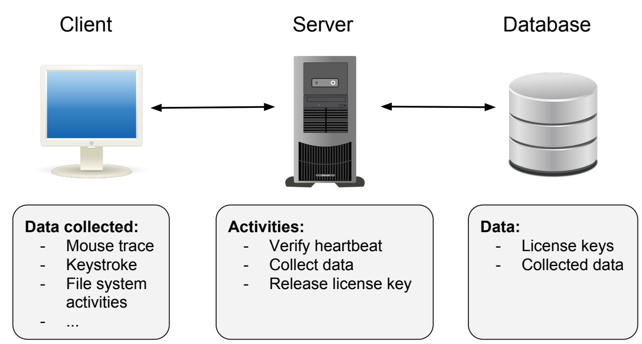

Figure 4: The architecture of proof-of-concept program. The client is a monitoring agent which collects user's activities, the server handles clients, and the database stores collect data and license keys.

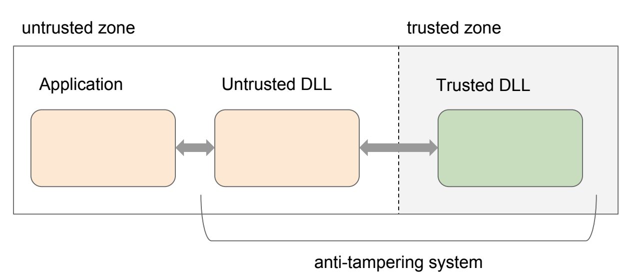

Figure 5: The software organization of the client.

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

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 [55] between the assembly code and the license key. SGX natively provides an implementation of this algorithm [25].

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.

5.2 Installation Phase

We achieve a secure installation by using an authentication protocol based on SGX remote attestation, the entire protocol is depicted in Figure 6. 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.

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 [24] (Figure 6-0). 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 6-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 6-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 6-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.

6 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.

6.1 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 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.

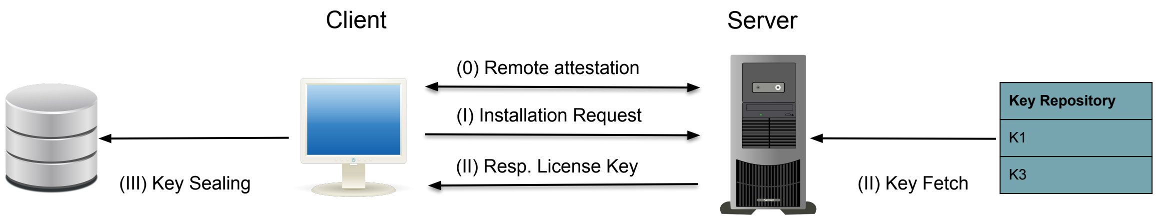

Figure 6: Secure installation protocol between client and server.

Table 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%

6.2 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 5). 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 7. 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.

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 8. 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.

6.3 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

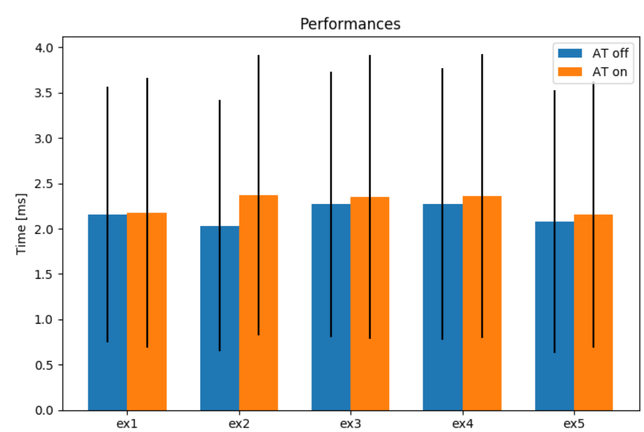

Figure 7: Average response time with and without anti-tampering technique

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.

