Hacking in Darkness: Return-oriented Programming against Secure Enclaves

Jaehyuk Lee[†] *Jinsoo Jang*[†] *Yeongjin Jang*[⋆] *Nohyun Kwak*[†] *Yeseul Choi*[†] *Changho Choi*[†]
Taesoo Kim[⋆] *Marcus Peinado*[‡] *Brent Byunghoon Kang*[†]

[†]KAIST [⋆]Georgia Institute of Technology [‡]Microsoft Research

Abstract

Intel Software Guard Extensions (SGX) is a hardware-based Trusted Execution Environment (TEE) that is widely seen as a promising solution to traditional security threats. While SGX promises strong protection to bug-free software, decades of experience show that we have to expect vulnerabilities in any non-trivial application. In a traditional environment, such vulnerabilities often allow attackers to take complete control of vulnerable systems. Efforts to evaluate the security of SGX have focused on side-channels. So far, neither a practical attack against a vulnerability in enclave code nor a proof-of-concept attack scenario has been demonstrated. Thus, a fundamental question remains: *What are the consequences and dangers of having a memory corruption vulnerability in enclave code?*

To answer this question, we comprehensively analyze exploitation techniques against vulnerabilities inside enclaves. We demonstrate a practical exploitation technique, called Dark-ROP, which can completely disarm the security guarantees of SGX. Dark-ROP exploits a memory corruption vulnerability in the enclave software through return-oriented programming (ROP). However Dark-ROP differs significantly from traditional ROP attacks because the target code runs under solid hardware protection. We overcome the problem of exploiting SGX-specific properties and obstacles by formulating a novel ROP attack scheme against SGX under practical assumptions. Specifically, we build several oracles that inform the attacker about the status of enclave execution. This enables him to launch the ROP attack while both code and data are hidden. In addition, we exfiltrate the enclave's code and data into a shadow application to fully control the execution environment. This shadow application emulates the enclave under the complete control of the attacker, using the enclave (through ROP calls) only to perform SGX operations such as reading the enclave's SGX crypto keys.

The consequences of Dark-ROP are alarming; the attacker can completely breach the enclave's memory protections and trick the SGX hardware into disclosing the enclave's encryption keys and producing measurement reports that defeat remote attestation. This result strongly suggests that SGX research should focus more on traditional security mitigations rather than on making enclave development more convenient by expanding the trusted computing base and the attack surface (e.g., Graphene, Haven).

1 Introduction

Computer systems have become very complex. Even simple, security-sensitive applications typically inherit the huge trusted computing base (TCB) of the platforms they run on. Trusted execution environments such as ARM TrustZone [2] or Intel TXT [14] were invented to allow small programs to run in isolation from the much larger underlying platform software. However, the adoption of these systems has been limited, as they were either closed or required trusted hypervisors or operating systems that have not materialized in the mass market.

Intel Software Guard Extensions (SGX) [16] is a new processor feature that isolates security-critical applications from system software such as hypervisors, operating systems, or the BIOS. SGX has been integrated into recent Intel processor models and is seeing mass-market deployment. It is widely seen as the technology that can finally enable applications with a small TCB in the mass market. A number of systems have been using SGX to protect applications from threats ranging from untrusted cloud providers to compromised operating systems [3, 6, 17, 27, 30, 36, 39].

Recent work has explored the practical limitations of this vision. Several authors [26, 33, 38] have identified side channels that can leak large amounts of sensitive information out of the application's isolated execution

environment (*enclave*). A synchronization bug has been shown to lead to a breakdown in enclave security [37]. However, a fundamental question about the security of SGX remains unanswered: *What is the effect of having a memory-corruption vulnerability in an enclave and how dangerous is it?*

This question is important, as such vulnerabilities have been found in a wide range of applications, including security applications [4, 12, 13]. Furthermore, a major branch of SGX-based system design runs unmodified legacy applications and their complex operating system support inside enclaves [6, 36]. The enclave software of such systems is bound to have memory corruption vulnerabilities.

In a regular environment, such vulnerabilities often result in an attack that changes the control flow of a victim program to execute arbitrary code. However, enclaves in SGX differ from such environments in several important ways. In particular, SGX protects the entire memory contents of the enclave program. Memory values and registers that are required to launch an attack are completely hidden from attackers. More important, recent SGX-based systems even keep the enclave code secret from attackers. For example, under VC^3 [30], the program binaries are encrypted. This poses a problem for ROP attacks [8, 9, 32], as the attacker needs to find a vulnerability and gadgets in the victim's code.

In this paper, we comprehensively analyze the aftermath of exploiting a vulnerability in enclave code by demonstrating a practical attack, called Dark-ROP. Dark-ROP can completely disarm the security guarantees of SGX. In essence, Dark-ROP exploits a control-flow hijacking vulnerability in the enclave software through return-oriented programming (ROP). Since SGX prevents all access to enclave code and data from outside the enclave, we cannot directly apply typical ROP attacks.

To overcome these challenges, we construct a novel method for finding a vulnerability and useful ROP gadgets in fully encrypted binaries (unknown code) running under SGX. The method constructs three oracles that (a) detect the number of register pops before a ret instruction, (b) reveal enclave register values, and (c) leak the secret enclave memory contents. The method requires *no knowledge of the content* of the binary running in the enclave. Dark-ROP can chain the gadgets found in this way and utilize them to invoke security-critical functions such as data sealing and generating measurement reports for remote attestation.

In addition, we construct a *shadow application* (i.e., SGX Malware) that runs outside an enclave but fully emulates the environment of an SGX enclave. This demonstrates the ability of Dark-ROP to fully control the enclave program. Dark-ROP utilizes ROP chains to copy the complete enclave state, including both code and data to unprotected memory. In addition to breaching enclave confidentiality, this also enables Dark-ROP to emulate the enclave software. It can run the enclave's code outside the enclave, except for a small number of SGX instructions. The latter are used for attestation and for obtaining the enclave's crypto keys. Dark-ROP emulates these instructions by invoking ROP calls into the victim enclave.

The shadow application runs in unprotected memory under the control of the attacker. When a remote server requests a measurement report to check the integrity of the victim enclave, the shadow application first receives the request (as a man-in-the-middle), and then invokes an ROP call that generates the correct measurement report in the victim enclave and sends a reply to the remote party to complete the attestation protocol. This man-in-the-middle construction allows attackers to have complete flexibility in executing any code of their choice in the shadow application because it is not protected by SGX at all. At the same time, the remote party cannot detect the attack through the remote attestation because the shadow application can use the real enclave to generate the correct measurement report.

We summarize the contributions of the Dark-ROP attack as follows:

1. **First ROP demonstration against an SGX program on real hardware.** The Dark-ROP attack can completely disarm the security guarantees of SGX. This includes 1) exfiltrating secret code and data from enclave memory, 2) bypassing local and remote attestation, and 3) decrypting and generating the correctly sealed data.
2. **New ROP techniques.** We devise a new way to launch a code-reuse attack by 1) blindly finding a vulnerability and useful gadgets from an encrypted program in the enclave and 2) constructing a shadow enclave that poses as a man-in-the-middle to masquerade the entire application of the enclave.
3. **Informing the community.** There is a temptation to focus on convenience (e.g., running unmodified programs on SGX via library OSes [3, 6, 36]) rather than security (e.g., verification of enclave programs [34, 35]).

   While SGX-like execution environments may make exploitation more difficult, software vulnerabilities continue to be a real threat. Thus, there is a need for well-studied security mechanisms that are tailored to the SGX environment.

We organize the rest of the paper as follows. §2 provides background on SGX. §3 discusses the challenges and the threat model of Dark-ROP. §4 illustrates the design of Dark-ROP. §5 describes various ways to further

develop this attack for malicious uses. In §7, we discuss the feasibility and effectiveness of our attack. §8 covers related work. We conclude in §9.

2 Background

In this section, we present the background on SGX that is necessary to further the understanding of Dark-ROP.

Intel SGX. Intel Software Guard Extensions (SGX) is an extension of the x86 instruction set architecture (ISA), which enables the creation of trusted execution environments (TEE), called *enclaves*. An enclave has an isolated memory space and execution runtime. SGX protects programs running in enclaves from attacks that undermine the integrity and the confidentiality of code and data of the program. For example, SGX prevents enclaves from being tampered with by privileged software (e.g., kernel), and from many physical attacks such as the *cold-boot attacks*.

2.1 Security Features of SGX

Memory encryption/isolation in SGX. SGX provides hardware-based access control mechanism and memory encryption to strongly guarantee the confidentiality and integrity of the entire memory used by an enclave program (*Enclave Page Cache (EPC)*).

