Practical Enclave Malware with Intel SGX

Michael Schwarz, Samuel Weiser, Daniel Gruss

Graz University of Technology

Abstract. Modern CPU architectures offer strong isolation guarantees towards user applications in the form of enclaves. However, Intel's threat model for SGX assumes fully trusted enclaves and there doubt about how realistic this is. In particular, it is unclear to what extent enclave malware could harm a system. In this work, we practically demonstrate the first enclave malware which fully and stealthily impersonates its host application. Together with poorly-deployed application isolation on personal computers, such malware can not only steal or encrypt documents for extortion but also act on the user's behalf, e.g., send phishing emails or mount denial-of-service attacks. Our SGX-ROP attack uses new TSX-based memory-disclosure primitive and a write-anything-anywhere primitive to construct a code-reuse attack from within an enclave which is then inadvertently executed by the host application. With SGX-ROP, we bypass ASLR, stack canaries, and address sanitizer. We demonstrate that instead of protecting users from harm, SGX currently poses a security threat, facilitating so-called super-malware with ready-to-hit exploits. With our results, we demystify the enclave malware threat and lay ground for future research on defenses against enclave malware.

Keywords: Intel SGX, Trusted Execution Environments, Malware

1 Introduction

Software isolation is a long-standing challenge in system security, especially if parts of the system are considered vulnerable, compromised, or malicious [24]. Recent isolated-execution technology such as Intel SGX [23] can shield software modules via hardware protected enclaves even from privileged kernel malware. Thus, SGX has been advertised as key enabler of trusted cloud computing, where customers can solely rely on the CPU hardware for protecting their intellectual property and data against curious or malicious cloud providers [47]. Another use case for SGX is protecting copyrighted material from piracy [5, 40] (DRM). Also, enclaves are explored for various other use cases, such as crypto ledgers [39], wallets [39], password managers [58] and messengers [36]. With the upcoming SGXv2 [23], Intel opens their technology for the open-source community, allowing to bypass Intel's strict enclave signing policy via their own key infrastructure.

However, there is a flip side to the bright future of isolated-execution technology painted by both the industry and community. Any isolation technology might also be maliciously misused. For instance, virtual machine extensions have been used to hide rootkits [29, 37] and exploit CPU bugs [60]. Researchers have warned

that enclave malware likely causes problems for today's anti-virus (AV) technology [14, 17, 46]. The strong confidentiality and integrity guarantees of SGX fundamentally prohibit malware inspection and analysis, when running such malware within an enclave. Moreover, there is a potential threat of next-generation ransomware [34] which securely keeps encryption keys inside the enclave and, if implemented correctly, prevents ransomware recovery tools. Although there are few defenses proposed against potential enclave malware, such as analyzing enclaves before loading [14] or inspecting their I/O behavior [14, 17], they seem too premature to be practical [34]. Unfortunately, there exist no practical defenses against enclave malware, partly due to the lack of a proper understanding and evaluation of enclave malware.

(Im-)Practicality of Enclave Malware. Is enclave malware impractical anyway due to the strict enclave launch process [26], preventing suspicious enclave code from getting launch permission? It is not, for at least four reasons: First, adversaries would only distribute a benign-looking loader enclave, receiving and decrypting malicious payloads at runtime [34, 46]. Second, Intel does not inspect and sign individual enclaves but rather white-lists signature keys to be used at the discretion of enclave developers for signing arbitrary enclaves [26]. Enclave developers might intentionally add malware to their legitimate enclaves, e.g., to support their DRM actitivies as Sony did in the early 2000s with their rootkit on millions of CDs [45]. In fact, we have a report from a student who independently of us found that it is easy to go through Intel's process to obtain such signing keys. Third, the flexible launch control feature of SGXv2 allows bypassing Intel as intermediary in the enclave launch process [23]. Fourth, by infiltrating the development infrastructure of *any* enclave vendor, be it via targeted attacks or nation state regulations, malware could be piggy-backed on their benign enclaves. Hence, there are multiple ways to make enclave malware pass the launch process, with different levels of sophistication.

Impact of Enclave Malware. Researchers have practically demonstrated enclave spyware stealing confidential information via side channels [48]. Apart from side-channel attacks, Costan et al. [14] correctly argues that enclaves cannot do more harm to a system than an ordinary application process. Yet, malware typically performs malicious actions from within an ordinary application process. As an example, Marschalek [34] demonstrated enclave malware which requires support of the host application to perform its malicious actions (*i.e.*, ransomware and shellcode). No prior work has practically demonstrated enclave malware attacking a benign host application that does not collude with the enclave. Hence, researchers believe that limitations in the SGX enclave execution mode severely restricts enclave malware in practice: "Everyone's first reaction when hearing this, is '*OMG bad guys will use it to create super malware!*'. But it shouldn't be that scary, because: Enclave programs are severely limited compared to normal programs: they cannot issue syscalls nor can they perform I/O operations directly." [4] Consequently, an enclave is believed to be limited by what its hosting application allows it to do: "analyzing an application can tell you a lot about what an enclave can do to a system, mitigating the fear of a '*protected malicious*

code running inside an enclave'." [1] At first glance, these statements seem reasonable, since syscalls are an essential ingredient for malware and enclaves can only issue syscalls through their host application. For example, Marschalek [34] implemented enclave malware via a dedicated syscall proxy inside the host application to forward malicious enclave actions to the system.

In this work, we expand the research on enclave malware by presenting stronger enclave malware attacks. As we show, enclave malware can overcome the SGX limitations. To that end, we develop a prototype enclave which actively attacks its benign host application in a stealthy way. We devise novel techniques for enclaves probing their host application's memory via Intel TSX. We find that enclave malware can effectively bypass any host application interface via code-reuse attacks, which we dub SGX-ROP. Thus, the attacker can invoke arbitrary system calls in lieu of the host process and gain arbitrary code execution. This shows that enclaves can escape their limited SGX execution environment and bypass any communication interface prescribed by their host.