6.4 Threat Mitigation

We explain how our approach mitigates threats according to the attacker model described in Section 3.

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

Protection against disarm. An attacker can always disarm a function by removing its invocation. Moreover, SGX is prone to *denial-of-service attacks* due to its nature (see Section 2.1). We protect trusted invocations by adopting the packaging tactic discussed in Section 4. 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), it is still possible to perform attacks from the user-space [19]. 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 [27]. To achieve an *on-line* tampering, as introduced in Section 3, 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.

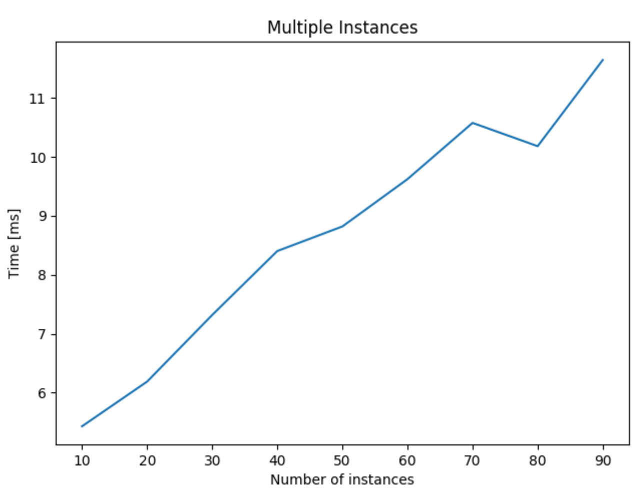

Figure 8: Average response time with multiple instances

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 6.5.

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 [23]. 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) 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 5, 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.

6.5 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, 70GHz 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.

6.6 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.

7 RELATED WORK

We considered works which deal with to Intel Software Guard eXtension (SGX) [40] and anti-tampering techniques.

Anti-tampering techniques Morang et al. [35], Ghost et al. [18], and Dewan et al. [15], base their anti-tampering techniques on hyper-visor level. In all works, authors rely trustiness on the hyper-visor, while we propose a *self-checking* mechanism which is built on top of a trusted module. Feng et al. [17] propose an anti-cheating mechanism for video-games. In their approach, they simulating client logic on a server to identify inconsistencies. Unlike them, we spot client anomalies by using a *self-checking* system.

Viticchié et al. [51] developed an anti-tampering mechanism which is based on a remote attestation. Here, the client is moved to a trusted server. Their approach is substantially different than ours because they do not rely on a trusted module. Also, their mechanism forces an application to be partially moved to a server, while we do not alter client structure.

Commercial anti-tampering solutions for video-games, such as [16, 50], perform a software signature which communicates with a trusted server; however, they do not consider trusted computing for improving software protection.

Software protection by using SGX The first strategy for protecting applications by using SGX was moving the whole code inside an enclave. This was proposed by Baumann et al. [7], and by Tsai et al. [47] after. They respectively developed Haven (for Windows) and Graphene (for Linux), which are tools that allow one to execute legacy applications inside an enclave. Their attacker model is the Iago attacker, which consider the host OS as malicious, while our attacker model aims at modifying code at runtime from user-space. These tools contain a micro-kernel inside an enclave that communicates through a Drawbridge system with the host OS [39]. A spin in this direction was proposed by Seo et al. [42], who extend the project Graphene by adding Address Space Layout Randomization (ASLR) features. Arnautov et al. [4], instead, deployed previous approaches to Docker containers in their tool SCONE. All these systems have a common approach: they run the whole application in an enclave. However, they need to introduce some compromise: Haven and Graphene need specific libraries for the OS. Moreover, they limit applications' features (*e.g.,* Haven does not allow graphical interfaces, SCONE do not support `fork()` operations), also, they limit application size because of the enclave limitation. Our approach, instead ,can protect a vast code surface exploiting few code lines in the enclave without any additional library or limitation in term of features.

