The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX

Andrea Biondo and Mauro Conti, *University of Padua;* Lucas Davi, *University of Duisburg-Essen;* Tommaso Frassetto and Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*

https://www.usenix.org/conference/usenixsecurity18/presentation/biondo

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX

Andrea Biondo, Mauro Conti
University of Padua, Italy

Lucas Davi
University of Duisburg-Essen, Germany

Tommaso Frassetto, Ahmad-Reza Sadeghi
TU Darmstadt, Germany

Abstract

Intel Software Guard Extensions (SGX) isolate security-critical code inside a protected memory area called enclave. Previous research on SGX has demonstrated that memory corruption vulnerabilities within enclave code can be exploited to extract secret keys and bypass remote attestation. However, these attacks require kernel privileges, and rely on frequently probing enclave code which results in many enclave crashes. Further, they assume a constant, not randomized memory layout.

In this paper, we present novel exploitation techniques against SGX that do not require any enclave crashes and work in the presence of existing SGX randomization approaches such as SGX-Shield. A key contribution of our attacks is that they work under weak adversarial assumptions, e.g., not requiring kernel privileges. In fact, they can be applied to any enclave that is developed with the standard Intel SGX SDK on either Linux or Windows.

1 Introduction

Intel recently introduced Software Guard Extensions (SGX), which aim at strongly isolating sensitive code and data from the operating system, hypervisor, BIOS, and other applications. In addition, SGX also features sophisticated memory protection techniques that prevent memory snooping attacks: SGX code and data is always encrypted and integrity-protected as soon as it leaves the CPU chip, e.g., when it is stored in main memory. SGX is especially useful in cloud scenarios as it ensures isolated execution of code and data within an untrusted computing environment.

SGX was designed to allow developers to protect small parts of their application that handle sensitive data, e.g., cryptographic keys, inside SGX containers called *enclaves*. An enclave is a strongly isolated execution environment that can be dynamically created while the main application, known as *host*, is running. The host can invoke specific functions in an SGX enclave by using one of the pre-defined entry points. The enclave can subsequently perform sensitive computations, call pre-defined functions in the host, and return to the caller.

In the ideal scenario, the enclave code only includes minimal carefully-inspected code, which could be formally proven to be free of vulnerabilities. However, legacy applications can be adapted as well to run inside SGX enclaves with relatively minor modifications. Formally verifying or manually inspecting such complex legacy software is not feasible, meaning that the same *memory-corruption vulnerabilities* that plague legacy software are also very likely to occur in those complex enclaves.

However, previous research on SGX has been mainly focused on side-channel attacks [31, 29, 6] and defenses [28, 12, 5]. Only recently, Lee et al. [19] presented the first memory-corruption attack against SGX. Their attack, called Dark-ROP, is based on several oracles and return-oriented programming (ROP) [27]. The oracles inform the attacker about the internal status of the enclave execution, whereas ROP maliciously re-uses benign code snippets (called *gadgets*) to undermine non-executable memory protection. In particular, Dark-ROP requires kernel privileges and is based on principles of blind ROP [3]: if an application is not randomized, or it is not re-randomized after crashing, crashes can and do leak useful information to the attacker. This allows Dark-ROP to extract secret code and data, as well as undermine remote attestation. However, Dark-ROP requires a constant, non-randomized memory layout as the oracles frequently crash enclaves. Hence, to address the Dark-ROP attack, Seo et al. demonstrated an implementation of SGX randomization called SGX-Shield [26], since this attack is not effective if the SGX code is randomized. Dark-ROP relies on running the target enclave multiple times to test multiple addresses, so randomizing the memory layout at initialization time makes previous results useless for new invocations.

However, SGX-Shield does not randomize the part of the SGX SDK [14, 15] that handles transitions between host code and enclave code. Thus, the location of this code, which contains a number of very interesting gadgets to mount ROP attacks, is known to the attacker. This paper demonstrates that this interface code is enough to mount powerful run-time attacks and bypass SGX-Shield without requiring kernel privileges. Extending the randomization to this interface code would be very technically involved due to its low-level nature and the architectural need to have a fixed entry point, as we discuss in Section 8. Moreover, even a finely-randomized interface code would be vulnerable to side-channel attacks. Finally, architectural limitations in SGX[1] force randomized code to be executed from writable pages, thus allowing simpler code-injection.

Goals and Contributions. We show that even fine-grained code randomization for SGX can be bypassed by exploiting parts of the SDK code, and point out the need for more advanced approaches to mitigate run-time attacks on SGX enclaves. In summary, our main contributions are:

- We propose two new code-reuse attacks against enclaves built on top of the Intel SGX SDK. By abusing preexisting SDK mechanisms, these attacks provide full control of the CPU's general-purpose registers to an attacker able to exploit a memory corruption vulnerability (Section 6). We also reverse-engineered and describe the internals of the ECALL, OCALL and exception handling mechanisms of the Intel SGX SDK (Section 4).
- To demonstrate that our new attacks are powerful, we show that they are effective and practical against SGX-Shield [26], a state-of-the-art fine-grained randomization solution for SGX enclaves (Section 7). Moreover, we highlight several discrepancies between the SGX-Shield paper and the proposed open source implementation.
- We discuss possible countermeasures and mitigations to prevent our attacks from two perspectives: hardening the enclave itself, and hardening the SDK (Section 8).

2 Related Work

Side-channel attacks. Multiple works have shown that SGX is vulnerable to micro-architectural side-channel attacks since untrusted code and enclave code share the same processor. Side-channel attacks can leak critical secrets from the enclave, such as cryptographic keys. Controlled-channel attacks [31] employ a malicious kernel to infer memory access patterns at the granularity of pages by triggering page faults in the enclave. They show how the strong adversary model of SGX can introduce new kinds of attacks. Cache-based side channels have been widely studied and exploit the caching mechanisms of the processor, as unrelated processes can share cache resources [13, 17, 21, 32]. Software Grand Exposure [6] and CacheZoom [23] further show how cache side channels are especially powerful within the strong adversary model of SGX. Another micro-architectural component that has been exploited is the branch predictor. Lee et al. [20] abuse collisions within the branch predictor to infer whether a branch inside the enclave has been taken. They demonstrate their attack by monitoring an RSA exponentiation routine to recover the key. All these side-channel attacks require frequent interruption of the enclave. Therefore, defenses such as T-SGX [28] and Déjá Vu [7] are based on avoiding or detecting enclave interruptions forced by a malicious kernel. In response, Van Bulck et al. [29] proposed an attack that can monitor memory accesses at page granularity without interrupting the enclave. A different mitigation strategy is making the location of data unpredictable to stop the attacker from extracting information from memory access patterns. On this note, DR. SGX [5] performs fine-grained randomization of data by permuting it at cache line granularity.

Memory corruption. Enclaves, just like normal applications, can suffer from memory corruptions vulnerabilities. SGXBounds [18] offers protection against out-of-bounds memory accesses. Dark-ROP [19] is a code-reuse attack that makes return-oriented programming (ROP) [27] possible against encrypted SGX enclaves. Haven [1, 2] and VC3 [24] deploy a symmetrically encrypted enclave along with a loader which will receive the key through remote attestation. Such enclaves cannot be analyzed or reverse engineered, as the key is only available within an enclave whose integrity has been verified via attestation. Therefore, typical ROP attacks do not work. Dark-ROP proposes a way to dynamically find ROP gadgets by building a series of oracles [19]. Those rely on being able to crash and reconstruct the enclave multiple times while preserving the memory layout, and possessing kernel privileges. Randomization schemes such as SGX-Shield [26] challenge this assumption, since the memory layout changes every time the enclave is constructed. Further, SGX-Shield makes traditional exploitation techniques extremely hard to apply because it employs fine-grained randomization and non-readable code. However, in this paper, we present exploits that undermine these mitigation techniques under weak adversarial assumptions.