We identify the core problem of research on enclave malware in a vagueness about the underlying threat model, which we seek to clarify in this work. Intel's SGX threat model only considers *fully trusted* enclaves running on an *untrusted* host, which fits many scenarios like [3, 9, 51, 57]. However, the asymmetry in this threat model ignores many other real-world scenarios, where enclaves might not be unconditionally trustworthy. In particular, while the (third-party) enclave vendor might consider its own enclave trustworthy, the user or the application developer that use a third-party enclave both have all rights not to trust the enclave. To address this asymmetry, we introduce a new threat model which specifically considers untrusted enclaves. This allows to reason about attacks from within enclaves, such as, e.g., enclave malware, and to identify scenarios under which potential enclave malware becomes decisive.

Contributions. We summarize our contributions as follows.

1. We introduce a new threat model which considers malicious enclaves.
2. We discover novel and stealthy TSX memory probing primitives.
3. We present SGX-ROP, a practical technique for enclave malware to perform malicious operations, e.g., on the system level, *without* collaboration from the host application.

The rest of the paper is organized as follows. Section 2 provides background. Section 3 describes our threat model. Section 4 overviews our attack. Section 5 shows how to locate gadgets, and Section 6 shows how to use them. Section 7 evaluates our attack. Section 8 provides a discussion. Section 9 concludes.

2 Background

In this section, we overview address spaces, Intel SGX, TSX as well as control-flow attacks and trigger-based malware.

Virtual Address Space. Modern operating systems rely on virtual memory as an abstraction layer to the actual physical memory. Virtual memory forms the basis for process isolation between user applications and towards the kernel.

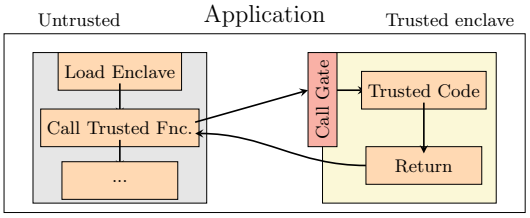

Fig. 1: In the SGX model, applications consist of an untrusted host application and a trusted enclave. The hardware prevents any direct access to the enclave code or data. The untrusted part uses the `EENTER` instruction to call enclave functions that are exposed by the enclave.

Permissions are set on a granularity of pages, which are usually 4 KB. Permissions include *readable*, *writable*, *executable*, and *user accessible*. On modern x86-64 CPUs, the usable virtual address space is 2^{48} bytes, divided into user space and kernel space, with 2^{47} bytes each. Within the user space of an application, the entire binary, as well as all shared libraries used by the application, are mapped.

Intel SGX. Intel SGX is an instruction-set extension for protecting trusted code, introduced with the Skylake microarchitecture [23]. Applications are split into untrusted and trusted code, where the latter is executed within an enclave. The threat model of SGX assumes that the enclave environment, *i.e.*, operating system and all normal application code, might be compromised or malicious and cannot be trusted. Hence, the CPU guarantees that enclave memory cannot be accessed from any other part of the system, except for the code running inside the enclave. Enclaves can therefore safely run sensitive computations, even if the operating system is compromised by malware. Still, memory-safety violations [31], race conditions [59], or side channels [8, 48] might lead to exploitation.

The integrity of an enclave is ensured in hardware by measuring the enclave loading process and comparing the result with the reference value specified by the enclave developer. Once loaded, the application can invoke enclaves only at defined entry points. After the enclave finishes execution, the result of the computation, and the control flow, is handed back to the calling application. Figure 1 illustrates the process of invoking a trusted function inside an enclave.

Enclave memory is mapped in the virtual address space of its host application. To allow data sharing between the enclave and host application, the enclave is given full access to the entire address space of the host application. This protection is not symmetric and gives rise to enclave malware.

Hardware Transactional Memory. Hardware transactional memory attempts to optimize synchronization primitives with hardware support. The hardware provides instructions to create so-called *transactions*. Any memory access performed within the transaction is not visible to the outside until the transaction is successfully completed, thus providing atomicity for memory accesses. However, if there is a conflict within the transaction, the transaction aborts and all changes done within the transaction are rolled back, *i.e.*, the previous state

is restored. A conflict can be the concurrent modification of a data value by another thread, or an exception, e.g., a segmentation fault.

Intel TSX is an instruction-set extension implementing transactional memory by leveraging the CPU cache. TSX works on a cache line granularity, which is usually 64 B. The CPU keeps track of a so-called read and write set. If a cache line is read or written inside the transaction, it is automatically added to the read or write set, respectively. Concurrent modifications to data in a read or write set from different threads cause a transaction to abort. The size of the read set, *i.e.*, all memory locations read inside a transaction, appears to be limited by the size of the L3 cache, and the size of the write set, *i.e.*, all memory locations modified inside a transaction, appears to be limited by the size of the L1 cache [33].

Transactional memory was described as a potential security feature, e.g., by Liu et al. [33] to detect rootkits, and by Guan et al. [21] to protect cryptographic keys when memory bus and DRAM are untrusted. Kuvaiskii et al. [30] showed that TSX can be leveraged to detect hardware faults and revert the system state in such a case. TSX also opens new attack vectors, e.g., by Jang et al. [27] abusing the timing of suppressed exceptions to break KASLR.

Control-Flow Attacks. Modern CPUs prevent code-injection attacks by marking writable pages as non-executable [23]. Thus, an attacker has to resort to *code-reuse attacks*. Shacham et al. [49] presented return-oriented programming (ROP), which abuses the stack pointer to control the instruction pointer. For this purpose, addresses of *gadgets*, *i.e.*, very short code fragments ending with a `ret` instruction are injected into the stack. Whenever executing `ret`, the CPU pops the next gadget address from the stack and continues execution at this gadget. Stitching together multiple gadgets enables arbitrary code execution.

A mitigation against these attacks present in modern operating systems is to randomize the virtual address space. Address space layout randomization (ASLR) [41] ensures that all regions of the binary are at random locations every time the binary is executed. Thus, gadget addresses are unpredictable, and an attacker cannot reliably reference gadgets anymore. Assuming no information leak and a large enough entropy, ROP attacks become infeasible, as addresses cannot be guessed [54, 55]. Furthermore, some techniques are deployed against such attacks, e.g., stack canaries [15, 42], shadow stacks [13], stack-pivot defenses [61].