The SGX processor enforces an access control policy that restricts all access to an enclave's memory to code running inside that enclave. That is, no other software, including the operating system, can read or write enclave memory. This access restriction is enforced by the Memory Management Unit (MMU) integrated in the processor package, which cannot be manipulated by the system software. Specifically, page miss handler (PMH) [23] checks an access permission of the EPC pages when any software requests read or write access to the enclave memory.

In addition, a memory encryption engine (MEE) [11, 15] that is a extension of the memory controller encrypts enclave code and data before they are being written to main memory. This reduces the hardware TCB of SGX to the processor package and prevents a variety of attacks such as cold boot or DMA attacks.

Ensuring program integrity through attestation. Attestation is a secure assertion mechanism that confirms the correct application has been properly instantiated on a specific platform [1].

The purpose of attestation in SGX is twofold: ensuring that an enclave is running an expected program on a certified SGX platform with a correct configuration and securely sharing a secret to build a secure communication channel between an enclave and a remote entity (e.g., the owner of the enclave).

A complete end-to-end SGX attestation involves a long series of steps, most of which are not relevant for this paper. The one step that is relevant to Dark-ROP is that an enclave needs to obtain a cryptographic message authentication code (MAC) from the processor as part of the attestation. The enclave calls the *EREPORT* instruction to obtain the MAC. *EREPORT* computes the MAC over a data structure that includes the calling enclave's cryptographic identity (digest) with a processor key that is not revealed to the caller.

Data sealing. SGX provides the means for securely exporting sensitive data from an enclave by encryption (i.e. *data sealing*).

The processor provides each enclave with crypto keys that are unique to the enclave's cryptographic identity (digest). That is, different enclaves will receive different keys. Enclave code can use these keys to implement data sealing: It can cryptographically protect (e.g., encrypt, MAC) data before asking untrusted code to store them persistently. At a later time, a different instance of the same enclave (with the same enclave digest) can obtain the same key from the processor and decrypt the data. Enclaves can use the *EGETKEY* SGX leaf function to access their keys.

Deploying an encrypted binary in SGX. Several researchers have pointed out and built systems [5, 6, 24, 29, 30] that can deploy a completely encrypted program to the SGX platform. This can increase program security by preventing attackers from reverse engineering the program.

In short, the enclave owner builds the enclave with a simple plaintext loader binary. The loader will copy a second, encrypted binary into enclave memory and decrypt it inside the enclave with a key that it can obtain from the enclave owner using remote attestation. The loader then invokes the second binary. Optionally, the loader code can be erased from enclave memory to deprive attackers of known gadget building material.

This process requires memory that is at some time writable and at another time executable. Current SGX specification (*SGX1 [19]*) does not allow changing memory page permissions after an enclave has been created. Thus, the pages into which the second binary is loaded have to be made writable and executable. A new SGX specification *(SGX2 [20])*, promises to support the modification of page permissions of running enclaves. That would allow the deployment of encrypted binaries without requiring pages to be both executable and writable.

In summary, SGX makes it possible to deploy encrypted binaries, which means that attackers may never be able to see the code running inside the enclave they are

Instruction	RAX value	Leaf function	Description
ENCLU	0x0	EREPORT	Create a cryptographic report
	0x1	EGETKEY	Retrieve a cryptographic key
	...		
	0x4	EEXIT	Synchronously exit an enclave
	...		
	0x6	EMODPE	Extend an EPC access permission

Figure 1: `ENCLU` instruction and its leaf functions. To invoke a leaf function of interest through the `ENCLU` instruction, an application developer can load the index of the function into the `rax` register and then execute `ENCLU`. For example, the value of `rax` is required to be 0x4 to invoke `EEXIT`.

trying to attack.

2.2 Instruction Specifications

SGX adds two new instructions, `ENCLU` and `ENCLS`, to the x86 ISA [19, 20]. `ENCLU` handles the user-level operations (i.e., Ring 3) such as deriving encryption keys and generating measurement reports. `ENCLS`, on the other hand, handles privileged level operations (i.e., Ring 0) such as creating enclaves, allocating memory pages. While SGX introduces many operations for creating enclaves and managing them, these two instructions work as gates that help dispatch a variety of functions, which are called *leaf functions* [19, 20].

Leaf functions. Figure 1 shows how a user-level process can invoke each leaf function through an `ENCLU` gate. To call a leaf function, a developer can load the index of a leaf function into the `rax` register and call `ENCLU`. For example, setting `rax` to `0x0` will call `EREPORT`, `0x1` will call `EGETKEY`, etc. Each leaf function requires different parameters, which are passed through the `rbx`, `rcx`, and `rdx` registers. For example, `EEXIT`, one of the leaf functions of `ENCLU`, requires two parameters: 1) a target address outside the enclave and 2) the address of the current Asynchronous Exit Pointer (AEP). These two parameters are passed through the `rbx` and `rcx` registers. After setting the required parameters, the developer can now set rax to the index of the leaf function (in this case, 0x4). Finally, executing the `ENCLU` instruction will execute the `EEXIT` leaf function. This calling convention for leaf functions is very similar to invoking a system call in Linux or Windows on the x86 architecture.

3 Overview

In this section, we present an overview of Dark-ROP with a simple enclave program that has a buffer overflow vulnerability as an example.

```
1  // EENTER can run this function
2  Data* import_data_to_enclave(char *out_of_enclave_memory)
3  {
4      // data to be returned
5      Data *data = new Data();
6      // a stack buffer in the enclave
7      char in_enclave_buffer[0x100];
8
9      // possible buffer overflow
10     strcpy(in_enclave_buffer, out_of_enclave_memory);
11
12     // ...
13     // do some processing
14     // ...
15     return data;
16 }
```

Figure 2: An example enclave program that has a buffer overflow vulnerability. The untrusted program can call an exported function `import_data_to_enclave()` in the enclave through the `EENTER` leaf function. The function will copy data from memory outside the enclave to an in-enclave stack buffer. However, the buffer can overflow during the copy because the size of data to be copied is not checked.

3.1 Launching the ROP attack in SGX

Figure 2 shows an example of a potentially exploitable vulnerability. In particular, the function `import_data_to_enclave()` reads the data from outside the enclave and creates a class object (i.e., `Data` in the code) by parsing the raw data. An untrusted program can invoke a function in the enclave (from outside the enclave) if an enclave program has exported the function. To call the function in the enclave, the untrusted program can set the `rbx` register as the address of the Thread Control Structure (TCS), which is a data structure that contains the entry point of the enclave (e.g., the `import_data_to_enclave()` function in this case) and its argument (i.e., the attack buffer as `out_of_enclave_memory`) as a pointer of the untrusted memory. Then, running `EENTER` will invoke the function in the enclave. In the function, the data at the untrusted memory will be copied (see line 10) using the `strcpy()` function, which does not check the size of the data to be copied so that the attacker can exploit this buffer overflow vulnerability. While the vulnerability does not have to be in this form specifically, the code is very simple to represent a general example of an enclave program that has an exploitable vulnerability.

To launch the ROP attack on the vulnerability, the attacker can fill the attack buffer to more than the size of the buffer in the enclave, which is `0x100`, to overwrite the return address and then build the stack with ROP gadgets and function arguments to control the program execution at the attacker's will.

However, the ROP attack against enclaves will not simply work in the typical way because the information for the execution environment as well as the program itself is encrypted, so it is hidden to attackers.

Challenge: encrypted binary makes the ROP attack difficult. In the example, since we know the source code of the program, we can easily find the location and the triggering condition of the vulnerability. However, in the most secure configuration of the SGX platform (deploying an encrypted binary as in §2.1), the assumption that we know the location of the vulnerability and the condition that triggers vulnerability does not hold. This makes the launching of an ROP attack harder even if there is a buffer overflow vulnerability because attackers are required to find the vulnerability while having no knowledge of the target program.

Additionally, finding gadgets over the encrypted program is another challenge that is orthogonal to finding vulnerabilities. Suppose that an attacker could find the location and the condition for triggering a vulnerability. To successfully exploit the vulnerability and take control of the program, the attacker is required to launch a code reuse attack (if there is no code injection vulnerability) through return-oriented programming (ROP).

Unfortunately, chaining the ROP gadgets to execute an arbitrary function is exceptionally difficult in enclaves because the program binary is encrypted. Deploying a program binary in a fully encrypted form in SGX results in the code in the binary being completely unknown to the attacker. In other words, the attacker has to find gadgets for their execution and chain them together under the blindness condition.