[1] In the current version of SGX, memory permissions cannot be changed after initialization. This limitation will be lifted in SGX2 [22]; however, no available processor currently supports this new version.

3 SGX Background

In this section, we recall selected background information on SGX. For a more thorough analysis, we refer to [8] and Intel's official reference manual on SGX [16].

3.1 Enclave Entry and Exit

SGX enclaves run on the same x86 processor as ordinary application code does. As such, mechanisms are required to switch between untrusted and trusted execution modes, as shown in Figure 1. The SGX instructions to interact with enclaves are organized as *leaf functions* under two real instructions: ENCLS and ENCLU. The former is used for kernel-mode operations, while the second for user-mode operations. SGX accomplishes synchronous enclave entry by means of the EENTER leaf function, which is invoked via the ENCLU instruction. The entry point is specified in the *Thread Control Structure* (TCS) for the relevant thread. Since EENTER does not clear the CPU registers, the untrusted code can pass additional information to the entry point. For instance, an enclave may expose various operations to its client. The untrusted code could pass a parameter that indicates what operation it wants the enclave to perform. To return back to untrusted code, the enclave uses the EEXIT leaf. Just like EENTER, EEXIT does not clear CPU registers, thereby allowing trusted code to pass data to untrusted code. An enclave can be entered multiple times concurrently within the same thread. The number of concurrent entries in the same thread is limited by the number of *State Save Areas* (SSAs) defined by the enclave. The SSA is used to store enclave state during asynchronous exits, which are described below. The *number of SSAs* (NSSA) field in the TCS defines how many SSAs are present.

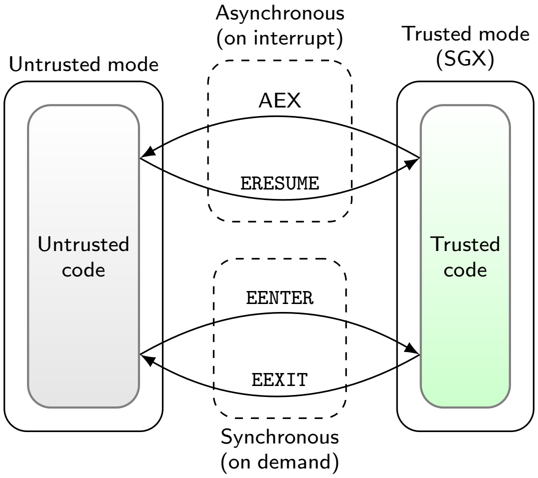

Figure 1: Enclave entry and exit mechanisms.

An enclave can also exit because of a hardware exception (such as an interrupt), which needs to be handled by the kernel in untrusted mode. This event is known as *Asynchronous Enclave Exit* (AEX). When an AEX occurs, the current enclave state is saved in an available SSA and the register values are replaced with a synthetic state before handing control to the interrupt handler. The synthetic state ensures the enclave's opacity and avoids leakage of secrets. Once the interrupt is dealt with, enclave execution can be resumed with the ERESUME leaf, which restores the previous state from the SSA.

4 SGX SDK Internals

In this section, we review selected internal mechanisms of the official SGX SDK[14, 15] that are relevant to our attack. In general, SGX software is developed based on the official SGX SDK, as it abstracts away the underlying complexity of SGX. Two SDK-provided libraries are vital for our attack and the correct execution of SGX code: the *Trusted Runtime System* (tRTS) and the *Untrusted Runtime System* (uRTS). While tRTS is executing inside an enclave, uRTS runs outside the enclave. The tRTS and uRTS interact with each other to handle the transitions between trusted and untrusted execution modes.

4.1 ECALLs

The ECALL mechanism allows untrusted code to call functions inside an enclave. The enclave programmer can arbitrarily select which functions are to be exposed for the ECALL interface. ECALLs can also be nested: untrusted code can execute an ECALL while handling an OCALL (see Section 4.2). The programmer can choose which ECALLs are allowed at the zero nesting level, and which are allowed for each specific OCALL. Every defined ECALL has an associated index. To perform an ECALL, the application calls into the uRTS library, which executes a synchronous enclave entry (EENTER), passing the ECALL index in a register. We recall that EENTRY does not clear the registers. The tRTS then checks whether an ECALL with that index is defined, and if it is allowed at the current nesting level. If the checks pass, it executes the target function. Once the function returns, it performs a synchronous exit (EEXIT) to give control back to the uRTS. Passing and returning arbitrarily complex data structures is possible because SGX enclaves can access untrusted memory. An enclave must expose at least an ECALL, otherwise there is no way to invoke enclave code: from the programmer's perspective, an enclave's code always executes in ECALL context.

4.2 OCALLs

The OCALL mechanism, shown in Figure 2, allows trusted code to call untrusted functions defined by the host application. The need for OCALLs mainly stems from the fact that system calls are not allowed inside an enclave. Like ECALLs, each OCALL is identified by an index. When the enclave code has to perform an OCALL, it calls into the tRTS (step 1 of Figure 2). The tRTS first pushes an *OCALL frame* onto the trusted thread stack, which stores the current register state (step 2). Next, it performs a synchronous enclave exit to return from the current ECALL, passing the OCALL index back to the uRTS (step 3). The uRTS recognizes that the exit is for an OCALL and executes the target function (step 4). Thereafter, it executes a special variant of ECALL known as ORET (step 5), which will restore the context from the OCALL frame through a function named `asm_oret`, thus returning to the trusted callsite (step 6). ORET is implemented in the tRTS. Like ECALLs, data is passed via shared untrusted memory.

4.3 Exception Handling

SDK enclaves can register handlers to catch exceptions within the enclave. This exception handling mechanism is shown in Figure 3. Upon an exception (e.g., invalid memory access, division by zero) an asynchronous enclave exit (AEX) occurs, which saves the faulting state to the state save area (SSA). The resulting interrupt is handled by the kernel, which delivers an exception to the untrusted application by means of the usual exception mechanism of the OS (e.g., signals in Linux-based systems, structured exception handling in Windows). An exception handler registered by the uRTS performs a special ECALL to let the enclave handle the exception. By default, SDK enclaves have two SSAs available (specified in the `NSSA` field in the TCS). Hence, it is possible to re-enter the enclave while an AEX is pending. The tRTS then copies the faulting state from the SSA to an exception information structure on the trusted stack, and changes the SSA contents so that `ERESUME` will continue at a second-phase handler in the tRTS, instead of executing the faulting instruction again. Once the ECALL returns, the uRTS issues an `ERESUME` for the faulting thread, which will resume at the second-phase handler. This traverses the registered exception handlers, which can then observe the exception information to determine whether they can handle the exception. To handle the exception, a handler can modify the CPU state contained in the exception information. If a handler succeeds, the tRTS uses a function named `continue_execution` to restore the CPU register context from the exception information, thus resuming enclave execution. If the exception cannot be handled, a default handler switches the enclave to a crashed state, which prevents further operations on it.