Trigger-based Malware. With increasing connectivity between computer systems in the past decades, malware evolved into a significant security threat. There is a market for malware with various targets [12, 20]. In many cases, malware remains in an inactive state, until a specific time [16] or a remote command triggers activation [2, 50]. This decorrelates attack from infection and enables synchronized attacks as well as targeted attacks (e.g., activating the malware only on certain target systems).

The entry point for malware is often a vulnerability, whose exploitation (e.g., via a control-flow attack) enables malicious operations on the target device. While userspace malware then typically misuses lax privilege management of commodity operating systems to access user documents or impersonate user action, more sophisticated malware seeks to elevate privileges even further.

Exploits can rely on an undisclosed vulnerability [12, 19], making it very difficult to mitigate attacks. For certain actors, there is an interest in keeping such zero-day exploits undisclosed for a long time [22]. As a consequence, modern malware is obfuscated to remain stealthy [62], e.g., via code obfuscation [50], or steganography [2]. However, a thorough malware analysis may revert obfuscation [28] and expose the underlying vulnerability.

Concurrent to our work, Borello et al. [7] also proposed to use ROP chains to hide trigger-based malware. Also closely related to our work, is the use of Intel's TPM to cloak malware [18]. However, due to the different design goals, the TPM is more powerful than an SGX enclave.

3 Threat Model

In this section, we show limitations of the SGX threat model regarding malicious enclaves and present our threat model considering enclave malware.

3.1 Intel's SGX Threat Model

In Intel's SGX threat model, the entire environment, including all non-enclave code is untrusted (cf. Section 2). Such a threat model is primarily useful for cloud computing with sensitive data if a customer does not fully trust the cloud provider, and for protection of intellectual property (e.g., DRM), secret data or even legacy applications inside containers [3, 9, 51, 57]. With SGX, developers can use enclaves without the risk of exposing sensitive enclave data.

However, this model provides no means to protect other software, apart from enclaves themselves. In particular, applications hosting enclaves are not protected against the enclaves they load. Furthermore, enclaves cannot be inspected if they choose to hide their code, e.g., using a generic loader. This asymmetry may foster enclave malware, as SGX can be abused as a protection mechanism in addition to obfuscation and analysis evasion techniques. One could argue that host applications themselves could be protected using additional enclaves. However, this is not always feasible and even impossible for certain code. Some reasons for keeping application code outside enclaves are the restricted feature set of enclaves (e.g., no syscalls), expensive encrypted enclave-to-enclave communication, and an increased porting effort. Hence, there are many practical scenarios, as we illustrate in which a host application might be threatened by an enclave, which are not covered by Intel's threat model.

3.2 Our Threat Model Considering Enclave Malware

Victim. In our threat model, we assume that a user operates a computing device which is the target of an attacker. The user might be a private person, an employee or a system administrator in a company. From the user's perspective, the system (including the operating system) is considered trusted and shall be

protected against malware from third-party software. The device has state-of-the-art anti-malware or anti-virus software installed for this purpose. This applies to virtually all Windows systems today, as Windows 10 comes with integrated anti-virus software. The user executes a benign application which depends on a potentially malicious (third-party) enclave. The benign host application communicates with the enclave through a tight interface (e.g., a single ECALL). This interface, if adhered to, would not allow the enclave to attack the application. Furthermore, we assume that the host application is well-written and free of software vulnerabilities. Also, the application incorporates some state-of-the-art defenses against runtime attacks such as ASLR and stack canaries.

Attacker. The attacker controls the enclave used by the host application, which we denote as the malicious enclave. The attacker seeks to escape the enclave and gain arbitrary code execution with host privileges. Also, the attacker wants to achieve plausible deniability until he chooses to trigger the actual exploitation, *i.e.*, the exploit should be undetectable until executed. This decouples infection from exploitation and allows the attacker to mount large-scale synchronous attacks (e.g., botnets, ransomware) or target individuals. To that purpose, the attacker encloses malware in the enclave in a way that prevents inspection by any other party. This can be done by receiving and decrypting a malicious payload inside the enclave at runtime via a generic loader [46], for example.

While the attacker can run most unprivileged instructions inside the enclave, SGX not only prevents enclaves from executing privileged instructions but also syscalls, among others [23, 24]. Moreover, enclaves can only execute their own code. An attempt to execute code of its host application (e.g., by using `jmp`, or `call`), results in a general protection fault, and, thus, termination of the enclave [24]. Thus, a successful attack must bypass these restrictions. Finally, we assume that the attacker does not exploit potential hardware bugs in the SGX implementation (e.g., CVE-2017-5691).

Scenarios. We consider three scenarios, two with a criminal actor and one with a surveillance state agency [10, 20]. In the first scenario, a criminal actor provides, e.g., a computer game requiring to run a DRM enclave, or a messenger app requiring to run an enclave for security mechanisms [36]. In the second, a criminal actor provides an enclave that provides an interesting feature, e.g., a special decoder, and can be included as a third-party enclave. These scenarios are realistic, given that Sony intentionally shipped malware on millions of CDs installing rootkits throughout the early 2000s [45]. In the last scenario, it may be an app the state endorses to use, e.g., an app for legally binding digital signatures which are issued by an enclave, or legal interactions with authorities. Also, in some countries, state agencies might be able to force enclave vendors to sign malicious enclaves on their behalf via appropriate legislation, e.g., replacing equivalent benign enclaves. In any case, the externally controlled enclave might perform unwanted actions such as espionage or hijacking of the user's computer.

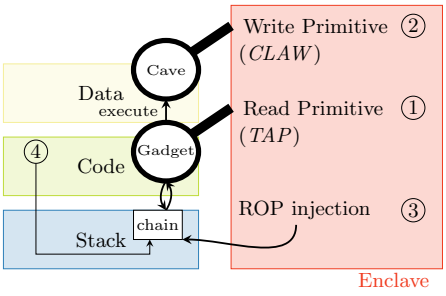

Fig. 2: Attack overview

4 Attack Overview

In this section, we outline how enclave malware can successfully attack a system using novel methods we discover. In particular, we show how enclave malware can evade all restrictions SGX poses on enclave execution. This allows the enclave to run arbitrary code disguised as the host process, similar to process hollowing [32], which is often used by malware. In fact, one can conceal existing user-space malware inside an SGX enclave, e.g., ransomware.