Although a recent work on Blind ROP [7] demonstrates an ROP attack against unknown code, the attack relies critically on properties of certain server applications that are based on the `fork()` system call, which does not hold for SGX enclaves.

3.2 The Dark-ROP Attack

Consequently, to launch a successful ROP attack against the enclaves in SGX, the attacker must overcome the aforementioned challenges. In Dark-ROP attack, we resolve the challenges as follows.

Finding a buffer overflow vulnerability. To find a buffer overflow vulnerability in an encrypted enclave program, the Dark-ROP attack exploits the exception handling mechanism of SGX as follows.

For an enclave program, it has a fixed number of (exported) entry points (i.e., functions of enclave program) specified in the enclave configuration. Because these are the only point at which an untrusted OS can supply an input to the enclave program, we enumerate those functions and apply fuzzing to its argument to find any memory corruption vulnerability. In fuzzing functions, we can detect a vulnerability by exploiting the exception handling mechanism of the enclave. Since an enclave program runs as a user-level program, which cannot handle processor exceptions, when it encounters memory corruption (i.e., page fault) on its execution, the enclave gives back the execution to the untrusted operating system to handle the fault. This fall-back routine for handling the exception is called Asynchronous Enclave Exit (AEX). If we detect any AEX caused by a page fault on fuzzing, this means that there was a memory corruption so that we set the function and the argument that currently fuzzed as a candidate for the buffer overflow vulnerability.

Next, to detect vulnerability triggering conditions such as the size of the buffer and the location of the return address, we exploit the value of the CR2 register at the AEX handler, the register that stores the source address of a page fault. By constructing the fuzzing buffer to contain an invalid memory address (e.g. `0x41414000`) in the buffer, we can determine the potential target of the return address if the exception arose from the supplied value (i.e., if the value of CR2 is `0x41414000`).

Finding gadgets in darkness. After finding a buffer overflow vulnerability in an enclave program, the Dark-ROP attack finds gadgets to exploit the vulnerability. To overcome the challenge of finding gadgets against the unknown binary, we make the following assumptions on the code in the binary.

First, the code must have the `ENCLU` instruction. This is always true for the binaries in enclaves because the enclave program can call the leaf functions only with the `ENCLU` instruction. Without having the instruction, the enclave cannot enjoy the features provided by SGX.

Second, the code should have the ROP gadgets that consist of one or multiple "pop a register" (i.e., `pop rbx`) instructions before the return instruction, especially for the `rax`, `rbx`, `rcx`, `rdx`, `rdi`, and `rsi` registers. The reason we require pop gadgets for such registers is that these registers are used for the index of the leaf function (`rax`), for arguments passing (the other registers) for the leaf function, and a library function in the x86-64 architecture. For the `rbx`, `rcx`, and `rdx` registers, the `ENCLU` instruction uses them for passing the arguments. Similarly, for the `rdi` and `rsi` registers, the library functions use them for passing the arguments. To successfully call the leaf functions and library functions, the value of these registers must be under the control of the attacker.

The second assumption is also a very common case for the enclave binary because these registers are callee-saved registers. As mentioned above, the leaf functions and the library functions use them for passing the argument so that the callee must have a routine that restores the registers, and this is typically done by running multiple "pop a register" instructions before the return of the function. Thus, the code typically includes the "pop a register" gadget for these registers. Furthermore, since `rax` is reserved

for passing the return value of the function in the x86-64 architecture, having an instruction such as `mov rax, rbx` before the function epilogue is a very common case.

Third, we assume that the program in the enclave has a function that operates as a `memcpy` function (e.g., `memcpy`, `memmove`, or `strncpy`, etc.). The assumption still targets a typical condition because the isolated architecture of SGX memory requires frequent copying of memory between the trusted in-enclave area and the untrusted area.

We believe the assumptions we made for the gadgets targets a typical condition of enclave programs because without such gadgets, the programs will be broken or run unconventionally.

Based on the assumption of gadgets, we attempt to find the useful ROP gadgets without having any knowledge of the code in the binary, so we called this attack "Dark-" ROP. To this end, we construct three oracles that give the attackers a hint of the binary code to find the useful gadgets: 1) a page-fault-based oracle to find a gadget that can set the general purpose register values; 2) the `EEXIT` oracle can verify which registers are overwritten by the gadgets found by 1); and 3) the memory oracle that can find the gadget has a memory copy functionality to inject data from untrusted space to the enclave or to exfiltrate the data *vice versa*. For the details of the oracles, please refer to §4 for the further descriptions.

By utilizing these three oracles, the Dark-ROP attack achieves the ability to execute security-critical functions such as key derivation for data sealing and generating the correct measurement report for attestation, and arbitrarily read or write data between the untrusted memory and the memory of the enclaves.

3.3 Threat Model

To reflect the environment of SGX deployed in the real world, the Dark-ROP attack is based on the following assumptions:

1. The target system is equipped with the processor that supports SGX, and we assume that the hardware is not vulnerable. Additionally, we also exclude the case that requires physical access to the machine because the Dark-ROP attack is a pure software-based attack.
2. SGX and the enclave application are configured correctly. That is, we assume that all software settings that affect the enclave such as BIOS settings and the setting of page permissions for the enclave etc. are configured correctly, as described in the Intel manual [19–22] to guarantee the security promised by SGX if the application has no vulnerability.
3. The application harvests the entire security benefit of SGX. That is, the application that runs in the enclave is distributed in an encrypted format and removing the loader program after launching the payload, which makes it completely hidden to the attacker, and the application uses data sealing for protecting application data as well as remote attestation to verify the running status of the enclave.
4. However, the application that runs inside the enclave has an exploitable memory-corruption vulnerability.
5. The attacker has full control of all software of the system, including the operating system and the untrusted application that interacts with the enclave, etc., except the software that runs inside the enclave.
6. The target application is built with a standard compiler (e.g. Visual Studio for SGX, or `gcc`), with the standard SDK that is supplied by Intel.

The threat model of Dark-ROP is pragmatic because it assumes the standard, and secure configuration of SGX for the attack target, as well as assuming only the software-level attacker. The extra assumption that we add to the standard is that the software in the enclave has an exploitable vulnerability. Since removing all vulnerabilities from the software is an inextricable challenge, we believe that the assumptions depict the common best practices of using of SGX.

4 Attack Design

In this section, we illustrate how an attacker can launch the ROP attack by overcoming the challenges of the attack in the SGX environment. We first describe how an attacker can find the gadgets required for the Dark-ROP attack by exploiting the three oracles that can provide the hints with respect to the code in the unknown (encrypted) binary in the enclave. After that, we demonstrate a proof-of-concept example that invokes security-critical functions within the enclave through the vulnerability by chaining the ROP gadgets.

4.1 Finding gadgets in a hidden enclave program

To find gadgets from the completely hidden binary in an enclave, we devised three techniques that can turn an enclave into an oracle for finding a gadget: 1) Reading the `cr2` register at the page fault handler to find the gadget with multiple register pops to control the value of registers. 2) Leaking the register values at the page fault handler by calling the `EEXIT` leaf function to identify which registers are changed by 1. 3) Examining the memory outside the enclave to find a function in the `memcpy()` family to perform arbitrary read/write on the enclave.

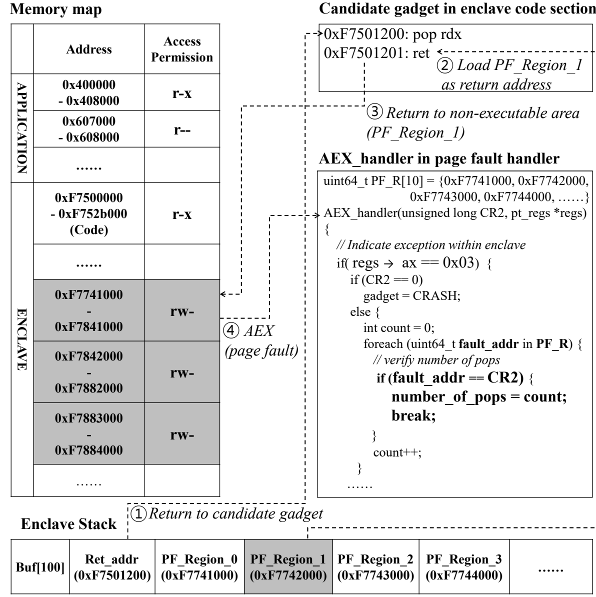

Figure 3: An overview of page fault oracle and the AEX handler. The workflow for identifying pop gadgets by using the *page fault oracle* is as follows: (1) The attacker sets an arbitrary address in the code section on the stack to probe if the address is for a pop gadget (e.g. `0xF7501200` in the figure) and then set several non-executable addresses in `PF_region`. (2) Because the probed address in the figure contains a single pop and a return instruction, the processor attempts to pop the first address in `PF_region` (i.e., `PF_region_0`) then return to the second address on the stack, `PF_region_1` (i.e., `0xF7742000`). (3) Returning to the `PF_region_1` address emits the page fault exception because the address is non-executable. (4) At the exception handler, the attacker can locate this address from the `cr2` register in the exception context so that the attacker can identify that only one pop is in the gadget.