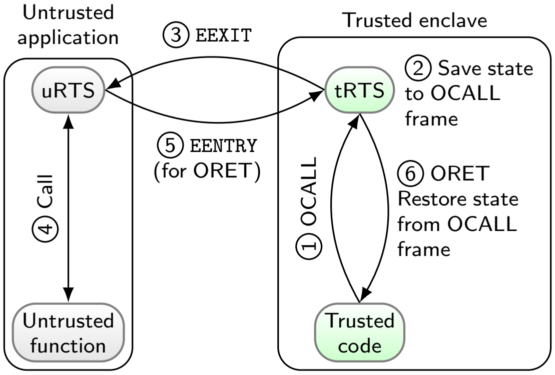

Figure 2: OCALL mechanism from the SGX SDK.

5 Threat Model and Assumptions

Previous work on SGX [19, 26] has considered a very strong adversarial model: the attacker has full control over the machine, e.g., through a malicious kernel. In this work, we consider a weaker attacker that has compromised the application that hosts the enclave, e.g., by exploiting a vulnerability. In some cases, as discussed below, an attacker might even be able to perform the attack without any control over the host process.

Offensive capabilities. Our attacker has the following capabilities:

- **Memory corruption vulnerability.** The attacker has knowledge of a vulnerability in the enclave that allows her to either corrupt stack memory (e.g., a stack overflow) or corrupt a function pointer on the stack, heap, or other memory areas (e.g., heap overflow, use-after-free or type confusion).

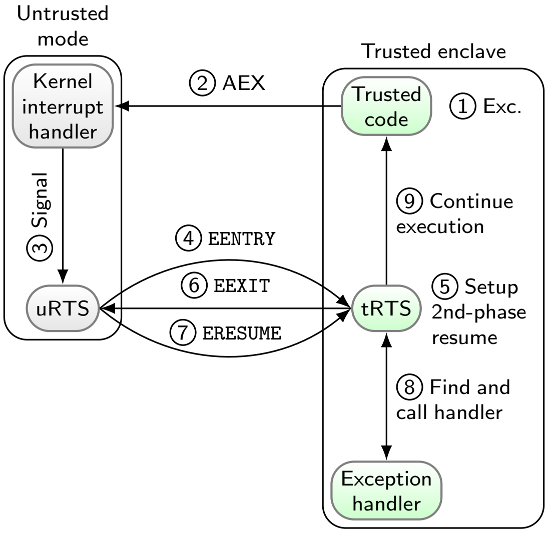

Figure 3: Exception handling mechanism from the SGX SDK.

- **Ability to create fake structures.** The attacker can place arbitrary data at some memory location accessible by the enclave. A malicious host process can easily do this given the unrestricted access over its own address space. An attacker could also possibly achieve this via normal functionality, for example by steering the application or the enclave into allocating attacker-controlled data at predictable addresses.
- **Knowledge of coarse-grained memory layout.** The attacker knows the victim enclave's external memory layout, i.e., its virtual address range. This is known to the process hosting the enclave, as the enclave virtual memory resides in its address space. Alternatively, information leakage vulnerabilities inside the enclave could provide this knowledge to an attacker who is not in control of the process.
- **Knowledge of the enclave's binary.** The attacker has access to the victim enclave's binary allowing her to run static analysis on the binary.

Defensive capabilities. We make the following assumptions about the victim enclave:

- **SDK usage.** The victim enclave is developed by means of the official SGX SDK from Intel. The SDK is used by almost all real-world enclaves, as it is the development environment endorsed by Intel. Furthermore, it has been used in various academic works [26, 30].
- **Randomized SGX memory.** We also assume that enclave code is additionally hardened by sophisticated mitigation technologies such as address space layout randomization (ASLR). That is, we assume that the victim enclave is protected by means of SGX-Shield [26], which is currently the only available ASLR solution for SGX. Recall that existing memory corruption attacks against SGX, e.g., Dark-ROP [19], are mitigated by SGX-Shield.

6 The Guard's Dilemma

We now present in detail our novel code-reuse attacks against SGX. The techniques we propose are applicable to a wide range of vulnerabilities, including stack overflows and corruption of function pointers. In particular, the latter is common in modern object-oriented code. Our ultimate attack goal is to execute a sequence (*chain*) of *gadgets*, i.e., existing functions or short instruction sequences, to perform a malicious activity of the attacker's choosing, without crashing the victim enclave. This is along the lines of any other common code-reuse attack such as return-oriented programming. However, the advantage of our attack is to allow the attacker to set all general-purpose CPU registers before executing each gadget. Controlling registers is essential in any code-reuse attack. For instance, they can prepare data for subsequent gadgets or set arguments for function calls. In contrast, existing code-reuse attacks on x86 require the attacker to use specific register-setting gadgets (e.g., *pop* gadgets) to set registers.

Not requiring those gadgets has two major benefits. First, it reduces the amount of application code needed for a successful code-reuse attack, which is helpful in constrained environments, as we demonstrate in Section 7 with an exploit against SGX-Shield [26]. Second, it simplifies payload development since the attacker does not need to find *pop* gadgets for all relevant registers. In fact, our attacks allow the attacker to use whole functions as building blocks instead of small gadgets, allowing her to work on a higher level and making it easier to port the exploit between different versions of a binary.

Our attacks abuse functionality in tRTS, a fundamental library of the Intel SGX SDK, which most enclaves use (Section 5). Hence, our attacks threaten a large amount of existing enclave code. Here lies the dilemma: the SDK is an important part in creating secure enclaves, but in this case it is actually exposing them to attacks.

We devise two new exploitation primitives to launch memory corruption attacks against SGX:

- **The ORET primitive.** Our first attack technique allows the attacker to gain access to a critical set of CPU registers by exploiting a stack overflow vulnerability (cf. Section 5).
- **The CONT primitive.** Our second attack technique is even more powerful as it allows the attacker to gain access to all general-purpose registers. It only requires control of a register (on x86_64, `rdi`). In addition, this attack can be combined with the ORET primitive to also apply it to controlled stack situations.

6.1 Overview and Attack Workflow

In this section, we present a high-level description of the exploitation primitives and the attack workflow.

6.1.1 Exploitation Primitives

In the following, we explain our exploitation primitives and their preconditions.

ORET primitive. This primitive is based on abusing the function `asm_oret` from the tRTS library in the Intel SGX SDK. Normally, this function is used to restore the CPU context after an OCALL. The prerequisites for this primitive are control of the instruction pointer (to hijack execution to `asm_oret`) and control of stack contents. For instance, any common stack overflow vulnerability such as a buffer overflow or format string is sufficient to use the ORET primitive. The ORET primitive gives control of a subset of CPU registers, including the register that holds the first function argument (`rdi`) and the instruction pointer.

CONT primitive. This primitive abuses the function `continue_execution` from the tRTS, which is meant to restore the CPU context after an exception. This primitive requires the ability to call that function with a controlled `rdi`, which is achievable by exploiting a memory corruption vulnerability affecting a function pointer (not necessarily located on the stack). This primitive yields full control over all general-purpose CPU registers.

ORET+CONT loop. The basic idea behind our attack is to use the CONT primitive repeatedly to invoke the various gadgets with the correct register values. Thus, the chain needs to have multiple CONT invocations. Recall that CONT requires a specific value for `rdi`, which the other gadgets might modify. An easy way to satisfy this constraint is to use ORET invocations to set `rdi` and invoke CONT, building an *ORET+CONT loop*. Each iteration of this loop executes one gadget and is structured as follows:

1. A CONT primitive manipulates the stack pointer to hijack it into attacker-controlled memory and executes a gadget.
2. Once the gadget completes, the previous stack manipulation causes the execution of an ORET primitive.
3. The ORET primitive triggers the CONT primitive for the next gadget, continuing the cycle from the first step.

6.1.2 Workflow

This section describes the workflow of our attack based on Figure 4.