Restricted Enclave Environment. In contrast to most traditional malware, a malicious enclave has to act blindly. SGX prevents enclaves from directly executing syscalls (cf. Section 3), an essential ingredient for user-space malware, and mandates the host application with this task. Also, the memory layout of the host application as well as its code base might be unknown to the enclave. Note that one enclave can be loaded by different host applications. Enclaves only have knowledge of the ECALL/OCALL interface through which they communicate with the host. Hence, the malicious enclave needs to assemble an attack without knowledge of the host application memory and without executing syscalls.

Novel Fault-Resistant Primitives. To overcome these restrictions, we leverage TSX and SGX to construct a *fault-resistant read* primitive as well as a *fault-resistant write-anything-anywhere* primitive. While the read primitive helps in scanning host memory, the write primitive identifies writable memory which we denote as a cave. Those primitives are fault resistant in the sense that the enclave can safely probe both mapped and unmapped memory without triggering exception behavior that would abort the enclave. By combining both primitives, the attacker can mount a code-reuse attack (*i.e.*, ROP) on the host application, which we call SGX-ROP.

SGX-ROP. The actual SGX-ROP attack is performed in four steps, as depicted in Figure 2. In step ①, the malicious enclave uses the read primitive to scan the host application for usable ROP gadgets. In step ②, the enclave identifies writable memory caves via the write primitive and injects arbitrary malicious payload into those caves. In step ③, the enclave constructs a ROP chain from the gadgets identified in ① and injects it into the application stack. Then, the enclave returns execution to the host application and the attack waits to be activated. When the application hits the ROP chain on the stack, the actual

exploitation starts (step ④). The ROP chain runs with host privileges and can issue arbitrary system calls. While this is already sufficient for many attacks, we go one step further and execute arbitrary attack code in the host application by marking the cave (cf. step ②) as executable and invoking the code stored in the cave. After exploitation, the cave can eliminate any traces in the host application and continue normal program execution.

SGX-ROP works without the necessity of a software bug in the host application. The write primitive further allows to even bypass some anti-control-flow-diversion techniques (cf. Section 2) as any writable data can be modified. This includes ASLR, stack canaries, and address sanitizer, which we all bypass with our attack (cf. Section 7.3).

5 Locating Code Gadgets

In this section, we show how an enclave attacker can stealthily scan its host application for ROP gadgets. The attacker does not need any a-priori knowledge about the host application memory layout. We first discuss why existing memory scanning techniques are not applicable. Next, we show how to use TSX to construct a novel fault-resistant memory disclosure primitive. Finally, we leverage this primitive to discover accessible code pages of the host application and subsequently leak the host application binary. This enables an attacker to search for ROP gadgets to construct the actual attack (cf. Section 6).

5.1 Problem Statement

The malicious enclave wants to scan host application memory to craft an SGX-ROP attack. Luckily for the attacker, the SGX memory protection is asymmetric. That is, SGX prevents non-enclave code from accessing enclave memory, while an enclave can access the entire memory of the host application as they run in the same virtual address space. Thus, the enclave naturally has a read primitive. However, the enclave might not know anything about the host application's memory layout (e.g., which pages are mapped, or their content), apart from the ECALL/OCALL interface. The enclave cannot query the operating system for the host memory mapping (e.g., via `/proc/pid/maps`), as system calls cannot be performed from within an enclave. The enclave could naively try to read arbitrary host memory. However, if the accessed memory is not accessible, *i.e.*, the virtual address is invalid for the application, this raises an exception and terminate enclave execution. Hence, it is a challenge to obtain host address-space information stealthily from within an enclave. To remain stealthy and avoid detection, the enclave needs a fault-resistant memory disclosure primitive. Even with blind ROP [6], fault resistance may be necessary as pages are mapped on demand, and pagefaults would give away the ongoing attack. **Achieving Fault Resistance.** For normal applications, fault resistance can be achieved by installing a user-space signal handler (on Linux) or structured exception handling (on Windows). Upon an invalid memory access, the operating system delegates

exception handling to the registered handler. Again, this is not possible from within an enclave. Instead, we resemble this approach via TSX.

5.2 TSX-based Address Probing

We present a novel fault-resistant read primitive called *TAP* (TSX-based Address Probing).[1] In contrast to previous work, our attack is not a timing attack [27], *i.e.*, we solely exploit the TSX error codes. *TAP* uses TSX to determine whether a virtual address is accessible by the current process (*i.e.*, mapped and user accessible) or not. *TAP* exploits a side effect of TSX: When wrapping a memory access inside a TSX transaction, all potential access faults are caught by TSX instead of throwing an exception. Accessing an invalid memory location only aborts the transaction, but does not terminate the application. Thus, TSX allows to safely access any address within a transaction, without the risk of crashing the enclave. The resulting memory-disclosure primitive is extremely robust, as it automatically prevents reading of invalid memory locations. This has the advantage that an attacker does not require any knowledge of the memory layout, *i.e.*, which addresses are accessible. *TAP* probes an address as follows. We wrap a single read instruction to this address inside a TSX transaction.

***Accessible* Address.** If the accessed address is user-accessible, the transaction likely completes successfully. In rare cases it might fail due to external influences, such as interrupts (e.g., scheduling), cache eviction, or a concurrent modification of the accessed value. In these cases, TSX returns an error code indicating that the failure was only temporary and we can simply restart the transaction.

***Inaccessible* Address.** If the address is inaccessible, TSX suppresses the exception [23] (*i.e.*, the operating system is not notified) and aborts the transaction. The user code receives an error code and can handle the transaction abort. Although the error code does not indicate the precise abort reason, it is distinct from temporary failures that suggest a retry. Thus, we can deduce that the accessed address is either not mapped, or it is inaccessible from user space (e.g., kernel memory). Both reasons imply that the malicious enclave cannot read from the address. Thus, a further distinction is not necessary.

TAP is Stealthy. Although TSX can suppress exceptions from trapping to the operating system, TSX operation could be traced using hardware performance counters. However, when running in enclave mode, most hardware performance counters are not updated [25, 48]. We verified that especially none of the TSX-related performance counters are updated in enclave mode. Thus, running TSX-based Address Probing (*TAP*) in enclave mode is entirely invisible to the operating system. Note that this primitive can also be used in regular exploits for "egg hunting", *i.e.*, scanning the address space for injected shellcode [35, 43]. As it does not rely on any syscalls, it can neither be detected nor prevented by simply blocking the syscalls typically used for egg hunting.