Page fault oracle for changing register values. We first find gadgets that can set a value to a specific register from the values in the stack. For instance, a pop gadget like `pop rbx; pop rcx; pop rdx; retq;` can change the value of the `rbx`, `rcx`, and `rdx` registers at once if values are set at the attack stack by exploiting a buffer overflow vulnerability.

To find such gadgets, we turn the Asynchronous Enclave Exit (AEX) and page fault handler into an oracle for detecting the gadgets. An interesting property of the Intel processor is that when a page fault exception arises, the `cr2` register stores the address of the page that generated the fault. On the other hand, if a page fault arises in the enclave, the AEX happens and it clears the least 12 significant bits of the `cr2` register and overwrites the General Purpose Registers (GPRs) with the synthesized value to protect its execution context. Therefore, for the page fault that arises in the enclave, we can identify which address triggered the page fault in a page granularity by examining the value in the `cr2` register at the page fault handler (i.e., AEX handler in this paper).

To turn this into a gadget-finding oracle, we set the attack stack as in Figure 3. In essence, by exploiting the memory corruption bug, we set the return address to be the address that we want to probe whether it is a pop gadget or not. The probing will scan through the entire executable address space of the enclave memory. At the same time, we put several non-executable addresses, all of which reside in the address space of the enclave, on the stack.

Because the untrusted operating system manages all the memory allocations, the attacker knows the coarse-grained memory map of the enclave (on the left side of the Figure 3) so that the attacker can easily identify the non-executable enclave memory pages (e.g., enclave stack or heap pages). We call this memory region as `PF_region` and, `PF_R` array in the code contains the list of non-executable page addresses.

For instance, we put `0xf7741000`, `0xf7742000`, `0xf7743000`, and `0xf7744000`, etc. on the enclave stack to set the register values if it is a `pop` gadget (see at the bottom of the Figure 3. For example, if the gadget at the return address is `pop rdx; ret;`, then `0xf7741000` will be stored into the `rdx` register, and the processor will attempt to return to the address of `0xf7742000`. However, the address `0xf7742000` is a non-executable address; returning to such an address will cause the processor to generate the page fault. Then, the AEX handler will catch this page fault. At the AEX handler, the attacker is able to distinguish the number of `pops` in the gadget by examining the value in the `cr2` register. In the case of the example, the value is `0xf7742000`, the second value on the stack, which means that the gadget has only one pop before the return because the first value, `0xf7741000`, is popped. Taking another example, when the gadget has three pops, the first three values on the stack will be removed so that the value in the `cr2` register will be `0xf7743000`.

Using this method, the attacker can identify the number of pops before the return on the gadgets. However, the oracle does not allow the attacker to figure out which registers are being popped. Moreover, the gadget found by this method could not be a pop gadget because the page fault can be triggered in other cases such as `pop rax; mov rbx,QWORD PTR [rax+0x4]` (fault by `mov` instruction). In the next oracle, we will remove the uncertainty of the gadgets found by this oracle.

Identifying the gadgets and the registers on `EEXIT`. The second oracle we build is for identifying pop gadgets among the gadget candidates found from the first AEX oracle. The second oracle exploits the fact that the values in registers are not automatically cleared by the hardware on the execution of the `EEXIT` leaf function. As a result,

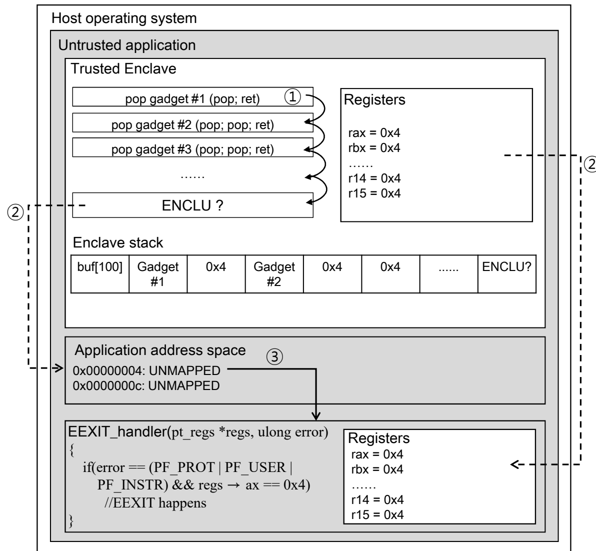

Figure 4: An overview of searching an `ENCLU` gadget and the behavior of `EEXIT`. (1) The attacker chains multiple pop gadgets found in Figure 3, as many as possible, and put the value `0x4` as the number of pops in the gadget. (2) If the probing address (the last return address) contains the `ENCLU` instruction, then it will invoke `EEXIT` and jump to the address specified in `rbx` (i.e., `0x4` because of the pop gadgets). (3) The execution of `EEXIT` generates the page fault because the exit address in `rbx` (`0x4`) does not belong to the valid address region. (4) At the page fault handler, the attacker can be notified that `EEXIT` is invoked accordingly by examining the error code and the value of the `rax` register. The error code of `EEXIT` handler contains the value that indicates the cause of page fault. In this case, the page fault is generated by pointing an invalid address `0x4` as jump address (i.e., the value of `rbx` register). So if the error code contains the flags for `PF_PROT` (un-allocated), `PF_USER` (userspace memory), and `PF_INSTR` (fault on execution), and the value of `rax` is `0x4` (the value for `EEXIT` leaf function), then the attacker can assume the probed address is where the `ENCLU` instruction is located.

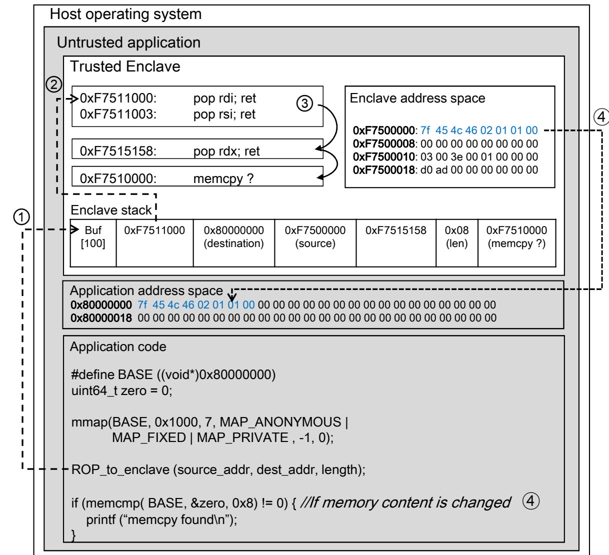

Figure 5: An overview of finding `memcpy()` gadget. (1) The attacker exploits a memory corruption bug inside the enclave and overwrites the stack with a gadget chain. (2) The gadgets in the chain sets the arguments (`rdi, rsi, rdx`) as the destination address (`0x80000000`) in `rdi`, the source address (`0x75000000`) in `rsi`, and the size (`0x8`) in `rdx` to discover the `memcpy()` gadget. (3) On the probing, if the final return address points to the `memcpy()` gadget, then it will copy the 8 bytes of enclave code (`0xf7500000`) to the pre-allocated address in application memory (`0x80000000`), which was initialized with all zero. (4) To check if the `memcpy()` gadget is found, the attacker (application code) compares the contents of the memory region (`0x80000000`) with zero after each probing. Any non-zero values in the compared area results the discovery of the `memcpy()`.

the attacker can identify the values of the registers that were changed by the pop gadget that is executed prior to `EEXIT`. This helps the attacker to identify the pop gadgets among the candidates and the registers that are popped by the gadgets.

To build this oracle, we need to find the `ENCLU` instruction first because the `EEXIT` leaf function can only be invoked by the instruction by supplying the index at the `rax` register as `0x4`. Then, at the `EEXIT` handler, we identify the pop gadgets and the registers popped by the gadget. To find the `ENCLU` instruction, we take the following strategy. First, for all of the pop gadget candidates, we set them as return addresses of a ROP chain. Second, we put `0x4`, the index of the `EEXIT` leaf function, as the value to be popped on that gadgets. For example, if the gadget has three pops, we put the same number (three) `0x4` on the stack right after the gadget address. Finally, we put the address to scan at the end to probe whether the address is a `ENCLU` gadget.