Step 1: Payload preparation. In preparation for the exploit, the attacker performs static analysis on the enclave binary to determine the gadgets she wants to reuse. Our attack supports classic ROP gadgets, i.e., code sequences ending with a return instruction, and any subroutine for function-reuse attacks. Note that, even if the main enclave code is randomized, it is very difficult to randomize *all* the enclave code (Section 8) and the non-randomized code contains enough gadgets to successfully mount an attack (Section 7). Next, the attacker constructs a gadget chain consisting of a sequence of gadgets which will perform the desired malicious activity, and defines the register state that should be set before executing each gadget. For instance, if the gadget is an entire function, registers will hold the function arguments. According to the threat model defined in Section 5, the attacker knows the external memory layout of the enclave, including its base address. Therefore, the attacker just needs to know the static offset of a gadget in the enclave binary to find its run-time address. In addition to the payload gadgets, the attacker has to determine the offsets of `asm_oret` and `continue_execution` (both in the tRTS) to apply our attack.

Step 2: Fake structures preparation. Our primitives work by abusing functions intended to restore CPU contexts by tricking them into restoring fake contexts, thus gaining control of the registers. In contrast to a standard ROP exploit, which is usually self-contained, our attacks require a number of auxiliary memory structures to hold these fake contexts and execute our primitives. Since enclaves can access user memory outside the enclave, the structures do not have to be within the trusted enclave memory. They can be in any memory shared with the enclave (e.g., in the host's memory) as long as its position is known. Specifically, our attack requires two kinds of fake structures:

- Multiple *fake exception information structures*, which contain register contexts for the CONT primitives. One fake exception information structure is required for each gadget, in order to set the registers to the correct values and execute the gadget.
- A *fake stack*, which is a supporting structure for the ORET+CONT loop that serves two purposes. On the one hand, it is used to bring control back to an ORET primitive after a gadget executes. On the other hand, it contains fake contexts for the transition from the ORET primitive to the CONT primitive to continue the loop.

Step 3: Attack execution. Thanks to the way the fake structures are set up, triggering the first CONT primitive will start the ORET+CONT loop. Every cycle will execute a gadget and advance the chain, thus running the attacker's payload. The only remaining aspect to analyze is how the first CONT is triggered. The easiest case is when the vulnerability already satisfies the CONT preconditions (e.g., exploitation of an indirect function call). In that case, the attacker can execute the first CONT directly. Exploiting a stack overflow is also possible with little additional effort. This kind of vulnerability allows to run an ORET primitive. Since it can be used to set the first function argument register and the instruction pointer, the attacker now has the controlled function call needed for CONT and can trigger the loop.

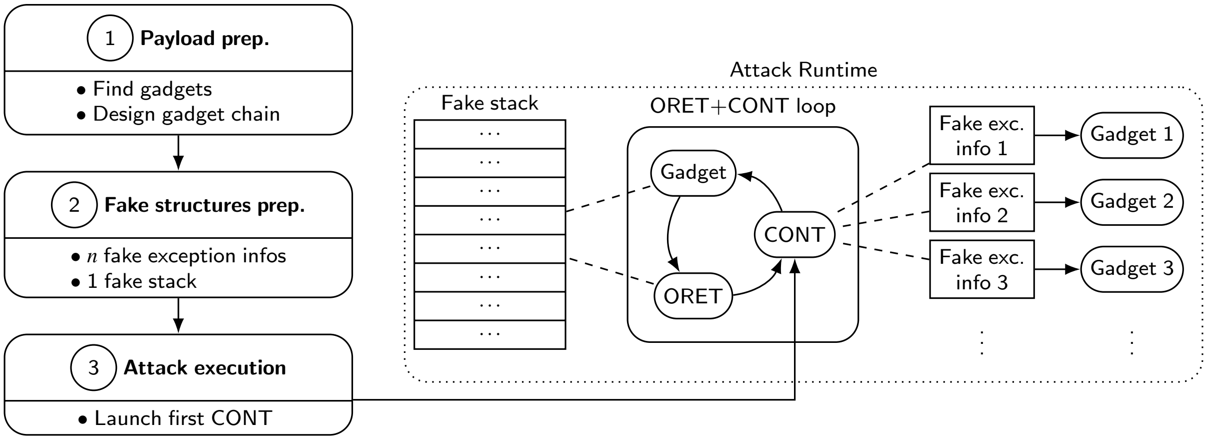

Figure 4: Overview of the workflow of our attack.

6.2 Details

In this section, we describe the technical details and interaction of our exploitation primitives to craft a memory corruption attack against SGX.

6.2.1 ORET Primitive

Our ORET primitive abuses the asm_oret function, used in the OCALL/ORET mechanism to restore the CPU context from the OCALL frame saved on the stack. This function allows controlling parts of the CPU context, and can be a stepping stone to the CONT primitive.

The prototype of the function is sgx_status_t asm_oret(uintptr_t sp, void *ms). The first argument (sp) points to the OCALL frame, which contains the partial CPU context to be restored, including saved values for rbp, rdi, rsi, rbx and r12 to r15. Listing 1 shows the layout of this structure. The second argument (ms) is not relevant for our attack. An attacker able to control the OCALL frame can set all the registers mentioned; moreover, the new instruction pointer (rip) can also be set. Since the attacker can control rdi (which contains the first argument) and the instruction pointer, she can execute the CONT primitive from ORET. This capability is important for the ORET+CONT loop, and additionally allows to bootstrap our attack from a stack overflow vulnerability, as will be shown towards the end of this section.

The exact values of rsp and rip after asm_oret depend on the SGX SDK version. For versions earlier than 2.0, the stack pointer is set to point to the ocall_ret field before issuing a ret instruction, which simply pops the return address from the stack and loads it into the instruction pointer rip. Hence, the new instruction pointer will be the value of ocall_ret, and the new stack pointer will point to the memory location immediately following the OCALL frame. From version 2.0, a more traditional epilogue is used: the base pointer (rbp) is moved into rsp, then rbp is popped from the stack, and finally a ret is issued. Therefore, rbp in the OCALL frame has to point to a memory area containing two 64-bit words: the new value for rbp, and the return address (i.e., the new instruction pointer). After returning, rsp will point 16 bytes past the rbp in the OCALL frame. Note that those addresses do not necessarily have to point to stack memory, nor to enclave memory, as enclaves can access untrusted memory.

```
typedef struct _ocall_context_t {
    /* ... */
    uintptr_t r15;
    uintptr_t r14;
    uintptr_t r13;
    uintptr_t r12;
    uintptr_t xbp; // rbp
    uintptr_t xdi; // rdi
    uintptr_t xsi; // rsi
    uintptr_t xbx; // rbx
    /* ... */
    uintptr_t ocall_ret;
} ocall_context_t;
```

Listing 1: Context structure for asm_oret. Fields not relevant to our attack are omitted.

The first operation done by asm_oret is shifting the stack pointer to the sp argument, i.e., the top of the OCALL frame. Subsequent references to the OCALL frame are made through the stack pointer. As a result, an attacker can jump to the code after the function prologue that sets up the stack and let asm_oret believe that the OCALL frame is at the top of the current stack. On SGX SDK versions earlier than 2.0, the stack pointer is shifted with a single instruction, mov rsp, rdi, at the beginning of asm_oret. This can be easily skipped by calling the second instruction instead of the real beginning of asm_oret. Starting with version 2.0 of the SDK, the code is more complex, as it also handles other tasks (such as restoring the extended processor state) before restoring the registers we are interested in. Simply skipping the stack shifting instruction would cause a crash because of other temporary registers that are set up in the meantime. However, it is still possible to skip the more complex first part and jump directly to the part that restores registers without inducing any side-effects. As such, it is always possible to abuse `asm_oret` to restore a fake OCALL frame at the top of the stack, without the need to control the first argument, by jumping to an appropriate instruction inside `asm_oret`. In the rest of this paper we will assume the attacker to always skip the initial part when reusing `asm_oret`.