[1] The implementation can be found at `https://github.com/IAIK/sgxrop`.

5.3 Address-Space Exploration

To mount a code-reuse attack, an attacker requires code gadgets to craft a chain of such gadgets. To collect enough gadgets, the enclave explores the host application's address space by means of *TAP*. Instead of applying *TAP* to every probed address, it suffices to probe a single address per page. This reveals whether the page is accessible to the enclave and allows the enclave to scan this entire page for gadgets via ordinary memory read instructions.

To detect gadgets, the attacker could scan the entire virtual address space, which takes approximately 45 minutes (Intel i7-6700K). To speed up the scanning, the attacker can apply JIT-ROP [52] to start scanning from a few known pointers. For example, the malicious enclave knows the host code address to which the ECALL is supposed to return. Also, the stack pointer to the host application stack is visible to the enclave. By scanning the host stack, the enclave can infer further valid code locations, e.g., due to saved return addresses. Thus, *TAP* can be used for the starting point of JIT-ROP, and to make JIT-ROP more resistant, as a wrongly inferred destination address does not crash the enclave.

Although JIT-ROP is fast, the disadvantage is that it is complex and only finds a fraction of usable executable pages [52]. With *TAP*, an attacker can choose the tradeoff between code coverage (*i.e.*, amount of discovered gadgets) and runtime of the gadget discovery. The most simple and complete approach approach is to linearly search through the entire virtual address space. To reduce the runtime of 45 minutes, an attacker can decide to use JIT-ROP for every mapped page instead of continuing to iterate through the address space.

After the address-space exploration, an attacker knows code pages which are usable to construct a ROP chain.

6 Escaping Enclaves with SGX-ROP

In this section, we present a novel way to mount a code-reuse attack from within SGX enclaves. We exploit the fact that SGX insufficiently isolates host applications from enclaves. In particular, we show that the shared virtual address space between host application and enclave, in combination with our address-space exploration (cf. Section 5), allows an attacker to mount a code-reuse attack on the application. Subsequently, the attacker gains arbitrary code execution within the host application, even if it is well-written and bug-free.

We discuss challenges in mounting the attack, and present solutions for all challenges. Moreover, we show how to construct a novel fault-resistant write primitive using TSX which allows an attacker to store additional shellcode.

6.1 Problem Statement

The attacker wants to gain arbitrary code execution, which is typically achieved by loading attack code to a data page and then executing it. However, this requires syscalls to make the injected code executable. To mount the attack, the

attacker first needs to escape the restricted enclave execution environment and bypass the host interface in order to execute syscalls. Until now it was unclear whether and how this could be achieved in practice. We show how to use SGX-ROP for that purpose. To inject an SGX-ROP chain (or arbitrary code) into the host application, the attacker requires knowledge about which memory locations are writable. Similar to before (Section 5), this demands a fault-resistant method to detect writable memory pages. Lastly, the attacker wants to remain stealthy and not perturb normal program execution. In particular, the malicious enclave shall always perform benign operations it is supposed to do and shall return to its host application via the intended interface. Also, after finishing the SGX-ROP attack, program execution shall continue normally.

6.2 Diverting the Control Flow

Towards SGX-ROP. In traditional code-reuse attacks, an attacker has to exploit a software bug (e.g., a buffer overflow) to get control over the instruction pointer. However, due to the shared address space of the host application and the enclave, an attacker can access arbitrary host memory. Thus, the attacker implicitly has a *write-anything-anywhere* primitive, allowing to directly overwrite instruction pointers, e.g., stored on the stack. Since the attacker knows the precise location of the host stack, he can easily locate a saved instruction pointer on the host stack and prepare a code-reuse attack by replacing it with a ROP chain. However, a code-reuse attack requires certain values to be on the current stack, e.g., multiple return addresses and possibly function arguments. Overwriting a larger part on the application stack might lead to data corruption and unexpected behavior of the host application. This would prevent recovering normal operation after the attack. Moreover, in contrast to traditional control-flow hijacking attacks, an SGX-ROP attacker does not only want to manipulate the control flow but also completely restore the original control flow after the attack to preserve functionality and remain stealthy.

Summing up, an SGX-ROP attacker cannot rely on any free or unused space on the current stack frame. Hence, the attacker requires a temporary stack frame to store the values required for the attack code.

Stealthy fake stack frames. We present a technique to store the SGX-ROP chain on a temporary *fake stack* which is an extension to stack pivoting [44]. The fake stack frame is located somewhere in unused writable memory, thus preserving stack data of the original program. First, the attacker copies the saved instruction pointer and saved base pointer to the fake stack frame. Then, the attacker replaces the saved instruction pointer with the address of a function epilogue gadget, *i.e.*, `leave; ret`, and the saved base pointer with the address of the fake stack frame. With the pivot gadget, the stack is switched to the new fake stack frame. However, in contrast to a normal stack pivot, preservering the old values allows the attacker to resume with the normal control flow when returning from the fake stack. Figure 3 illustrates the stealthy stack pivoting process. The injected stack frame contains a ROP chain which is used as attack code and continues normal execution after the ROP chain was executed.

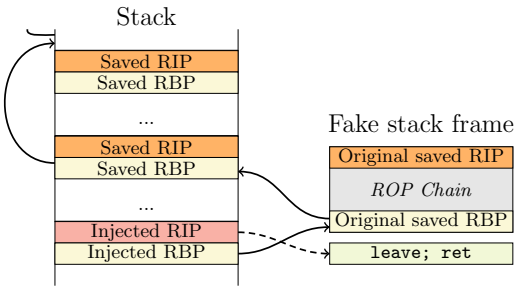

Fig. 3: To divert the control flow without interfering with legitimate stack frames, the attacker injects a new stack frame. The new stack frame can be used for arbitrary code-reuse attacks without leaving any traces in stack frames of other functions.

If the compiler saves the base pointer on a function call, a fake stack frame can be placed between any two stack frames, not only after the current function. Thus, attack code can be executed delayed, *i.e.*, not directly after the enclave returns to the host application, but at any later point where a function returns.

