COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX

Mustakimur Rahman Khandaker
mrk15e@my.fsu.edu
Florida State University

Yueqiang Cheng✉
chengyueqiang@baidu.com
Baidu Security

Zhi Wang
zwang@cs.fsu.edu
Florida State University

Tao Wei
lenx@baidu.com
Baidu Security

Abstract

Intel SGX is a hardware-based trusted execution environment (TEE), which enables an application to compute on confidential data in a secure enclave. SGX assumes a powerful threat model, in which only the CPU itself is trusted; anything else is untrusted, including the memory, firmware, system software, etc. An enclave interacts with its host application through an exposed, enclave-specific, (usually) bi-directional interface. This interface is the main attack surface of the enclave. The attacker can invoke the interface in any order and inputs. It is thus imperative to secure it through careful design and defensive programming.

In this work, we systematically analyze the attack models against the enclave untrusted interfaces and summarized them into the COIN attacks – Concurrent, Order, Inputs, and Nested. Together, these four models allow the attacker to invoke the enclave interface in any order with arbitrary inputs, including from multiple threads. We then build an extensible framework to test an enclave in the presence of COIN attacks with instruction emulation and concolic execution. We evaluated ten popular open-source SGX projects using eight vulnerability detection policies that cover information leaks, control-flow hijackings, and memory vulnerabilities. We found 52 vulnerabilities. In one case, we discovered an information leak that could reliably dump the entire enclave memory by manipulating the inputs. Our evaluation highlights the necessity of extensively testing an enclave before its deployment.

CCS Concepts. • **Security and privacy** → **Trust frameworks**; **Vulnerability scanners**.

Keywords. Intel SGX; enclave; vulnerability detection

ACM Reference Format:
Mustakimur Rahman Khandaker, Yueqiang Cheng✉, Zhi Wang, and Tao Wei. 2020. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378486

1 Introduction

Intel Software Guard Extensions (SGX) introduces a set of ISA extensions to enable a trusted execution environment, called enclave, for security-sensitive computations in user-space processes. The enclave, for the most part, is strictly protected from other parts of the system through memory access control, memory encryption and integrity-check, attestation, etc [9, 27]. By design, Intel SGX assumes a powerful attack model where only the CPU is trusted; everything else could be compromised or malicious, including the firmware (i.e., BIOS), the VMM (virtual machine monitor), and the OS (operating system) kernel. Because of this, Intel SGX becomes increasingly popular for security-sensitive applications, such as secure remote computation, authentication, and digital rights management. For example, enclaveDB is an SGX-based database engine that can guarantee confidentiality, integrity, and freshness for data and queries [31]; CYCLOSA is a secure, scalable, and accurate private web search system [30]; and IRON is a secure functional encryption system [12]. In addition, many popular libraries, such as mbed TLS and Tor, have been ported to run in the SGX enclave [11, 17, 43], and most major cloud providers also provide SGX-based remote confidential computing services [16, 29].

Recent research on side-channel attacks has brought the security of SGX into question [7, 18, 38, 41]. For example, SgxPectre, the SGX-variant of Spectre [7], can read the secret keys of SGX by exploiting the race condition between injected, speculatively executed memory references and the latency of branch resolution. There is relatively little attention paid to the (in)security of the SGX interface [24, 42]. In particular, AsyncShock proposes a method to exploit known synchronization bugs in the SGX by manipulating the thread scheduler [42]; while Lee et al. exploit uninitialized structure padding to leak enclave data [24].

Securing the enclave interface demands more than simple sanitization of inputs: an enclave is an isolated execution environment in the user-space. It, therefore, relies on the *untrusted* supporting system to interact with the external environment. Such interaction is bi-directional. Specifically, an SGX application (i.e., the application whose address space hosts the enclave) can call an enclave function using an ECALL, an RPC-like mechanism. Vice versa, the enclave can use an OCALL to temporarily exit the enclave and call an untrusted function to access OS services. For example, an SGX-based SQLite engine can use OCALLs to open an (encrypted) database file and read its content to serve a query from the host application (ECALL). This design makes the enclave rather challenging to secure because the enclave interface can be called in arbitrary order and with any inputs, yet the enclave has little control over it besides input sanitization.