An attacker who has control over the stack contents can reuse `asm_oret` to set the registers mentioned in `ocall_context_t`. An example is depicted in Figure 5. The application is vulnerable to a buffer overflow error on the stack. The attacker exploits this to overwrite the function's return address with the address of `asm_oret`, properly adjusted to account for skipped instructions. Moreover, she places a fake `ocall_context_t` immediately after the return address. Once the function returns, control is transferred to `asm_oret` with the fake OCALL frame at the top of stack, since the return address has been popped by the return instruction. Finally, `asm_oret` restores the fake context, thus granting control of those registers to the attacker.

6.2.2 CONT Primitive

The CONT primitive is based on `continue_execution`, a function used in the exception handling mechanism to restore a CPU context from an exception information structure, thus allowing exception handlers to change CPU register values. As such, it can be abused in a similar way to `asm_oret`. In comparison, `continue_execution` provides more control than `asm_oret` as the context it restores encompasses all general-purpose CPU registers.

The prototype of this function is `void continue_execution(sgx_exception_info_t *info)`, where `info` is a pointer to the exception information structure that contains the CPU context to restore. The only field used by `continue_execution` is `cpu_context`, of type `sgx_cpu_context_t`, which contains all sixteen general-purpose registers and the instruction pointer. Listings 2 and 3 show the definitions of those structures. `continue_execution` is an ideal target for a memory corruption attack as it grants control of all CPU registers. Notably, the stack pointer (`rsp`) and the instruction pointer (`rip`) are part of this context. Since the attacker can control the stack pointer, she can hijack it to attacker-controlled memory (the *fake stack*). The code will now believe that the attacker-controlled memory is the real stack, so the attacker gets control over the stack contents. This technique is known as *stack pivoting*. Since the attacker also controls the instruction pointer, all the requirements for executing an ORET primitive are met. Therefore, it is possible to chain the ORET primitive to the CONT primitive. This is an essential ingredient for our ORET+CONT loop.

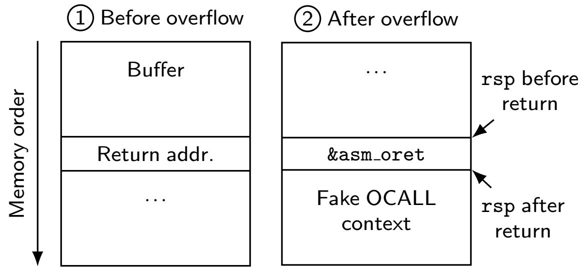

Figure 5: Stack layout when launching the ORET primitive from a stack overflow.

We noticed an issue in `continue_execution` on SDK versions prior to 1.6, which results in registers `r8`-`r15` not being restored and `rsi` being restored with the value of

```
typedef struct _exception_info_t {
    sgx_cpu_context_t cpu_context;
    sgx_exception_vector_t
        exception_vector;
    sgx_exception_type_t
        exception_type;
} sgx_exception_info_t;
```

Listing 2: Exception information structure for `continue_execution`.

```
typedef struct _cpu_context_t {
    uint64_t rax;
    uint64_t rcx;
    uint64_t rdx;
    uint64_t rbx;
    uint64_t rsp;
    uint64_t rbp;
    uint64_t rsi;
    uint64_t rdi;
    uint64_t r8;
    uint64_t r9;
    uint64_t r10;
    uint64_t r11;
    uint64_t r12;
    uint64_t r13;
    uint64_t r14;
    uint64_t r15;
    uint64_t rflags;
    uint64_t rip;
} sgx_cpu_context_t;
```

Listing 3: CPU context information structure for `continue_execution`.

r15. Since rsi can be controlled anyway (through r15), and r8-r15 are temporary registers that are not typically of interest to an attacker, this issue does not reduce the power of continue_execution reuse significantly.

As an example, continue_execution can be reused by corrupting a function pointer and hijacking it to point to continue_execution. Moreover, the attacker needs to control rdi or, equivalently, the memory pointed to by rdi. Given those preconditions, the attacker can call continue_execution with a fake sgx_exception_info_t structure and gain full CPU context control.

In another scenario, the attacker only has stack control, for example because of a stack overflow vulnerability. In that case, she can apply the ORET primitive first. Since that primitive grants control of rdi and of the instruction pointer, the attacker can chain continue_execution to get full register control.

6.2.3 Putting the Pieces Together

In this section, we finally put the primitives together to create the ORET+CONT loop to mount a code-reuse attack. The loop workflow is depicted in Figure 6. The steps of an iteration are as follows:

1. The CONT primitive is used to pivot the stack pointer into the fake stack and execute the gadget with controlled registers.
2. When the gadget returns, it will do so through the fake stack. Hence, the gadget returns to asm_oret, launching an ORET primitive.
3. The ORET primitive restores the context from the fake stack. The context is crafted to launch a CONT primitive for the next gadget to continue the loop.

Using the ORET+CONT combination is necessary because the attacker might want to control rdi, or the gadget might corrupt it; therefore, chaining CONT to CONT directly might not be possible. We discuss this aspect further in Section 6.2.4.

We now describe in detail the fake structures that the attacker needs to set up beforehand. Those can be constructed anywhere in memory, as long as they are accessible to the enclave and located at known locations.

Fake stack. The fake stack is used to chain CONT to ORET. It is composed of a sequence of frames. Each frame consists of the address of asm_oret (properly adjusted) followed by an ocall_context_t structure. The CONT in the loop invokes a gadget with the stack pointer set to the top of a frame in the fake stack. Just before the gadget returns, the address of asm_oret will be at the top of the stack and will be used as the return address. The gadget will return to asm_oret, launching an ORET primitive that will restore the context from the frame, which is at the top of the stack after returning. The situation is very similar to the stack layout in Figure 5, except that stack control is achieved with pivoting instead of a stack overflow. The context is set up so that rdi points to the exception information structure for the next gadget's CONT, and the instruction pointer is set to continue_execution. This will result in a call to continue_execution which will execute the next gadget. Note that from SDK version 2.0, the ORET context has to set rbp properly as detailed in Section 6.2.1 to control the instruction pointer.

Fake exception information. For each gadget, the attacker sets up a fake sgx_exception_info_t structure with the desired register values and the instruction pointer set to the gadget's address. The stack pointer is set to the top of the next frame in the fake stack. After continue_execution is called, the gadget will be executed with the desired register context. The return instruction at the end of the gadget will transfer control through the fake stack back to an ORET primitive, which will in turn execute the next gadget's CONT.

6.2.4 Optimizations

Gadget execution is handled by the CONT primitive, while ORET just acts as glue to chain multiple CONTs. However, it is possible to chain CONT to CONT directly, without ORET, and obtain the same effect. To do this, the attacker points rdi in the first CONT to the fake exception information for the second CONT, and returns to continue_execution from the gadget via the fake stack, as shown in Figure 7. The benefit is that ORETs are no longer needed. The fake stack only contain copies of the address of continue_execution to use them as return addresses for the gadgets. However, this optimization ties up the rdi register: the gadget must not use or corrupt it. Whether this optimization is applicable depends on the gadgets that are used. For example, it applies to the SGX-Shield exploit in Section 7.