The mechanism behind the scene is like the following. The value `0x4` is the index for the leaf function `EEXIT`. What we aim to change the value for is the `rax` register because it is the selector of the `EEXIT` leaf function. For the combinations of pop gadget candidates and the address of probing, the enclave will trigger `EEXIT` if the address of a gadget that changes `rax` and the address of `ENCLU` sits on the stack. The attacker can catch this by using an `SIGSEGV` handler because the return address of `EEXIT` (stored in the `rbx` register) was not correct so that it will generate the exception. If the handler is invoked and the value of `rax` is `0x4`, then the return address placed at the end of the attack stack points to the `ENCLU` instruction.

After we find the method to invoke `EEXIT`, we exploit the `EEXIT` gadget to identify which registers are popped by the pop gadget. This is possible because, unlike AEX, the processor will not automatically clear the register values on running the `EEXIT` leaf function. Thus, if we put a pop gadget, and put some distinguishable values as its items to

be popped, for instance, 0x1, 0x2, and 0x3, and then run the EEXIT at the end, we can identify the popped registers by the values.

For example, if the pop gadget is pop rdi; pop rsi; pop rdx; ret, then at the handler, we can see the value of 0x1 at rdi, value of 0x2 at rsi, value of 0x3 at rdx. Accordingly, we can determine that the gadget pops the rdi, rsi, and rdx registers.

By using this oracle, we probe all candidates of pop gadgets until we can control all six registers that are required to launch the Dark-ROP attack.

Untrusted memory as a read/write gadget oracle. The last oracle we build is the memory-based one to find a function that can copy data between the enclave and the untrusted memory.

To find such a function, we build an ROP chain that copies data from the memory in the enclave to the untrusted area only if the probed address (set as a return address) is matched with the starting of the memcpy() function. In particular, we set the stack to have an address at the untrusted area for the first argument (i.e., the destination of memcpy()), an address in the enclave for the second argument (i.e., the source of memcpy()), and the size of data to be copied for the third argument in order to probe the return address as one of the functions in the memcpy() family. Then, we set the value of the destination address (at the untrusted area) with all zero bytes. After this, we probe each address of the enclave to find the memcpy() function. The probing finishes when we detect any change in the untrusted memory because the change proves that the memory copy is executed.

The memcpy() ROP gadget allows attackers to have an arbitrary read/write in both directions in between the enclave and the untrusted memory space because the attacker can set the source and destination addresses arbitrarily at the attack stack.

4.2 A proof-of-concept Dark-ROP attack

After finding all gadgets, including the register pop gadget, ENCLU, and memcpy(), an attacker can control the enclave in two ways. First, the attacker can run any leaf function through ENCLU by setting arbitrary values in the registers that are used for setting parameters. Second, the attacker can copy-in and copy-out the data from the untrusted memory to the trusted in-enclave memory by using the memcpy() gadget. In the Dark-ROP attack, we chain those two capabilities together to run the security-critical operations in SGX and then extract generated (secret) data from the enclave to the untrusted space solely based on launching the ROP attack. In particular, for the proof-of-concept demonstration, we execute EGETKEY, a leaf function for encryption key derivation, and extract the generated key from the enclave Note that EGETKEY must be executed in the enclave because the return value, which is an encryption key, is unique to the enclave and tied to the hardware.

Leaking the encryption key for data sealing. The EGETKEY leaf function handles the generation of the encryption key used for data sealing and verifying the REPORT in attestation. The requirement for calling the EGETKEY function is that, first, the value of rax register, which is the selector of ENCLU, should be set as 0x1. Second, the rbx register should point to the address of KEYREQUEST, which is a metadata that contains configurations for key generation, and the address must be aligned in 128 bytes. Third, the rcx register should point to a writable address in the enclave because the processor will store the generated key into that address.

To call EGETKEY through ROP gadgets correctly, we do use the following steps. We first construct a KEYREQUEST metadata object in the untrusted space and place a memcpy() gadget to the attack stack to copy this object to an 128-byte aligned in-enclave address that is both writable and readable. Finding such memory area in the enclave is not difficult. In the SGX security model, the attacker already knows the region of the memory that is used by the enclave because all the memory allocation is handled by the untrusted operating system. Even though the page permission in the page table entry could not be matched with the permission on EPCM, the attacker can scan the entire address space to find the in-enclave address that can be used for a buffer. Second, we place multiple pop gadgets to change the value of the registers. We set rbx to be the in-enclave destination address and rcx to be both a readable and writable region in the enclave. At the same time, we set the rax register to 0x1, the index of the EGETKEY leaf function. Third, we place the ENCLU gadget to execute the EGETKEY leaf function. Finally, we put the memcpy() gadget again by chaining the pop gadgets to set rdi to a writable untrusted memory address and rsi to the address of the generated key in the enclave, which is the value of rcx on the second step.

The chain of gadgets will first call memcpy() to copy the KEYREQUEST data from the untrusted space to the in-enclave memory, execute EGETKEY with the prepared KEYREQUEST as a parameter, and then call memcpy() again to copy the generated key from the enclave to the untrusted space. At the end of the chain, the attacker can extract the key at the untrusted memory address that is set on rdi at the final step of memcpy() chaining. Using the extracted key, the attacker can freely encrypt/decrypt the data as well as generate the MAC to seal the data at the untrusted space because SGX uses the standard encryption algorithm (e.g., AES-256-GCM), which can be replicated anywhere if the same encryption key is supplied.

5 The SGX Malware

In this section, we demonstrate how the Dark-ROP attack can be applied in the real world to completely disarm the security guarantees of SGX.

From the proof-of-concept attack, the attacker can obtain the ability to call any leaf functions of SGX within the enclave to extract the secret data and inject data into the (trusted) enclave space. In addition to calling leaf functions to invoke the security-critical functions of SGX, we present techniques to implement the SGX malware, which can perform the man-in-the-middle (MiTM) attack to mimic the real enclave program for running security-critical operations within the enclave and to freely run attackers' code outside the enclave without any restrictions.

To achieve full control of the enclave, we construct the SGX malware as follows: 1) By using the `memcpy()` gadget, the attacker can extract any secret data in the enclave, including the program binary and data. Additionally, the attacker runs the extracted program binary outside the enclave to replicate the enclave execution. Moreover, the attacker can inject any arbitrary code to this extracted binary because it runs outside the enclave, which is fully controllable by the attacker. 2) The attacker is able to launch the security-critical operations of SGX that must be run in the enclave at any time. This can be done by launching the Dark-ROP attack to call the target leaf function with arbitrary register values. 3) The remote party must not know that the enclave is under attack, even with the remote attestation feature provided by SGX. This can be achieved by hijacking remote attestation by calling the `EREPORT` leaf function and constructing the correct measurement data outside the enclave.

In the following, we illustrate how we construct the SGX malware with preserving such requirements so that the SGX malware can run at the attacker's discretion while bypassing attack detection using the remote attestation.

Extracting the hidden binary/data from the enclave. The Dark-ROP attack allows the attacker to call the `memcpy()` function with arbitrary source and destination addresses (i.e., arbitrary read/write functionality). By utilizing this, the attacker can set the source address to be the start address of the binary section of the enclave, the destination to be untrusted memory region, and the size to be the entire mapped space for the enclave. Then, an execution of the `memcpy()` gadget will copy the hidden content of the binary from the enclave to the untrusted area. After obtaining the binary by dumping the area, the attacker can analyze the dump and run it to mimic the real enclave program. Moreover, because this binary does not run in the protected space, the attacker can freely inject the code to alter the program for his/her own purpose.

Using a similar method, by setting the source address to be the address of the secret data in the enclave, the attacker can extract them to process them outside the enclave without being protected by SGX.

Man-in-the-Middle ROP for launching the leaf functions. While running extracted binary at the untrusted space can mimic the execution of the regular instructions, however, the leaf functions of SGX must be run inside the enclave. Thus, when the extracted binary requires calling the leaf functions, the SGX malware invokes the function by launching the Dark-ROP attack against the real enclave.

To this end, we construct the SGX malware as a Man-in-the-Middle (MitM) architecture. In particular, the general mechanism for calling the leaf function in the enclave by exploiting the ROP attack works as follows. The SGX malware first injects required data for the target leaf function into the enclave using the `memcpy()` gadget. Next, the SGX malware loads the required parameters of the leaf function at the general purpose registers by using pop gadgets, and then jumps into `ENCLU` to call the leaf function. Finally, the malware copies the generated data by the leaf function from the enclave to the untrusted memory.