Another approach for SGX technology is to move into enclave only those parts of the application which are considered secure-sensitive. This approach was adopted by Schuster et al. [41], who combined MapReduce framework and SGX enclaves to perform big data analysis. In this work, authors define ad-hoc enclave for their application. This approach was then automatized by Joshua et al.[30], who proposed a tool (Glamdring) that moves critical sections (and variables) into automatically generated enclaves. Our approach is different because we protect critical sections without moving them inside enclaves.

8 CONCLUSION

In this work, we presented a novel anti-tampering technique which leverages trusted computing technologies. We achieved this by adopting a packing strategy that is similar to the one used by malware to hide its functionality. Our approach forces a program to call trusted functions in order to be executed properly (by unpacking a piece of software from a trusted container).

We implemented a proof-of-concept prototype of our technique by using Intel Software Guard eXtension (SGX) [40] technology. We illustrated our approach by protecting an agent that was designed to collect user's event and ship them to a central server. Through this implementation, we showed how our architecture can guarantee further security properties such as a secure installation and a continuous client monitoring.

Using our prototype, we measured the overhead in terms of lines of code (less than 10 lines), execution time (on average 5.7% more), and space required for the trusted container (300KB). In sum, our approach results in a scalable and practical software protection solution.

ACKNOWLEDGMENTS

This research was supported by ST Engineering Electronics and National Research Foundation, Prime Minister's Office Singapore, under Corporate Laboratory @ University Scheme (Programme Title: STEE Infosec - SUTD Corporate Laboratory). This work was also partly supported by SUTD start-up research grant SRG-ISTD-2017-124.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[2] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 743–754. https://doi.org/10.1145/2976749.2978358

[3] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. 2015. Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications* 48 (2015), 44 – 57. https://doi.org/10.1016/j.jnca.2014.10.009

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *OSDI*. 689–703.

[5] Sebastian Banescu, Mohsen Ahmadvand, Alexander Pretschner, Robert Shield, and Chris Hamilton. 2017. Detecting Patching of Executables without System Calls. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 185–196.

[6] Sebastian Banescu and Alexander Pretschner. 2017. A tutorial on software obfuscation. *Advances in Computers* (2017).

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[8] Philippe Biondi and Fabrice Desclaux. 2006. Silver needle in the Skype. *Black Hat Europe* 6 (2006), 25–47.

[9] David Brumley and Dawn Song. 2004. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*. 57–72.

[10] Hoi Chang and Mikhail J Atallah. 2001. Protecting software code by guards. In *Digital Rights Management Workshop*, Vol. 2320. Springer, 160–175.

[11] Ping Chen, Christophe Huygens, Lieven Desmet, and Wouter Joosen. 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, 323–336.

[12] Christian S. Collberg and Clark Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering* 28, 8 (2002), 735–746.

[13] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[14] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection.. In *USENIX Security Symposium*, Vol. 2014.

[15] Prashant Dewan, David Durham, Hormuzd Khosravi, Men Long, and Gayathri Nagabhushan. 2008. A Hypervisor-based System for Protecting Software Runtime Memory and Persistent Storage. In *Proceedings of the 2008 Spring Simulation Multiconference (SpringSim '08)*. Society for Computer Simulation International, San Diego, CA, USA, 828–835. http://dl.acm.org/citation.cfm?id=1400549.1400685

[16] Evenbalance. 2017. PunkBuster. http://www.evenbalance.com/pbsetup.php Last visit on 13 Nov 2017.

[17] Wu-chang Feng, Ed Kaiser, and Travis Schluessler. 2008. Stealth Measurements for Cheat Detection in On-line Games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '08)*. ACM, New York, NY, USA, 15–20. https://doi.org/10.1145/1517494.1517497

[18] Sudeep Ghosh, Jason D Hiser, and Jack W Davidson. 2010. A secure and robust approach to software tamper resistance. In *International Workshop on Information Hiding*. Springer, 33–47.