On the other hand, if all registers needed by the gadgets can be set via the ORET primitive, it is possible to chain exclusively ORET primitives. In this case, the attacker just sets up a fake stack which runs each gadget from an ORET and makes each gadget return to asm_oret. Note that, as explained in Section 6.2.1, ORET might or might not be able to pivot the stack depending on the SDK version. In SDKs from 2.0 onwards, it is possible to manipulate rsp through rbp. On earlier versions, the stack pointer cannot be manipulated in a single call. This is problematic when exploiting a buffer overflow: if the stack cannot be pivoted, the whole fake stack has to be

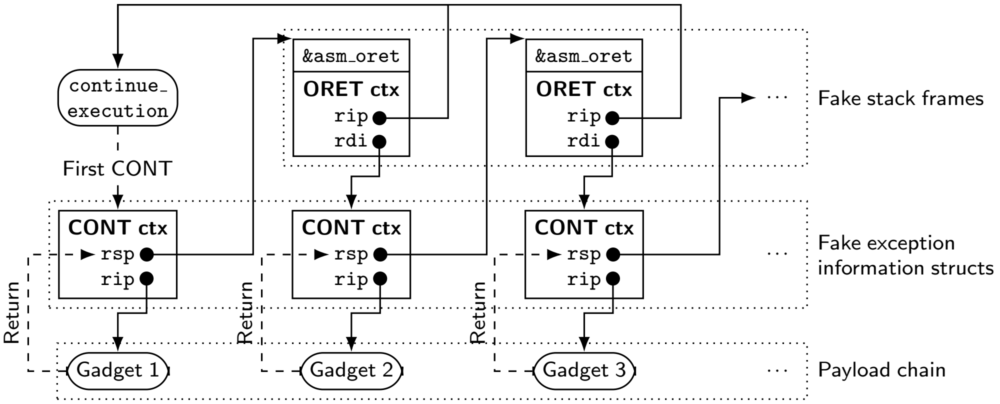

Figure 6: Workflow of the ORET+CONT loop.

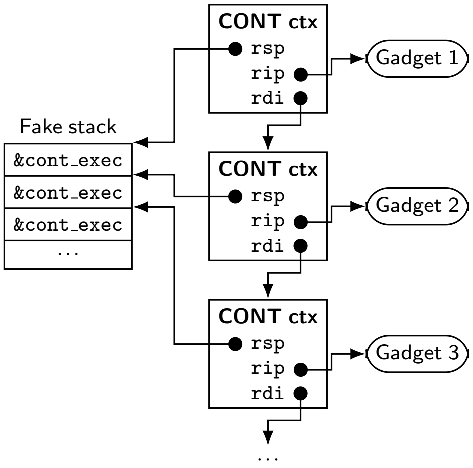

Figure 7: Simplified attack using only CONT primitives.

written to the real stack through a very large overflow. It is still possible to pivot the stack with some additional effort. For example, an attacker could use a single CONT just to set `rsp`, and then proceed with chained ORETs. Another strategy could be using the adjusted `asm_oret` to make a proper function call to the actual `asm_oret` entry point, which will restore the stack pointer from its first argument.

7 Case Study: Attacking SGX-Shield

In this section, we present an attack against an enclave hardened with SGX-Shield [26].

7.1 Overview on SGX-Shield

SGX-Shield is a hardening solution for SGX enclaves, which integrates multiple mitigation technologies:

- **Fine-grained randomization.** The enclave code is split up in 32- or 64-byte chunks, called *randomization units*, and each randomization unit is placed at an independent, randomized memory position aligned to its size. Randomization units are chained by tail jumps, since they are no longer spatially contiguous after randomization. Data objects, the heap, and the stack are also finely randomized.
- **Software DEP.** Control transfers are instrumented to enforce a W⊕X policy, i.e., writable memory areas are not executable.
- **Software Fault Isolation.** Memory accesses are instrumented to enforce an execute-only policy on code, i.e., code cannot be read or written, but only executed.
- **Coarse-grained Control Flow Integrity.** Control transfers are instrumented to force them to target the beginning of a randomization unit, so that checks cannot be circumvented by jumping in the middle of a randomization unit.

SGX does not support changing memory permissions for memory mappings after enclave initialization. This limitation will be lifted in SGX2 [22]. Because SGX-Shield needs writable code pages during loading, the enclave code will stay writable for the whole enclave's lifecycle. To protect against code injection, a software DEP policy is implemented by sandboxing data accesses inside a fixed boundary called *NRW boundary*.

7.2 Problems

Unfortunately, we identified significant differences between the SGX-Shield paper [26] and the open source implementation [25] (commit `04b09dd`, 2017-09-27). Further, there are several subtle implementation issues that we discuss below.

According to the paper's description, SGX-Shield removes the loader code from memory after loading the guest enclave. However, this is not done in the implementation. At first sight, this problem could be dismissed as trivial to solve. In fact, removing the code of the loader itself is not an issue, and we pretended the loader was erased while designing our attack. However, in the current design, the loader supplies the tRTS for the guest enclave. Specifically, OCALLs from the guest enclave are supported by routing them through the loader's tRTS. As such, one cannot simply eliminate the loader's tRTS. Moreover, since the tRTS code is part of the loader and not of the enclave, it is not randomized. Randomizing the tRTS would require significant additional work (cf. Section 8).

We also observed that the open source implementation does not enforce backwards-edge CFI, i.e., the protection of return instructions. The SGX-Shield paper describes that backwards-edge CFI can be obtained by instrumenting return instructions and forcing the return address to point to the beginning of a randomization unit. However, without extra instrumentation, a call's return address will hardly be at a randomization unit boundary. If a call is not the last instruction of a randomization unit, then the return address will point to the middle of the unit. On the other hand, if a call is the last instruction in a randomization unit, then the return address will point to the instruction immediately after the call: there is no guarantee that such an address marks the beginning of a unit. To achieve correctness, SGX-Shield would have to terminate randomization units after calls, and replace the call with a push of the address of the next randomization unit and a jump to the call target. However, the paper does not describe such an instrumentation for calls. As such, we assume that backwards-edge CFI is not present.

Hence, for our exploits explained in the remainder of this section, we do not consider backward-edge CFI protection or the absence of the tRTS.

7.3 Exploit

We now detail the steps of our attack following the workflow presented in Section 6.1.2. We assume that the attacker has discovered a stack overflow vulnerability in the hardened enclave. Moreover, we assume the SDK version is 1.6, as this is the version targeted by the public implementation of SGX-Shield that we consider. Note that our attack also applies to newer SDKs as explained in Section 6.2.1. The general idea is to use a multi-stage exploit, i.e., utilize our new code-reuse techniques to initiate a code-injection attack. This is possible since SGX-Shield enclaves feature writable code pages. As such, the exploit will be divided in two stages: the first stage, based on code reuse, injects the second-stage code, also known as *shellcode*. Once arbitrary code is injected and executed, the attacker has full control over the enclave. To demonstrate a proof-of-concept attack, our shellcode extracts secret cryptographic keys from the enclave which are used for the remote attestation process.