After this process, the SGX malware can continue to execute the code in the extracted binary by supplying the (extracted) return values of the leaf function (e.g., a derived encryption key for `EGETKEY`) to the current (untrusted) execution. This shows that the attacker has full control over the binary because the untrusted execution can run the regular instructions as well as the leaf functions whenever they are required.

Bypassing remote attestation. The last attack target of the SGX malware is to bypass remote attestation while running the binary at the untrusted area. Since the attestation requires generating the report in the enclave, primarily, we call the `EREPORT` leaf function by the Dark-ROP attack to generate the measurement report, and we emulate the entire process of the remote attestation in the binary outside the enclave to reply the correct measurement report to the remote server.

Before describing the emulation step, we present the background on how remote attestation typically works, as in Intel SGX SDK.

Remote attestation in Intel SGX SDK. The purpose of remote attestation is to ensure the correct settings and running of the enclave before conducting secret operations such as provisioning secrets and establishing a secure communication channel with the enclave in the remote machine.

The Intel SGX SDK uses the protocol in Figure 6 for

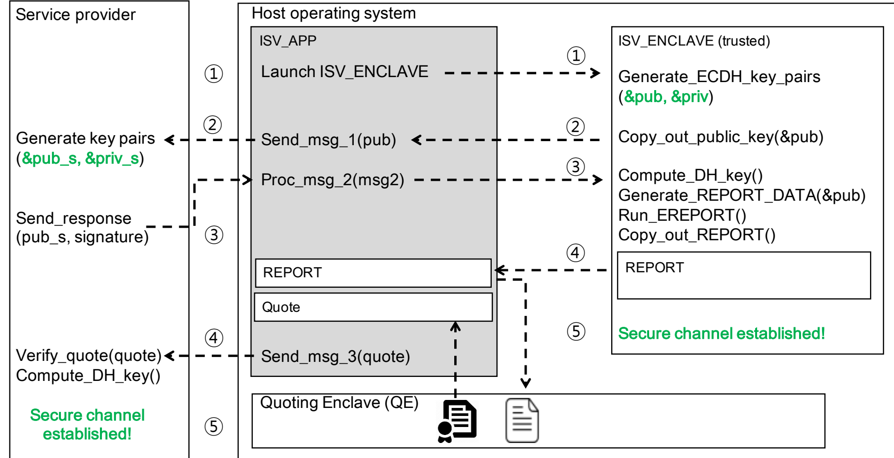

Figure 6: The (simplified) remote attestation protocol of SGX.

the remote attestation of the enclave and establishing a secure communication channel between the remote server and the enclave. First, (1) the untrusted part of the application deployed by an Independent Software Vendor (ISV, i.e., software distributor), called the *untrusted* program `isv_app`, launches the enclave program (we call this *trusted* program `isv_enclave`). On launching `isv_enclave`, `isv_app` requests the generation of Elliptic-Curve Diffie-Hellman (ECDH) public/private key pair to the enclave. The ECDH key pair will be used for sharing secret with the remote server. Then, the `isv_enclave` generates the key pair, securely stores the private key in the enclave memory and returns the public key to `isv_app`. This public key will be sent to the remote server for later use of sharing the secret for establishing a secure communication channel.

Second, on receiving the "hello" message from `isv_enclave`, (2) the remote server generates its own ECDH key pair that the server will use.

Third, (3) the server sends a quote request to the `isv_app`, to verify if the public key that the server received is from `isv_enclave`. Also, the server sends back the public key (of the remote server) to `isv_enclave`. To process the request, `isv_app` will invoke the function named `Compute_DH_Key` in `isv_enclave` to generate the shared secret and the measurement report (we refer this as `REPORT`). It contains the ECDH public key that `isv_enclave` uses as one of the parameters to bind the public key with the `REPORT`. Inside the enclave, `isv_enclave` uses the `EREPORT` leaf function to generate `REPORT`. On calling the leaf function, `isv_enclave` sets the `REPORTDATA`, an object that passed as an argument to the `EREPORT` leaf function, to bind the generated ECDH public key to the `REPORT`. After `isv_enclave` generates the `REPORT`, the untrusted `isv_app` delivers this to a *Quoting Enclave*(QE), a new enclave (trusted) for verifying the `REPORT` and then signs it with Intel EPID securely. As a result, the `REPORT` generated by `isv_enclave` contains the information for the ECDH public key that the enclave uses, and this information is signed by the QE.

Fourth, (4) the signed `REPORT` will be delivered to the remote server. The remote server can ensure that the `isv_enclave` runs correctly at the client side and then use the ECDH public key received at step (1) if the signed `REPORT` is verified correctly.

Finally, the server run `Compute_DH_Key` to generate the shared secret. (5) the remote server and `isv_enclave` can communicate securely because they securely shared the secret through the ECDH key exchange (with mutual authentication).

Controlling the REPORT generation. To defeat the remote attestation, and finally defeat the secure communication channel between the remote server and `isv_enclave`, in the SGX malware, we aim to generate the `REPORT` from `isv_enclave` with an arbitrary ECDH public key. For this, we especially focus on part (3), how `isv_enclave` binds the generated ECDH public key with the `REPORT` on calling the `EREPORT` leaf function.

The Dark-ROP attack allows the SGX malware to have the power of invoking the `EREPORT` leaf function with any parameters. Thus, we can alter the parameter to generate the `REPORT` that contains the ECDH public key that we chose, instead of the key that is generated by `isv_enclave`. On generating the `REPORT`, we prepare a `REPORTDATA` at the untrusted space using the chosen ECDH public key, and then chain the ROP gadgets to copy the `REPORTDATA` to the enclave space. Note that the `EREPORT` requires its parameters to be located in the enclave space. After copying the `REPORTDATA`, we call the `EREPORT` leaf function with copied data to generate the `REPORT` inside the `isv_enclave`. After this, we copy the generated `REPORT` from the `isv_enclave` to `isv_app` and delivers the `REPORT`

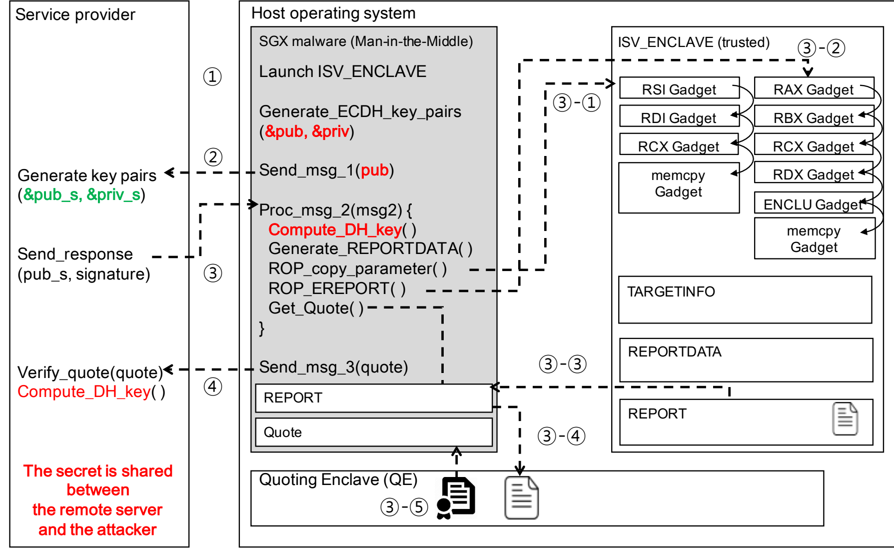

Figure 7: The Man-in-the-middle (MitM) attack of the SGX malware for hijacking the remote attestation in SGX.

to the QE to sign it.

As a result, at the untrusted space, the attacker can retrieve the `REPORT` that contains the ECDH parameter of his/her own choice, and the `REPORT` is signed correctly.

Hijacking remote attestation. The full steps of hijacking the remote attestation of an enclave are as follows (see Figure 7).

First, (1) instead of `isv_enclave`, the SGX malware generates an ECDH public/private key pair and own the private key. (2) the SGX malware sends the generated public key to the remote server.