[19] D. Gullasch, 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*. 490–505. https://doi.org/10.1109/SP.2011.22

[20] Walter Allen Helbig Sr. 1998. Trusted computer system. US Patent 5,841,868.

[21] G Hoglund. 2006. Hacking world of warcraft: An exercise in advanced rootkit design. *Black Hat* (2006).

[22] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E Tarjan. 2001. Dynamic self-checking techniques for improved tamper resistance. In *ACM Workshop on Digital Rights Management*. Springer, 141–159.

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

[24] Intel. 2016. Remote (Inter-Platform) Attestation. https://software.intel.com/en-us/node/702984 Last visit on 6 Dec 2017.

[25] Intel. 2018. Rijndael AES-GCM encryption API. https://software.intel.com/en-us/node/709139 Last visit on 10 Mar 2017.

[26] ISO. 2015. ISO/IEC 11889-1:2015. https://www.iso.org/standard/66510.html Last visit 13 Nov 2017.

[27] Kernel.org. 2018. CFS Scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt Last visit on 20 Aug 2018.

[28] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*. 523–539.

[29] J. Li, X. Tong, F. Zhang, and J. Ma. 2018. Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. *IEEE Transactions on Information Forensics and Security* 13, 6 (June 2018), 1535–1550. https://doi.org/10.1109/TIFS.2018.2797932

[30] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan OâĂŹKeeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic application partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 24.

[31] Cybellum Technologies LTD. 2017. Double Agent. https://github.com/Cybellum/DoubleAgent Laste visit 15 Nov 2017.

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

[33] Microsoft. 2017. Driver Signing. https://docs.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing Last visit on 02 Mar 2018.

[34] Anders Møller and Michael I Schwartzbach. 2012. Static program analysis.

[35] B. Morgan, ÃĽ. Alata, V. Nicomette, M. KÃćaniche, and G. Averlant. 2015. Design and Implementation of a Hardware Assisted Security Architecture for Software Integrity Monitoring. In *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*. 189–198. https://doi.org/10.1109/PRDC.2015.46

[36] Jasvir Nagra and Christian Collberg. 2009. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education.

[37] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 49–58.

[38] onhax.me. 2017. Bypass license in Android. https://onhax.me/bypass-license-verification-failed Last visit on 15 Nov 2017.

[39] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. 2011. Rethinking the library OS from the top down. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 291–304.

[40] Carlos Rozas. 2013. Intel® Software Guard Extensions (Intel® SGX). (2013).

[41] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 38–54.

[42] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.

[43] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. 2006. Reducing TCB complexity for security-sensitive applications: Three case studies. In *ACM SIGOPS Operating Systems Review*, Vol. 40. ACM, 161–174.

[44] Scott F Smith and Mark Thober. 2006. Refactoring programs to secure information flows. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*. ACM, 75–84.

[45] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 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, 574–588.

[46] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM.. In *USENIX Security Symposium*. 941–955.

[47] Chia-Che Tsai, 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*.

[48] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2016. Rambo: Run-time packer analysis with multiple branch observation. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 186–206.

[49] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.

[50] Valve. 2017. Valve Anti-Cheat System (VAC). https://support.steampowered.com/kb_article.php?p_faqid=370 Last visit 13 Nov 2017.

[51] Alessio Viticchié, Cataldo Basile, Andrea Avancini, Mariano Ceccato, Bert Abrath, and Bart Coppens. 2016. Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks. In *Proceedings of the 2016 ACM Workshop on Software PROtection*. ACM, 73–84.

[52] Zhi Wang 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, 380–395.

[53] Windows. 2012. Windows Media DRM 10 (Janus). http://www.samsung.com/ca/support/skp/faq/20189 Last visit on 15 Nov 2017.

[54] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries.. In *USENIX Security Symposium*. 337–352.

[55] Gang Zhou, 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, 185–192.