7.4 First Stage

Step 1: Payload preparation. The attacker starts by determining the offsets of `asm_oret` and `continue_execution`. Since they are part of the loader, which is not randomized (see Section 7.2), those offsets will not change at runtime. Next, for the code-injection attack, the attacker needs a gadget to write to memory. In general, enclaves feature a function to copy memory (e.g., `memcpy`). This can be abused to overwrite enclave code with shellcode from untrusted memory. In the case of SGX-Shield, such a function might be randomized, or placed in SDK libraries that are not essential for the guest enclave and could be erased. For this reason, we decided to use a less convenient ROP gadget from tRTS, shown in Listing 4, located in the `do_rdrand` function. This gadget writes the value in `eax` (32 lower bits of `rax`) to the address in `rcx`, sets `eax` to 1, and returns. Our chain repeatedly invokes this gadget to write the shellcode 4 bytes at a time, followed by invocation of the shellcode. Since the only gadget we use preserves `rdi`, we can use the simplification described in Section 6.2.4 to only chain CONTs. This is done only for simplicity: we have tested the exploit with the full ORET+CONT loop and confirmed it works. The address to place the shellcode at is taken from the writable SGX-Shield code pages. Since the shellcode will be run from a CONT primitive, the initial register values are controlled. Hence, the shellcode can be simplified by omitting register initialization.

Step 2: Fake structures preparation. Before exploiting the stack overflow, the attacker needs to set up the fake data structures that will be used in the exploit. Since this exploit uses an optimized chain with only CONTs, its data structure layout follows Figure 7. Those structures can be within the enclave or in the untrusted application, depending on what the attacker has control over. The only requirement is that these addresses are known. The attacker starts by creating a fake stack that contains the address of `continue_execution` repeated $n-1$ times, where n is the number of gadgets in the chain. A `sgx_exception_info_t` structure is set up for the shellcode, with `rip` set to the shellcode's

```
mov dword ptr [rcx], eax
mov eax, 1
ret
```

Listing 4: Memory write ROP gadget from `do_rdrand` in tRTS.

address and the other registers at the attacker's discretion. For each 4-byte shellcode write, the attacker sets up a sgx_exception_info_t structure such that:

- rax is set to the 4 code bytes that will be written.
- rcx points to the destination address for the current 4-byte code write.
- rdi points to the next sgx_exception_info_t structure in the write sequence; if this is the last one, rdi points to the fake exception information for the shellcode.
- rsp for the *i*-th structure points to the *i*-th address in the fake stack.
- rip points to the write gadget.

Step 3: Attack execution. The attacker now triggers the stack overflow vulnerability in the enclave. She overwrites a return address with the address of asm_oret, and places a fake ocall_context_t structure immediately after it. This structure has rdi set to the address of the fake sgx_exception_info_t structure for the first write gadget, and ocall_ret set to the address of continue_execution. This will result in continue_execution being called on that first exception information structure, which starts the chain. When continue_execution is called, it will restore the registers from the attacker's fake exception information and then transfer control to the address specified in the rip field. In this case, the write gadget will be executed with the proper rax and rcx to place 4 bytes of the attacker's code at the proper location. The stack pointer in the exception information was pointed to one of the addresses in the fake stack, which are all continue_execution. Therefore, when the write gadget returns, it will transfer control back to continue_execution. Since rdi was previously pointed to the next exception information structure, the cycle will repeat and write the next 4 bytes of code. Once all the writes are done, continue_execution will be called to execute the shellcode.

7.5 Second Stage

The shellcode has full control over the enclave. In our case, we extract the cryptographic keys used during the remote attestation process through the shellcode in Listing 5 in Appendix A. Once an attacker is in possession of those keys, she can impersonate the enclave when communicating with the remote server.

The keys are obtained with the EGETKEY leaf function. This instruction takes a KEYREQUEST structure as input, which specifies which key has to be generated. While most of the KEYREQUEST structure can be filled out by the attacker, some fields are not known outside the enclave. Therefore, the shellcode has to retrieve those values and complete the KEYREQUEST structure. This is done by generating an *enclave report* via the EREPORT leaf. This leaf requires two structures, which can be filled by the attacker: TARGETINFO and REPORTDATA. Both the EREPORT and the EGETKEY leafs only operate on enclave memory, so the shellcode has to take care of copying data in and out of the enclave. To simplify the shellcode, we use the final CONT to initialize various registers. The shellcode workflow is as follows:

1. The filled TARGETINFO and REPORTDATA structures are copied from attacker-controlled memory into enclave memory, along with a partially filled KEYREQUEST.
2. A report is generated via the EREPORT leaf.
3. The KEYREQUEST structure is completed with the information from the report.
4. The cryptographic key is generated with the EGETKEY leaf.
5. The key is copied from enclave memory into attacker-controlled memory for the attacker's consumption.
6. The enclave exits back to the attacker's code.

8 Discussion

We have shown that our attack based on the ORET and CONT primitives is highly practical and poses a severe threat to SGX enclave code. Further, our attack is even able to undermine SGX-Shield, a strong hardening scheme for SGX enclaves. Our exploitation technique can be applied to a wide range of memory corruption vulnerabilities and significantly eases SGX exploits development. In addition, our attack is highly portable. Due to the combination of the two exploitation primitives, our attack is very modular and lends itself to various simplifications and optimizations to better fit into the concrete attack situation. Consequently, we believe future mitigation schemes must take into serious consideration the implications of leaving SDK code easily accessible to attackers.

Our attack also draws a parallel to Sigreturn Oriented Programming (SROP) [4] in the SGX world. SROP abuses the UNIX signal mechanism through the sigreturn function, which restores the CPU context after an exception. The attacker can control the CPU context and chain together multiple sigreturn calls to build more complex payloads. In a similar vein, our attack abuses context-restoring mechanisms, but in the context of SGX enclaves.

8.1 SDK Versions and Platforms

Throughout this paper we focused on the Linux SDK since the SDK is open source. However, we also analyzed the Windows SDK and recognized that its low-level details are very similar to the Linux SDK. Our experiments show that only a very small adjustment is required on Windows: when chaining CONT to ORET, we require a jump to the `continue_execution` callsite rather than the function itself. This is because the exception context is passed in `rcx` on Windows - a register which is not directly controllable through ORET. However, at the callsite, `rcx` is set based on values that can be controlled via ORET.

While analyzing the low-level internals of our primitives in the Linux SDK, we also noticed several differences between SDK versions that influence our exploits:

- Setting the instruction pointer in `asm_oret` differs before and after version 2.0. However, the ORET primitive is still usable in both cases.
- In SDK version from 2.0 onwards, `asm_oret` performs some additional operations before restoring the registers. Thus, the instructions that have to be skipped differ.
- In SDKs prior to 1.6, `continue_execution` suffers from a bug that results in registers `r8` to `r15` not being set properly. Those registers are not highly relevant for executing our attack. Further, 1.6 (released in 2016) has been superseded by newer SDK versions.

8.2 SGX-Shield

Our attack against SGX-Shield exploits the lack of randomization of the tRTS. We argue that simply randomizing the SDK is not a trivial task for several reasons: first, fine-grained randomization of the tRTS likely requires manual intervention. Parts of the tRTS code are handwritten assembly, which likely requires manual splitting of the randomization units. The SDK should be made part of the guest enclave, and randomized together with the other guest's code. The loader would have its own copy of the SDK, as it is still a proper SGX enclave. The tRTS in the SDK provides the entry point code, from which the enclave starts executing when entered through `EENTER`. Initially, the entry point would be from the loader's tRTS. After the guest is loaded, the entry point has to be switched over to the guest's tRTS. The entry point address is specified in the TCS, which cannot be modified after the enclave has been initialized. Thus, SGX-Shield would have to patch its own entry point to act as a passthrough for the guest's entry point before wiping out the loader. The guest's SDK state would also need to be properly initialized. The cost of those extensions would be a slightly longer startup time, as they are just additions to the loading phase. We expect the runtime overhead of the extra entry point indirection to be completely negligible.