Then, (3) on receiving the quote request from the server, the SGX malware calculates the shared secret corresponding to the parameters received by the remote server. Also, the SGX malware prepares `TARGETINFO` and `REPORTDATA` at `isv_app`. The `TARGETINFO` contains the information of the QE that enables the QE to cryptographically verify and sign the generated `REPORT`. The `REPORTDATA` is generated with the chosen public key as a key parameter to run `EREPORT` in the `isv_enclave`. After that, SGX malware launches the Dark-ROP attack (3-1, 3-2 and 3-3) to copy prepared parameters (`TARGETINFO` and `REPORTDATA`) from the untrusted app to the enclave and generate `REPORT` with the ECDH public key that the SGX malware generated at the first step. Moreover (3-4), the generated report will be copied out to the SGX malware from the `isv_enclave`, and the SGX malware sends the generated `REPORT` to the Quoting Enclave to sign this with the correct key. Because the `REPORT` is generated by the enclave correctly, the QE will sign this and return it to the attacker.

Finally, (4) the SGX malware sends this signed `REPORT` to the remote server. Now, the remote server shares the secret; however, it is not shared with the `isv_enclave`, but with the SGX malware so that the secure communication channel is hijacked by the SGX malware. Note that the remote server cannot detect the hijacking because all parameters and the signature are correct and verified.

6 Implementation

We implemented both the proof-of-concept attack and the SGX malware in the real SGX hardware. For the hardware setup, we use the Intel Core i7-6700 Skylake processor, which supports the first and only available specification of SGX, SGXv1. For the software, we run the attack on Ubuntu 14.04 LTS, running Linux kernel 4.4.0. Additionally, we use the standard Intel SGX SDK and compiler (`gcc-5`) to compile the code for the enclave for both attacks.

To launch the Dark-ROP attack on the real SGX hardware, we use the `RemoteAttestation` example binary in the Intel SGX SDK, which is a minimal program that only runs the remote attestation protocol, with slight modification, to inject an entry point that has a buffer overflow vulnerability, as mentioned in Figure 2.

Because the example is a very minimal one, we believe that if the Dark-ROP attack is successful against the `RemoteAttestation` example, then any other enclave programs that utilizes the remote attestation are exploitable by Dark-ROP if the program has memory corruption bugs.

Finding gadgets from standard SGX libraries. First, we search for gadgets from the example binary. To show the generality of finding gadgets, we find gadgets from the standard SGX libraries that are essential to run enclave

Table 1: Information for the length of ROP gadget chains for launching functions that breach the security of SGX.

Length of gadget chains (byte)			
memcpy	LEAF FUNCTION	EGETKEY	EREPORT
80	88	248	248

programs such as the library for controlling the enclave (*libsgx_trts.a*), the library that handles remote attestation protocol (*libsgx_tkey_exchange.a*), and the standard C library for SGX (*libsgx_tstdc.a*) because these libraries will be linked regardless of the program logic.

From the example binary, we found that four gadgets are enough to fulfill the gadget requirement described in §3 to launch the Dark-ROP attack against the `RemoteAttestation` example. Table 2 lists these four gadgets found in the example binary.

Constructing ROP chains for Dark-ROP. By chaining these gadgets, we construct ROP chains for calling the `memcpy()` function, and the `EREPORT` and `EGETKEY` leaf functions. To call the `memcpy()` function, we chained the four gadgets as follows. To set the registers for calling the memcpy function, we chained three gadgets, `pop rsi; pop r15; ret` and `pop rdi; ret` to set the destination and source address of memory copy, and `pop rdx; pop rcx; pop rbx; ret` to set the length of the data to be copied. As a result, we constructed an ROP chain for calling the `memcpy()` function. The total size of the gadget chain was 80 bytes, as shown in Table 1. To call the `EGETKEY` leaf function, we should call the `memcpy()` function to copy the `KEYREQUEST` structure first, set the register arguments for `EGETKEY`, and then call the `memcpy()` function again to move the generated key out to the untrusted area. By chaining two `memcpy()` gadgets and the leaf function gadgets, calling `EGETKEY` requires 248 bytes for gadget chaining. Similar to above, calling the `EREPORT` also requires 248 bytes for gadget chaining. Because the size of the chain is small enough (248 bytes as max) to fit into the overflowed stack (or heap area), we believe that the attack will work well in most cases.

7 Mitigation

We expect the adoption of traditional defense mechanisms in SGX to possibly mitigate Dark-ROP. However, since there are discrepancies between the normal execution environment and SGX, the specific features of SGX, which facilitate the attack in some aspects, need to be considered in the implementation of those defenses.

Gadget elimination. As shown in [28], the useful gadget that can be exploited to launch Dark-ROP can be eliminated before the enclave is deployed. For instance, we can transform the enclave code in a way to ensure that it does not contain any non-intended *ret* instructions. Moreover, we need to consider how to manage the non-removable SGX specific gadgets that contain the ENCLU instruction. For the transition between the host program and the enclave, at least one ENCLU instruction (for EEXIT leaf function) is required for the enclave, the requirement that makes it hard to completely remove the gadgets. We expect that implanting the register validation logic right after the ENCLU instruction could be a possible solution. Specifically, we can ensure that the ENCLU instruction in a certain location is tightly coupled with one of the pre-defined leaf functions. Besides, the way to remove the gadget that performs as a *memcpy* function, which is generally required to operate (un)marshalling the parameters between the host program and the enclave, should also be considered.

Control flow integrity. Deploying the CFI in the enclave also needs to consider the SGX-specific features. For instance, as shown in Figure 3, an attacker can arbitrarily incur the AEX to freeze the status (context) in the enclave. Then, he can create another thread to leak or manipulate the context (e.g., the saved general-purpose registers in the stack) of the trapped thread. Therefore, if the CFI implementation uses one of the general registers to point to the reference table that defines allowed target blocks, it can be easily bypassed by the attacker's manipulating the context saved in the stack of the trapped thread.

Fine-grained ASLR. Research projects that adopt fine-grained ASLR on enclave programs such as SGX-Shield [31] would possibly mitigate Dark-ROP. However, it should also accompany with enclave developer's careful configuration since Dark-ROP can still be effective by exploiting the number of state save area (NSSA) field that defines the number of allowed re-entrances to the enclave without reconstructing it. More specifically, SGX allows multiple synchronous entrances (EENTER) depending on the value configured in the NSSA field, even after the AEX happens (if ERESUME is executed instead of EENTER, the enclave crashes and thus the attacker needs to reconstruct the enclave). Therefore, if the value of the NSSA field is large enough, the attacker might be able to continuously reenter the enclave without reconstructing it, which enables the preservation of the previous memory layout. According to SGX specifications [20, 21], the value of NSSA can be up to a 4-byte integer, and we expect this to be enough to reliably locate all necessary gadgets.

8 Related work

In this section, we describe SGX-related prior works in the following respects: (1) application of SGX, (2) attacks

against SGX, (3) enclave confidentiality protection, and (4) comparison between BROP and Dark-ROP.

SGX application. Intel SGX has been utilized to secure various applications. Ryoan [17] ported Google NaCl in and SGX enclave to create a distributed sandbox that prevents sensitive data leakage. SCONE [3] leverages SGX to host a Docker container in the enclave, which specifically concerns the security enhancement and low overhead. Town Crier [39] isolates the crypto functions for the smart contract in the enclave. To prevent an Iago [10] attack, Haven [6] isolates the unmodified Windows application, library OS, and shielded module together in the enclave. Network services and protocols such as software-defined inter-domain routing are shown to possibly coordinate with SGX in Kim et al [25].

Attacks on SGX. Several research projects have explored potential attack surfaces on SGX. The controlled side-channel attack [33, 38] shows that the confidentiality of the enclave can be broken by deliberately introducing page faults. Asyncshock [37] presents how a synchronization bug inside the multi-threaded enclave can be exploited. Unfortunately, this work does not target the attacker who has full control over the enclave program. Instead, the work describes how the proposed attack can subvert the confidentiality and integrity of SGX.

Enclave confidentiality protection. As described in [18, 24, 30], an enclave binary can be distributed as a cipher text to preserve the confidentiality of the code and data deployed in the enclave. VC3 [30] shows a concrete implementation example that partitions the enclave code base as public (plaintext) and private (encrypted) and enables the public code to decrypt the private code. CONFIDENTIAL [34] provides a methodology that prevents the secret leakage from the enclave by enforcing the narrow interface between the user program and small library, and defining the formal model to verify the information release confinement. In addition, Moat [34, 35] tracks information flows by using static analysis to preserve the enclave confidentiality. Our work shows that, even with the protection of enclave confidentiality, Dark-ROP can be successfully deployed by exploiting a certain SGX hardware design and its functionality.