In this paper, we first systematically study the attack surface of the enclave interface, and then propose an extensible framework to automatically analyze enclave libraries for common vulnerabilities. Specifically, we categorize the enclave attack surface into four models: input manipulation, call permutation, concurrent ECALLs, and nested calls. The first model consists of attacks that manipulate the inputs to enclave calls to check whether sufficient sensitization of the inputs has been applied; The second one consists of attacks that make arbitrary calls to the ECALL interface from the untrusted code. Libraries often assume their API functions are called in certain orders. For example, a database engine (reasonably) assumes a database is opened before any queries are executed. However, SGX itself does not impose any restrictions on the invocation of ECALLS. This model thus check the enclave code when these assumptions do not hold; The third model tests the enclave code for concurrency bugs by invoking the ECALL interface from multiple threads; Finally, the last model checks the enclave code when ECALLs are nested, i.e., a second ECALL is made while the enclave is executing an OCALL in response to the first ECALL. These four models are collectively called the **COIN** attacks, which stands for **C**oncurrency, **O**rder, **I**nputs, and **N**ested, each corresponding to one attack model. In this paper, we focus on the first three models and leave the nested model as future work. We include the nested mode for completeness.

Given these attack methods, we designed an extensible framework to systematically study the (in)security of enclave libraries. The framework implements these attack models in the front-end using instruction emulation and symbolic execution. Specific policies to detect vulnerabilities can be plugged into the framework to provide more accurate identification and classification of vulnerabilities. In our prototype, we implemented eight policies that cover information leaks, control-flow hijackings, and memory vulnerabilities (e.g., use-after-frees, heap/stack overflows). Other types of vulnerabilities can be readily added to the framework with additional policies. We applied the prototype to evaluate ten popular open-source SGX projects on the GitHub, including Intel SSL, SQLite3, mbedTLS, etc. We found 52 vulnerabilities in them, including double-frees, use-after-frees, heap and stack overflows, null-pointer de-references, heap and stack memory leaks, etc. In one case, we can *reliably dump the entire memory of the mbedTLS-SGX enclave* by manipulating the untrusted inputs. Each attack model contributed to the discovery of these vulnerabilities, with input manipulation being the most effective (42 vulnerabilities). We have reported all these vulnerabilities to the affected projects and provide bug fixes for some. These results demonstrate the need for automated vulnerability detection tools that are specifically designed to handle the complexity of the SGX programming and trust model.

This paper makes the following contributions:

- We introduced the COIN attacks, a systematic analysis of the SGX interface attack surface. COIN attacks consist of concurrency, order, input, and nested call attacks.
- We proposed the design of an extensible framework targeting the COIN attacks and implemented the design with eight detection policies that cover many common vulnerabilities.
- We evaluated our system with ten open-source SGX projects and found (and reported) 52 vulnerabilities in them, including a whole SGX memory leak.

The rest of the paper is organized as follows. We first describe the SGX programming model in Section 2. We then explain the COIN attacks in Section 3. Next, Section 4 presents the system design and Section 5 evaluates the system. Later, we discuss the related works in Section 6. Finally, we conclude the paper in Section 7.

2 Background

In this section, we introduce background information about the SGX programming model to help understand the COIN attack models.

2.1 SGX Programming Model

An SGX enclave is an isolated execution unit in the user address space of its host process. It relies on the host process as a proxy to interact with the environment, for example, to access database files. The way this works is similar to RPC (remote procedure call): the host process calls an enclave function by marshaling the parameters before passing them to the enclave; the enclave then unpacks the parameters, executes the function, and returns the (marshaled) results. This is named an ECALL. Vice versa, the enclave can make an OCALL to execute an untrusted function in the host process. ECALLs and OCALLs constitute the enclave's interface, i.e., its software attack surface.

```
enclave {
  include "../ocall_types.h"
  from "sgx_tstdc.edl" import *;

  trusted {
    public void ecall_opendb([in, string] const char *dbname);
    public void ecall_execute_sql([in, string] const char *sql);
    public void ecall_closedb(void);
  };

  untrusted {
    int ocall_stat([in, string] const char *path,
            [in, out, size=size] struct stat *buf, size_t size);
    int ocall_ftruncate(int fd, off_t length);
    int ocall_getpid(void);
    char* ocall_getenv([in, string] const char *name);
  };
};
```