Our attack also exploits the backwards-edge CFI issues in SGX-Shield to hijack the control flow. The arms race between CFI defenses and attacks is still ongoing [9, 10, 11]. Hence, we believe that even in the presence of backward-edge CFI, a skilled attacker could still be able to launch our exploit, although the reusable code base has been reduced.

On another note, we argue that the current Software Fault Isolation scheme deployed in SGX-Shield can be undermined by our attack. SGX-Shield enforces an execute-only policy on code by instrumenting memory accesses. To do so, it keeps the so-called *NRW boundary* between execute-only code and read-write data. Every memory access is instrumented, so that code, which is above the NRW boundary, cannot be accessed. The boundary is kept in a fixed register (`r15`), initialized before launching the guest enclave. Since our attack can control this register, the NRW boundary can be shifted, thus disabling SFI.

8.3 Countermeasures

We now propose two complimentary mitigations to stop our attack. On the one hand, we suggest hardening measures for the SDK. On the other hand, we discuss considerations for designing hardening schemes.

The first avenue to mitigate our attack is hardening the SDK. A common strategy to make crafting fake structures harder is to integrate a secret value into the structures. The secret is then checked at runtime before performing any operation on the structure. Since the attacker does not know the secret, she cannot craft valid structures. This approach, however, can be defeated if the attacker exploits an information leakage vulnerability to read the secret from a valid structure. Moreover, in our attack scenario, the developer has to be careful that the check cannot be skipped by jumping over it. This method is therefore weak and error-prone.

A better method is *mangling* the data within the structure. The contents are stored combined with the secret in a reversible way, e.g., via XOR. The attacker would have to know the secret to craft data that, when the mangling is reversed, produces a valid structure. Leaking is also more difficult. For example, when using XOR, the attacker not only has to leak the mangled data, but also know the unmangled data to recover the secret. This method is much stronger than just embedding a secret, and its overhead would be negligible in our case, as the structures we target are not accessed very often.

The second mitigation avenue is taking the SDK code base into serious consideration when designing hardening schemes. Specifically, we focus on the problems we raised with SGX-Shield. The first step would be providing fine-grained randomization for the SDK and solving the backwards-edge CFI issue (cf. Section 7.2). Moreover, the NRW boundary has to be stored at a less accessible location. We propose the thread-local storage. This memory area is accessed via a segment selector, which cannot be altered with our attack. However, the performance implications of this choice have to be evaluated, as it would cause an extra memory access for each instrumented access.

9 Conclusion and Summary

Intel Software Guard Extensions (SGX) is a promising processor technology providing hardware-based support to strongly isolate security-critical code inside a trusted execution environment called enclave. Previous research has investigated side-channel attacks against SGX or proposed sophisticated SGX-enabled security services. However, to our surprise, memory corruption attacks such as return-oriented programming (ROP) are not yet well understood in the SGX threat model. In fact, recently presented ROP attacks against SGX rely on a strong adversarial setting: possessing kernel privileges, frequently crashing enclaves, and assuming a constant memory layout. In this paper, we systematically explore the SGX attack surface for memory corruption attacks. In particular, we present the first user-space memory corruption attack against SGX. Our attack undermines existing randomization schemes such as SGX-Shield without requiring any enclave crashes. To do so, we propose two new exploitation primitives that exploit subtle intrinsics of SGX exception handling and the interaction of enclave code to its untrusted host application. Furthermore, given a memory corruption vulnerability, our attacks apply to any enclave developed with the Linux or Windows Intel SDK for SGX. As we argue, building randomization-based defenses for SGX enclaves is challenging as it requires careful support of SDK library code and additional protection of SGX context data.

References

[1] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 267–283.

[2] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 8.

[3] Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., and Boneh, D. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014), SP'14.

[4] Bosman, E., and Bos, H. Framing signals - A return to portable shellcode. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 243–258.

[5] Brasser, F., Capkun, S., Dmitrienko, A., Frassetto, T., Kostiainen, K., Müller, U., and Sadeghi, A. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR abs/1709.09917* (2017).

[6] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies* (2017).

[7] Chen, S., Zhang, X., Reiter, M. K., and Zhang, Y. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 7–18.

[8] Costan, V., and Devadas, S. Intel SGX explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[9] Davi, L., Lehmann, D., Sadeghi, A.-R., and Monrose, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[10] Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., and Sidiroglou-Douskos, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM CCS* (2015).

[11] Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014), SP'14.

[12] Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., and Costa, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium* (2017).

[13] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2016), Springer, pp. 279–299.

[14] Intel. Intel® Software Guard Extensions SDK. `https://software.intel.com/en-us/sgx-sdk`.

[15] Intel. Intel® Software Guard Extensions SDK for Linux*. `https://01.org/intel-software-guard-extensions`.

[16] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*, December 2017. Order Number 332831-065US.

[17] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 591–604.

[18] Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., and Fetzer, C. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 205–221.

[19] Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., and Kang, B. B. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security* (2017), pp. 523–539.

[20] Lee, S., Shih, M.-W., Gera, P., Kim, T., Kim, H., and Peinado, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security* (2017), pp. 16–18.

[21] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 605–622.

[22] McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., and Rozas, C. Intel® Software Guard Extensions (Intel® SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016* (New York, NY, USA, 2016), HASP 2016, ACM, pp. 10:1–10:9.

[23] Moghimi, A., Irazoqui, G., and Eisenbarth, T. CacheZoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems* (2017), Springer, pp. 69–90.

[24] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. VC3: Trustworthy data analytics in the cloud using SGX. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 38–54.

[25] Seo, J. SGX-Shield open source repository. `https://github.com/jaebaek/SGX-Shield`. Commit `04b09dd`, 2017-09-27.

[26] Seo, J., Lee, B., Kim, S., Shih, M.-W., Shin, I., Han, D., and Kim, T. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2017).

[27] 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* (2007), ACM, pp. 552–561.

[28] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Network and Distributed System Security Symposium* (2017).

[29] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security)* (2017).

[30] Weichbrodt, N., Kurmus, A., Pietzuch, P., and Kapitza, R. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security* (2016), Springer, pp. 440–457.

[31] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy* (2015).

[32] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium* (2014), pp. 719–732.

Appendix A: Shellcode

```
; Initial register state:
; rax = 0 (EREPORT leaf)
; rbx = EEXIT return address
; rcx = 512+512+64
;       (total size of structures)
; rdx = writable 512-byte aligned enclave
;       area for temporary data
; rdi = writable 512-byte aligned enclave
;       area to copy structures into
; rsi = address of attacker's KEYREQUEST +
;       TARGETINFO + REPORTDATA
; rbp = address of attacker's key buffer
; rsp = writable area for shellcode stack
push rbx
push rdi
; Copy KEYREQUEST, TARGETINFO,
; REPORTDATA to enclave memory
rep movsb
; EREPORT
lea rcx, [rdi-64]
lea rbx, [rcx-512]
enclu
; Copy report's ISVSVN to KEYREQUEST
pop rbx
mov ax, [rdx+258]
mov [rbx+4], ax
; Copy report's CPUSVN to KEYREQUEST
vmovdqa xmm0, [rdx]
vmovdqu [rbx+8], xmm0
; Copy report's KEYID to KEYREQUEST
vmovdqa ymm0, [rdx+384]
vmovdqu [rbx+40], ymm0
; EGETKEY
push rdx
pop rcx
mov al, 1
enclu
; Copy key to attacker's memory
movdqa xmm0, [rdx]
movdqu [rbp], xmm0
; EEXIT to attacker's code
pop rbx
mov al, 4
enclu
```

Listing 5: Shellcode for cryptographic key extraction (74 bytes).