Revisiting BROP for Dark-ROP. Blind ROP [7] is an attack technique that can locate and verify the required gadgets in the restrictive environment where neither the target binaries nor the source code is known to the attacker. To this end, it depends on two primary gadgets, which are called the `trap gadget` and the `stop gadget`, both of which incur the program to be crashed or stopped when they are consumed (popped) as part of the input payload that is crafted by an attacker to specify the potential (and currently probed) gadget.

On the contrary, the Dark-ROP attack takes an orthogonal approach, which exploits the three oracles that allow the attacker to obtain hints of the gadgets by the features of SGX (i.e., page fault, `EEXIT`, and the memory) to identify required gadgets from a completely hidden environment. Additionally, the Dark-ROP attack can be applied to any application that runs in an enclave, whereas original Blind ROP is only applicable to server-like applications.

9 Conclusion

Dark-ROP is the first practical ROP attack on real SGX hardware that exploits a memory-corruption vulnerability and demonstrates how the security perimeters guaranteed by SGX can be disarmed. Despite the vulnerability in the enclave, realizing the attack is not straightforward since we assume the most restrictive environment where all the available security measures based on Intel SGX SDK and recent SGX-related studies are deployed in the enclave; thus, the code reuse attack and reverse engineering on the enclave binary may not be conducted. To overcome this challenge and accomplish the attack, Dark-ROP proposes the novel attack mechanism, which can blindly locate the required ROP gadgets by exploiting SGX-specific features such as enclave page fault and its handling by an asynchronous exception handler, ENCLU introduced as part of new SGX instructions, and shared memory for the communication between the enclave and the non-enclave part of program. Finally, as a consequence of Dark-ROP, we show that the attacker can successfully exfiltrate the secret from the enclave, bypass the SGX attestation, and break the data-sealing properties. We hope that our work can encourage the community to explore the SGX characteristic-aware defense mechanisms as well as an efficient way to reduce the TCB in the enclave.

10 Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2017R1A2B3006360), ICT R&D programs MSIP/IITP [R-20150223-000167] and MSIP/IITP [R0190-15-2010]. Jaehyuk Lee was partially supported by internship at Microsoft Research. This research was also partially supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851, ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, and DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy* (2013), vol. 13.

[2] ARM. Building a secure system ising trustzone technology, Dec. 2008. PRD29-GENC-009492C.

[3] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., ET AL. Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association.

[4] BARNETT, R. Ghost gethostbyname () heap overflow in glibc (cve-2015-0235), january 2015.

[5] BAUMAN, E., AND LIN, Z. A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution* (2016), ACM, p. 4.

[6] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014), pp. 267–283.

[7] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 227–242.

[8] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), ACM, pp. 30–40.

[9] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 27–38.

[10] CHECKOWAY, S., AND SHACHAM, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 253–264.

[11] CHHABRA, S., SAVAGAONKAR, U., LONG, M., BORRAYO, E., TRIVEDI, A., AND ORNELAS, C. Memory encryption engine integration, June 23 2016. US Patent App. 14/581,928.

[12] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., ET AL. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 475–488.

[13] GOOGLE. glibc getaddrinfo() stack-based buffer overflow (cve-2015-7547), february 2016.

[14] GREENE, J. Intel trusted execution technology. *Intel Technology White Paper* (2012).

[15] GUERON, S. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. http://eprint.iacr.org/.

[16] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (Tel-Aviv, Israel, 2013), pp. 1–8.

[17] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association.

[18] INTEL. SGX Tutorial, ISCA 2015. http://sgxisca.weebly.com/, June 2015.

[19] INTEL CORPORATION. Intel Software Guard Extensions Programming Reference (rev1), Sept. 2013. 329298-001US.

[20] INTEL CORPORATION. Intel Software Guard Extensions Programming Reference (rev2), Oct. 2014. 329298-002US.

[21] INTEL CORPORATION. Intel SGX Enclave Writers Guide (rev1.02), 2015. https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf.

[22] INTEL CORPORATION. Intel SGX SDK for Windows* User Guide (rev1.1.1), 2016. https://software.intel.com/sites/default/files/managed/d5/e7/Intel-SGX-SDK-Users-Guide-for-Windows-OS.pdf.

[23] JOHNSON, S., SAVAGAONKAR, U., SCARLATA, V., MCKEEN, F., AND ROZAS, C. Technique for supporting multiple secure enclaves, June 21 2012. US Patent App. 12/972,406.

[24] JP AUMASSON, L. M. Sgx secure enclaves in practice:security and crypto review, 2016. [Online; accessed 16-August-2016].

[25] KIM, S., SHIN, Y., HA, J., KIM, T., AND HAN, D. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)* (Philadelphia, PA, Nov. 2015).

[26] 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 (to appear). In *Proceedings of the 26th USENIX Security Symposium (Security)* (Vancouver, Canada, Aug. 2017).

[27] OHRIMENKO, O., SCHUSTER, F., FOURNET, C., MEHTA, A., NOWOZIN, S., VASWANI, K., AND COSTA, M. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium* (2016), pp. 619–636.

[28] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 601–615.

[29] RUTKOWSKA, J. Thoughts on Intel's upcoming Software Guard Extensions (Part 2), Sept. 2013. http://theinvisiblethings.blogspot.com/2013/09/thoughts-on-intels-upcoming-software.html.

[30] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2015).

[31] 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 (to appear). In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2017).

[32] 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.

[33] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), ACM, pp. 317–328.

[34] SINHA, R., COSTA, M., LAL, A., LOPES, N., SESHIA, S., RAJAMANI, S., AND VASWANI, K. A design and verification methodology for secure isolated regions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016), ACM.

[35] SINHA, R., RAJAMANI, S., SESHIA, S., AND VASWANI, K. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1169–1184.

[36] TSAI, C.-C., ARORA, K. S., BANDI, N., JAIN, B., JANNEN, W., JOHN, J., KALODNER, H. A., KULKARNI, V., OLIVEIRA, D., AND PORTER, D. E. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 9.

[37] 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.

[38] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 640–656.

[39] ZHANG, F., CECCHETTI, E., CROMAN, K., JUELS, A., AND SHI, E. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 270–282.

A Dark-ROP gadgets

Table 2: Gadgets used for launching the Dark-ROP attack against the `RemoteAttestation` example code in the Intel SGX SDK. We note that we found all gadgets from the standard library files, which are usually linked to the enclave program. First, the entire objects in the *libsgx_trts.a* must be linked to the enclave binary because the library contains the code for controlling the enclave and communication between the untrusted app and the enclave, which are essential to function the enclave. Finally, we found `memcpy()` gadget from the standard c library for SGX (*libsgx_tstdc.a*).

Gadget	Description	From

ENCLU Gadget
`do_ereport:`
`enclu`
`pop rdx`
`pop rcx`
`pop rbx`
`ret`
	
	The ENCLU gadget for invoking the leaf functions.
	The gadget is followed by three pop gadgets
	so that the attacker can set the
	rdx, rcx, and rbx registers to arbitrary values,
	which will be used for passing arguments to the leaf functions.
		
		libsgx_trts.a

`sgx_register_exception_handler:`
`mov rax, rbx`
`pop rbx`
`pop rbp`
`pop r12`
`ret`
	
	A gadget for manipulating the rax register.
	Since the attacker can control the rbx register with the gadget above,
	the attacker can set rax to be an arbitrary value.
	This is for setting the index of the leaf function for
	the ENCLU instruction.
		
		libsgx_trts.a

`relocate_enclave:`
`pop rsi`
`pop r15`
`ret`
`pop rdi`
`ret`
	
	A gadget for manipulating rsi and rdi registers
	to set arguments for invoking memcpy
	and the other library functions.
		libsgx_trts.a

Memcpy Gadget `memcpy:`	 A gadget for copying enclave code and data to untrusted memory, and for copying in the reverse direction *vice versa*.	 libsgx_tstdc.a

Table 3: Gadgets used to launch Dark-ROP in Windows 64bit.

Gadget	Description	From

GPR Modification Gadget
`__intel_cpu_indicator_init:`
`pop r15`
`pop r14`
`pop r13`
`pop r12`
`pop r9`
`pop r8`
`pop rbp`
`pop rsi`
`pop rdi`
`pop rbx`
`pop rcx`
`pop rdx`
`pop rax`
`ret`
	
	This gadget is used for manipulating GPRs
	All Pop-gadgets required for launch Dark-ROP
	can be located in this one function
	
	This function is introduced by *libirc.a*
	which is an Intel support library for CPU dispatch
	Note that this function is also available at
	libsgx_tstdc.a in Linux 64bit.
		
		sgx_tstdc.lib

ENCLU Gadget
`do_ereport:`
`enclu`
`pop rax`
`ret`
		
		sgx_trts.lib