SGX-ROP evades a variety of ROP defense mechanisms. For example, stack canaries do not protect against SGX-ROP, since our fake stack frame bypasses the stack smashing detection. For software-based shadow stacks without write protection [13, 53], the attacker can perform SGX-ROP on the shadow stack as well. The write-anything-anywhere primitive can also be leveraged to break CFI policies [11], hardware-assisted CFI extensions [56], and stack-pivot defenses [61].

Gaining arbitrary code execution. With SGX-ROP, an attacker can stitch ROP gadgets together to execute syscalls in the host application. To gain arbitrary code execution, the enclave can inject attacker payload on a writable page and then use the ROP chain to instruct the operating system to bypass execution prevention (*i.e.*, the non-executable bit). On Linux, this can be done with a single mprotect syscall.

6.3 Detecting Writable Memory Locations

For SGX-ROP, the attacker requires unused, writable host memory to inject a fake stack frame as well as arbitrary attack payload. The enclave cannot allocate host application memory for that purpose but instead attempts to misuse existing host memory. However, as before, the attacker initially does not know the memory layout of its host application. In this section, we present *CLAW* (Checking Located Addresses for Writability), a combination of two TSX side effects to detect whether an arbitrary memory location is writable. This can be used to build a fault-resistant write primitive.

CLAW first leverages *TAP* to detect whether a virtual address is present, as shown in Figure 4.[1] Then, *CLAW* utilizes TSX to test whether this page is also writable. To do so, we encapsulate a write instruction to the page of interest within a TSX transaction and explicitly abort the transaction after the write.

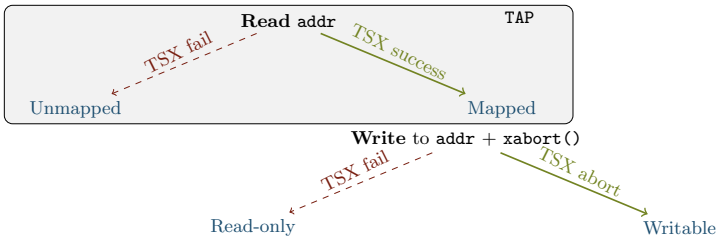

Fig. 4: *CLAW* exploits that memory writes in TSX are only visible to the outside of a transaction if it succeeds, and that TSX distinguishes between implicit and explicit aborts. Thus, the return value of TSX after writing to an address and explicitly aborting determines whether the memory location is writable without changing it.

Based on the return value of the transaction, we can deduce whether the page is writable. If the return value indicates an explicit abort, the write would have succeeded but was aborted by our explicit abort. In this case, we can deduce that the page is writable. If the page is read-only, the transaction fails with an error distinct from an explicit abort. The return value indicates that the transaction would never succeed, as the page is not writable. By observing those two error codes, one can distinguish read-only from writable pages, as shown in Figure 4.

A property of *CLAW* is that it is stealthy. Since all memory writes within a transaction are only committed to memory if the transaction succeeds, our explicit abort ensures that memory remains unmodified. Also, as with *TAP*, *CLAW* neither causes any exceptions to the operating system nor can it be seen in hardware performance counters.

Fault-resistant write-anything-anywhere primitive. With *CLAW*, building a fault-resistant write primitive is straightforward. Before writing to a page, *CLAW* is leveraged to test whether the page is writable. Then, the content can be safely written to the page.

Host Infection. Both the fake stack frame as well as placing arbitrary attack payload (e.g., a shellcode) require an unused writable memory location in the host application, which we denote as *data cave*. After finishing address space exploration (Section 5.3), the malicious enclave uses *CLAW* to test whether the found pages are writable. Again, probing a single address with *CLAW* suffices to test whether the entire page is writable. Moreover, the enclave needs to know whether it can safely use writable pages as data caves without destroying application data. We consider empty pages (*i.e.*, they only contain '0's) as safe. Note that the ROP chain and possible shellcode should always be zeroed-out after execution to obscure the traces of the attack.

7 Attack Evaluation

In this section, we evaluate *TAP* and *CLAW*, and show that *TAP* can also be used in traditional exploits for egg hunting. We scan Graphene-SGX [57] (an

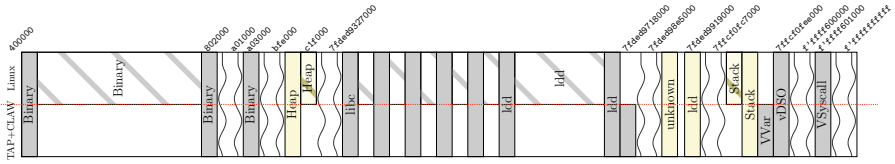

Fig. 5: The virtual memory layout of a simple program on Linux (x86_64) as provided by `/proc/<pid>/maps` (top) and reconstructed using *TAP+CLAW* (bottom).

SGX wrapper library) for data caves and ROP gadgets and also scan the SGX SDK for ROP gadgets. Finally, we present a simple proof-of-concept exploit. All evaluations were performed on an Intel i7-6700K with 16 GB of memory.

7.1 *TAP+CLAW*

We used the combination *TAP+CLAW* to scan the virtual memory of a process and also distinguish writable from read-only pages. Figure 5 shows the memory map of a process recovered with *TAP+CLAW* (bottom), compared with the ground truth directly obtained from the procfs file system (top). The procfs file system shows more areas (shaded), as it also includes pages which are not mapped in memory, but only defined in the binary. All mapped pages were correctly identified using *TAP*, and also the distinction between read-only and writable pages using *CLAW* was always correct.

Both *TAP* and *CLAW* are very fast, taking only the time of a cache read or write (around 330 cycles for an uncached memory access on our test machine) plus the overhead for a TSX transaction, which we measured as 30 cycles. Scanning the entire virtual address space takes 45 min, resulting in a probing rate of 48.5 GB/s. To estimate the runtime of *TAP* and *CLAW* on real-world applications, we evaluted both primitives on the 97 GNU Core Utilities with ASLR enabled. We linearly explored the binary starting from one known address (similarly to JIT-ROP [52]). On average, *TAP* located all pages of the application within 73.5 ms. This time can be reduced further, as an attacker can stop probing as soon as all required gadgets are found.