Figure 1. A part of the EDL file from SGX-SQLite

Intel provides an SDK to automate the creation of the enclave interface, defined by a domain language called EDL (enclave definition language). A tool in the SDK called Edger8r parses an EDL file and creates stubs for the host program and the enclave. A part of the EDL file from SGX-SQLite is shown in Fig. 1. It defines three ECALLs (e.g., opendb and execute_sql) and four OCALLs (e.g., getpid). The declaration of parameters are annotated with keywords in, out, string, and size. in and out denote the data flow direction. For example, the buf parameter of ocall_stat is an in and out parameter, i.e., the ocall will read from and write to the associated buffer. If a parameter is a pointer, the EDL file could also specify its size with either the size or string (a null-terminated buffer) keywords. For instance, the size of buf in ocall_stat is specified by the function's third parameter, also named as size. Edger8r parses the EDL file and generates two stub files, one for the host program and one for the enclave. Both files contain a definition for each ECALL and OCALL, similar to RPC: one stub marshals the parameters; the other unpacks the parameters, executes the function, and returns the results. Edger8r automatically generates code to sanitize parameters according to the annotation. For example, it will copy only size bytes from and into buf for ocall_stat. This addresses some basic security issues but, as we will show, is grossly inadequate. Our system also parses EDL files to obtain the basic semantics of the ECALL/OCALL parameters.

The execution of the enclave starts with an ECALL. During that, the enclave might make a few OCALLs in order to serve the ECALL. For example, a contact manager can call execute_sql to securely query its SGX-based database; to process the query, the enclave makes a few OCALLs to read the encrypted database file, which is normally encrypted and stored in the (untrusted) file system. Therefore, ECALLs and OCALLs constitute the software attack surface of the enclave.

3 COIN Attacks

We systematically analyzed the software attack surface of an enclave and summarized it with COIN attacks. COIN stands for concurrency, order, inputs, and nested. Note that we assume the attacker has full control over the host process and the underlying OS, but he can only interact with the enclave through its defined ECALL/OCALL interface. This is consistent with the official SGX threat model.

Concurrent calls (concurrency): the attacker tries to call enclave functions concurrently from multiple untrusted threads. An enclave cannot assume that it will be called only from a single thread at a time, or external synchronization primitives will enforce proper lock semantics (instead, it should use locks such as in-enclave spinlocks).

Call permutation (order): the attacker calls enclave functions (i.e., ECALLs) in arbitrary orders. This model works because an enclave often has an implicit assumption about the order of various ECALLs. For example, SGX-SQLite exposes three ECALLs, opendb, execute_sql, and closedb. Apparently, opendb must be called before the other two. Though, an enclave cannot assume any order of invocation of its ECALLs. This attack could be particularly troublesome for SGX libraries converted from existing libraries, which almost always assume the correct order of invocation is the response of their users.

Input manipulation (inputs): the enclave accepts inputs from the parameters of ECALLs and the return-values of OCALLs. Even though Intel SGX SDK adds simple sanitization to the parameters, such sanitization is often insufficient because it lacks the knowledge of how the enclave uses its inputs. The effectiveness of this attack model is demonstrated by the heap and stack overflows we found despite the main focus of the added sanitization is to prevent buffer overflows.

Nested calls (nested): the attacker makes new ECALLs in an (untrusted) OCALL handler, which has been initiated by an earlier ECALL (thus the name of nested calls). Use SGX-SQLite as an example again, the attacker first calls opendb and execute_sql to query a database. The enclave makes an OCALL in execute_sql, to read the database file. In the (untrusted) handler of this OCALL, the attacker can make any ECALLs he wants, such as closedb. As mentioned before, we defer this attack model to future work.

These attack models are not mutually exclusive. They can be combined together to discover more vulnerabilities than the sum of the individual models.

The COIN attacks are closely related to the Iago attack [6], in which the untrusted OS kernel tries to explore the trusted user-space process by manipulating syscall returns. The Iago attack corresponds to input manipulation of OCALL returns in our system. As such, our system has more attack models with regard to SGX architecture.

4 System Design

In this section, we present the design of an extensible framework to check the security of SGX projects against (three of) the COIN attacks. The framework aims at discovering vulnerabilities before the production release of the enclave code. Consequently, we assume that the enclave source code is available.