Egg Hunting. We also evaluated *TAP* as a novel method for egg hunting in regular (non-enclave) attacks, *i.e.*, scanning the address space for injected shellcode [35, 43]. State-of-the-art egg hunters for Linux [35, 38] rely on syscalls (e.g., `access`) which report whether one of the parameters is a valid address. However, issuing a syscall requires the `syscall` instruction as well as setting up the required parameters. Thus, such egg hunters are usually larger than 30 bytes [35]. Nemeth et al. [38] argued that egg hunters with fault prevention cannot be smaller. However, our *TAP* egg hunter is only 16 bytes in size,[1] *i.e.*, the smallest egg hunter with fault prevention. With 360 cycles per address, it is also significantly faster (by a factor of 4.8) than egg hunters leveraging the `access` syscall (1730 cycles per address).

7.2 Code-reuse Gadgets and Data Caves in SGX Frameworks

To evaluate the viability of a code-reuse attack using a fake stack frame (cf. Section 6.2), we inspected Graphene-SGX for data caves (cf. Section 6.3) and ROP gadgets. We chose Graphene-SGX, as it is open source[2], actively maintained, and designed to run unmodified binaries within SGX [57]. Furthermore, we also analyzed the Intel SGX SDK for ROP gadgets, as it is the most common framework for SGX applications.

Our simple attack enclave used *TAP+CLAW* to find code pages and data caves. We successfully detected all mapped pages of the host application, and also distinguished between writable and read-only pages.

Data Caves. With *CLAW*, we were able to detect which pages are not only present but also writable. For the writable pages, we further analyzed whether they contain only '0's and are thus data caves. We found 16 594 data caves in Graphene-SGX, which took on average 45.5 ms. This amounts to around 64.8 MB of memory which can be used by an attacker. Such data caves also exist in the Intel SGX SDK. Thus, even highly complex malware such as zero-day exploits can be stored there. For traditional shellcode, a one-page data cave is already sufficient, as such a shellcode fits within a few bytes.

Gadgets. Data caves allow storing arbitrary code to the memory. An attacker requires a ROP chain which makes the data caves executable, e.g., the `mprotect` syscall on Linux. This syscall can be called using a ROP chain consisting of only 4 gadgets: `POP RDI`, `POP RSI`, `POP RAX`, and `SYSCALL`. We analyzed the code pages of Graphene-SGX identified using *TAP* (cf. Section 5.3). We found all gadgets required to call `mprotect` in multiple pages of Graphene-SGX, e.g., in the binary (pal-linux), math library (libm), GNU C library (libc) and GNU linker (ld).

Furthermore, 3 out of the 4 gadgets are not only in one of the core libraries of Graphene-SGX (libsysdb), but also in the Intel SGX SDK itself (libsgx_urts). The fourth gadget (`SYSCALL`) to build a complete chain can, e.g., be found in the virtual syscall page, which is mapped in every process on modern Linux systems, or in the libc.

7.3 Full Exploit

Our proof-of-concept exploit consists of a benign application hosting a malicious enclave. We use the most restricted enclave interface possible: the enclave may not use any OCALLs. After entering the enclave via any ECALL, the enclave uses *TAP* and *CLAW* to find and inject code and data into a data cave. Using *TAP*, the enclave detects host binary pages and builds a ROP chain which creates a new file (in a real attack, the enclave would encrypt existing files) and displays a ransom message. We divert the control flow (cf. Section 6.2) to let the host application execute the ROP chain, and immediately continue normal execution.

Our host application uses ASLR, stack canaries, and address sanitizer. The host application does not provide any addresses to the enclave which can be used as a starting point. Still, the end-to-end exploit[1] takes on average only 20.8 s.

[2] `https://github.com/oscarlab/graphene`

8 Discussion

SGX-ROP surpasses traditional ROP attacks, as the enclave isolation works only in one direction, *i.e.*, the enclave is protected from the host application, but not vice-versa. A write-anything-anywhere primitive is sufficient to break even extremely strict CFI policies [11] and hardware-assisted control-flow integrity extensions [56]. In contrast to regular ROP attacks, we do not require a memory safety violation. Also, the Intel SGX SDK yields enough ROP gadgets and data caves to gain arbitrary code execution. Hence, SGX-ROP is always possible on current applications if, inadvertently, a malicious enclave is embedded.

With SGX-ROP, porting malware to SGX becomes trivial, thus intensifying the threat of enclave malware. Moreover, hiding malware in an SGX enclave give attackers plausible deniability and stealthiness until they choose to launch the attack. This is particularly relevant for trigger-based malware that embeds a zero-day exploit, but also to provide plausible deniability for legal or political reasons, e.g., for a state actor [10, 20]. Possible scenarios range from synchronized large-scale denial-of-service attacks to targeted attacks on individuals.

9 Conclusion

We practically demonstrated the first enclave malware which fully and stealthily impersonates its host application. Our attack uses new TSX-based techniques: a memory-disclosure primitive and a write-anything-anywhere primitive. With SGX-ROP, we bypassed ASLR, stack canaries, and address sanitizer, to run ROP gadgets in the host context enabling practical enclave malware. We conclude that instead of protecting users from harm, SGX currently poses a security threat, facilitating so-called super-malware with ready-to-hit exploits. Our results lay ground for future research on more realistic trust relationships between enclave and non-enclave software, as well as the mitigation of enclave malware.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

Bibliography

[1] Adamski, A.: Overview of Intel SGX - Part 2, SGX Externals (Aug 2018)

[2] Andriesse, D., Bos, H.: Instruction-level steganography for covert trigger-based malware. In: DIMVA (2014)

[3] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., et al.: SCONE: Secure Linux Containers with Intel SGX. In: OSDI (2016)

[4] Aumasson, J.P., Merino, L.: SGX Secure Enclaves in Practice: Security and Crypto Review. In: Black Hat Briefings (2016)

[5] Bauman, E., Lin, Z.: A case for protecting computer games with SGX. In: Workshop on System Software for Trusted Execution (2016)

[6] Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., Boneh, D.: Hacking blind. In: S&P (2014)

[7] Borrello, P., Coppa, E., D'Elia, D.C., Demetrescu, C.: The rop needle: Hiding trigger-based injection vectors via code reuse. In: ACM Symposium on Applied Computing (SAC) (2019)

[8] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT (2017)

[9] Brenner, S., Hundt, T., Mazzeo, G., Kapitza, R.: Secure cloud micro services using Intel SGX. In: IFIP International Conference on Distributed Applications and Interoperable Systems (2017)

[10] Gesetz zur effektiveren und praxistauglicheren Ausgestaltung des Strafverfahrens (2017)

[11] Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: Control-flow bending: On the effectiveness of control-flow integrity. In: USENIX Security (2015)

[12] Caulfield, T., Ioannidis, C., Pym, D.: The US vulnerabilities equities process: An economic perspective. In: International Conference on Decision and Game Theory for Security (2017)

[13] Chiueh, T.c., Hsu, F.H.: Rad: A compile-time solution to buffer overflow attacks. In: Conference on Distributed Computing Systems (2001)

[14] Costan, V., Devadas, S.: Intel SGX explained (2016)

[15] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security (1998)

[16] Crandall, J.R., Wassermann, G., de Oliveira, D.A., Su, Z., Wu, S.F., Chong, F.T.: Temporal search: Detecting hidden malware timebombs with virtual machines. In: ACM SIGARCH Computer Architecture News. vol. 34 (2006)

[17] Davenport, S., Ford, R.: SGX: the good, the bad and the downright ugly (Jan 2014), `https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly`

[18] Dunn, A.M., Hofmann, O.S., Waters, B., Witchel, E.: Cloaking malware with the trusted platform module. In: USENIX Security Symposium (2011)
[19] Egelman, S., Herley, C., Van Oorschot, P.C.: Markets for zero-day exploits: Ethics and implications. In: New Security Paradigms Workshop (2013)
[20] Electronic Frontier Foundation: New FBI documents provide details on government's surveillance spyware (2011)
[21] Guan, L., Lin, J., Luo, B., Jing, J., Wang, J.: Protecting private keys against memory disclosure attacks using hardware transactional memory. In: S&P (2015)
[22] Hall, C.G.: Time sensitivity in cyberweapon reusability. Ph.D. thesis, Monterey, California: Naval Postgraduate School (2017)
[23] Intel: Intel® 64 and IA-32 Architectures Software Developer′s Manual, Volume 3 (3A, 3B & 3C): System Programming Guide (325384) (2016)
[24] Intel Corporation: Software Guard Extensions Programming Reference, Rev. 2. (2014)
[25] Intel Corporation: Intel SGX: Debug, Production, Pre-release what's the difference? (Jan 2016)
[26] Intel Corporation: Enclave Signing Key Management (May 2018)
[27] Jang, Y., Lee, S., Kim, T.: Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS (2016)
[28] Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In: CCS (2007)
[29] King, S., Chen, P.: SubVirt: implementing malware with virtual machines. In: S&P (2006)
[30] Kuvaiskii, D., Faqeh, R., Bhatotia, P., Felber, P., Fetzer, C.: Haft: Hardware-assisted fault tolerance. In: EuroSys (2016)
[31] Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., Kang, B.B.: Hacking in darkness: Return-oriented programming against secure enclaves. In: USENIX Security (2017)
[32] Leitch, J.: Process hollowing (2013)
[33] Liu, Y., Xia, Y., Guan, H., Zang, B., Chen, H.: Concurrent and consistent virtual machine introspection with hardware transactional memory. In: High Performance Computer Architecture (HPCA) (2014)
[34] Marschalek, M.: The Wolf In SGX Clothing. Bluehat IL (Jan 2018)
[35] Miller, M.: Safely searching process virtual address space (2004)
[36] Moxie Marlinspike: Technology preview: Private contact discovery for signal (2017)
[37] Myers, M., Youndt, S.: An introduction to hardware-assisted virtual machine (HVM) rootkits. Mega Security (2007)
[38] Németh, Z.L., Erdődi, L.: When every byte counts - writing minimal length shellcodes. In: Intelligent Systems and Informatics (SISY) (2015)
[39] Nicolas Bacca: Soft launching ledger SGX enclave (2017)
[40] Noubir, G., Sanatinia, A.: Trusted code execution on untrusted platforms using Intel SGX. Virus Bulletin (2016)
[41] PaX Team: Address space layout randomization (ASLR) (2003)
[42] PaX Team: Rap: Rip rop (2015)

[43] Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Comprehensive shellcode detection using runtime heuristics. In: ACSAC (2010)
[44] Prakash, A., Yin, H.: Defeating rop through denial of stack pivot. In: ACSAC (2015)
[45] Russinovich, M.: Sony, Rootkits and Digital Rights Management Gone Too Far (Oct 2005)
[46] Rutkowska, J.: Thoughts on Intel's upcoming Software Guard Extensions (Part 2) (2013)
[47] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: VC3: trustworthy data analytics in the cloud using SGX. In: S&P (2015)
[48] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA (2017)
[49] Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS (2007)
[50] Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: NDSS (2008)
[51] Shinde, S., Le Tien, D., Tople, S., Saxena, P.: PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In: NDSS (2017)
[52] Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: S&P (2013)
[53] Stack Shield: A stack smashing technique protection tool for linux (2011)
[54] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: EuroSys (2009)
[55] Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal War in Memory. In: S&P (2013)
[56] Theodorides, M., Wagner, D.: Breaking active-set backward-edge cfi. In: Hardware Oriented Security and Trust (HOST) (2017)
[57] Tsai, C.C., Porter, D.E., Vij, M.: Graphene-SGX: A practical library OS for unmodified applications on SGX. In: USENIX ATC (2017)
[58] Vrancken, K., Piessens, F., Strackx, R.: Hardening Intel SGX applications: Balancing concerns. In: Workshop on System Software for Trusted Execution (2017)
[59] Weichbrodt, N., Kurmus, A., Pietzuch, P., Kapitza, R.: Asyncshock: Exploiting synchronisation bugs in Intel SGX enclaves. In: ESORICS (2016)
[60] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F., Yarom, Y.: Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution (2018)
[61] Yan, F., Huang, F., Zhao, L., Peng, H., Wang, Q.: Baseline is fragile: on the effectiveness of stack pivot defense. In: ICPADS (2016)
[62] You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: Broadband, Wireless Computing, Communication and Applications (2010)