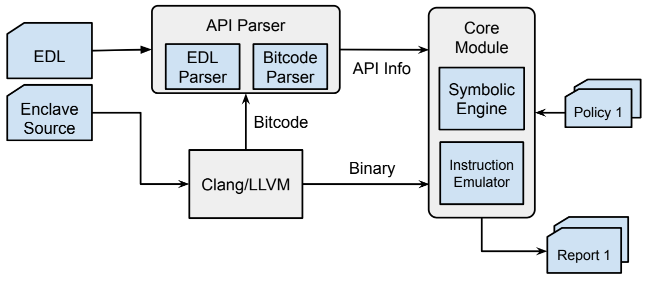

Figure 2. Overview of the enclave analysis framework

Fig. 2 illustrates the overall architecture of the system. Specifically, we compile the enclave source with the Clang compiler into LLVM bitcode. We then parse the EDL file and the bitcode to automatically extract the enclave interface definition. The interface definition and the enclave binary are passed to the core module for analysis. The core module is the key component of the system. It employees an instruction emulator (based on QEMU [4]) and a symbolic execution engine to systematically explore the enclave code. The core module implements the three attack models to drive the exploration of the enclave code. However, the attack models themselves cannot detect vulnerabilities. The core module instead relies on extensible policies to detect specific vulnerabilities. Our prototype provides eight policies that can detect common vulnerabilities such as heap/stack information leaks and use-after-frees. More policies can be readily integrated into the framework to detect more vulnerabilities. In the rest of this section, we describe the main components of our system in more detail. In this architecture, only the API parser requires the source code of the enclave. It is possible to redesign the API parser to take the enclave binary instead of the bitcode.

4.1 API Parser

The API parser analyzes the EDL file and the LLVM bitcode of the enclave to extract the basic information about the interface of the enclave. This information is passed to the core module's symbolic engine to map symbolic variables to their memory layouts.

The API parser has two components, the EDL parser and the bitcode parser. The former reads the EDL file and extract the information such as variable types, data flow directions, and variable of object size if it is a pointer. For example, the `dbname` parameter of the `ecall_opendb` function is an input-only string (Fig. 1); The latter parse the enclave bitcode to extract memory layout of structure variables, such as the `buf` parameter of `ocall_stat`, a variable of the `struct stat` type. This is feasible because the Clang/LLVM compiler encodes the memory layouts of structural variables in the bitcode. It is possible to obtain the same information from the binary or the source code alone. But this approach is more reliable.

4.2 Core Module

The core module is the key component of the framework. It consists of the controller, the instruction processor, the symbolic engine, and the OCALL hooks, as shown in Fig. 3. The controller manages the overall execution of the module; the instruction processor uses a symbolic engine to run instructions of the enclave in a single- or multi-threaded setting; the OCALL hooks provide an interface for the symbolic engine to manipulate OCALL returns, which are a part of the enclave inputs. The policy module has close interaction with all the components of the core module in order to control their execution or to detect vulnerabilities. The policy module can be extended with detectors for many types of vulnerabilities. We will describe the details of the policy module later. The main function of the core module is to implement the three COIN attack models: input manipulation, call permutation, and concurrent calls. Next, we describe how each attack model is implemented in detail.

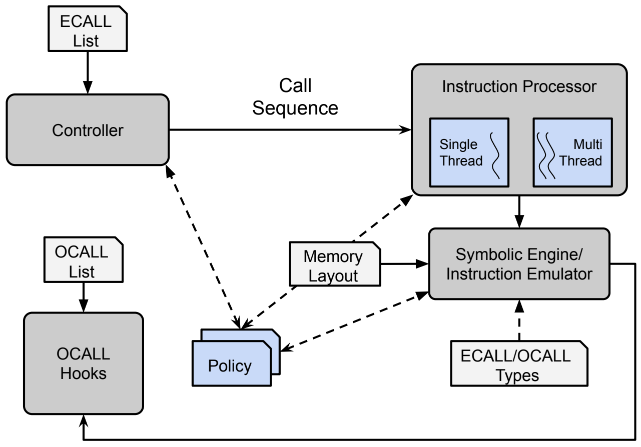

Figure 3. Core module architecture

Call permutation: an SGX library often assumes specific orders in which its exposed functions are called by the untrusted code. This model checks whether the library correctly handles functions called in arbitrary/unexpected orders. It can potentially discover memory management vulnerabilities such as uninitialized memory accesses, use-after-frees, double-frees, etc. This model is implemented by the core module's controller. It accepts a list of ECALLs supported by the enclave. The controller generates many random sequences of ECALLs and passes them to the instruction processor. The permutation of the calls randomizes the number and order of ECALLs. For example, it could call the `execute_sql` function of SGX-SQL without calling `opendb` first, or call `closedb` twice, etc. Each call sequence is executed with the same enclave instance, i.e., we create the (emulated) enclave before the first ECALL and destroy it after the last ECALL in order to keep the enclave states across the calls in the same sequence. A new enclave is created for every sequence from the call permutation list.

Call permutation is mixed with other two attack models for effectiveness: a call sequence is emulated by the instruction processor in both single-threaded and multi-threaded environments; and during the emulation, the symbolic engine tries to explore the enclave by symbolizing the inputs, i.e., ECALL parameters and OCALL returns.

Concurrent calls: each call sequence is executed in two modes: the single-threaded and multi-threaded model. In the former, each ECALL in the sequence is executed sequentially by a single thread; while in the latter, each ECALL is executed by a separate thread concurrently. All the threads for a call sequence share a single lock to ensure that each thread executes one instruction at a time. To control the concurrency, the controller maintains a thread to wait counter map. The wait counter can be set by the policy module to pause a thread from processing its current instruction until other threads have completed a specific number of instructions. This allows the policy module to control the concurrent execution of the threads and makes the analysis possible to reproduce.

Input manipulation: an enclave accepts two sets of inputs: ECALL parameters and OCALL returns, both under the attacker's control. Accordingly, an enclave should not trust its inputs. Even though the SGX SDK automatically turns the ECALL parameter annotation into sanitization checks, these checks only cover a few basic cases. In addition, it cannot provide any protection against OCALL returns (because there is no annotation for OCALL returns). As such, enclave developers need to apply additional checks for inputs. This attack model tests whether these checks are adequate or not. For better results, the framework employs a symbolic execution engine to extensively explore the enclave code. We first describe how inputs are symbolized.

Enclave inputs are symbolized into symbolic variables according to their types, constraints, and layouts (from the API parser). If an input is a pointer to a structure, we allocate 8 bytes of concrete memory for the pointer and target the pointer to a chunk of the enclave heap memory. Fields of the structure are then converted to symbols by the structure layout. We then symbolize all the rest scalar inputs (e.g., `int`). In most cases, inputs are passed to the enclave in registers. As registers are not uniquely named, they will cause conflicts during the execution. Therefore, we symbolize instead the memory, into which the first `mov` instruction moves data from the associated register. We build the enclave with the `-O0` flag to ensure that the function prologue always contains the expected `mov` instructions.

The symbolic engine is based on concolic execution, i.e., the enclave code is executed concretely while maintaining the relationship of (symbolic) variables. Therefore, the same call sequence will be executed many times, firstly with dummy seeds for symbolic variables (e.g., 0 for integers). The framework maintains a symbolic variable to seeds mapping to avoid re-execute the enclave with the same seeds. During execution, the engine collects path constraints in terms of symbolic variables. After the enclave is executed, the engine uses a constraint solver to solve path constraints in order to expand the part of the enclave code explored. Here, we also use the API information to discard seeds that violate it. The engine then executes the enclave again with the generated inputs. As mentioned earlier, the symbolic engine hooks into all the OCALL functions. When an OCALL function is to be executed, it can determine the OCALL's return and whether the OCALL function will be executed or not. As such, the core module can manipulate the inputs from both ECALLs and OCALLs.

4.3 Policy Module

The policy module has close interaction with the components of the core module. This allows custom policies to access the program/enclave states and control aspects of the instruction emulation. For example, a use-after-free policy can hook memory allocation and free functions of the enclave to keep track of the valid memory, and memory access instructions to check whether the accessed memory is valid or not. In addition, if the current call sequence is being executed in the multi-thread mode, the policy can temporarily pause the execution of the current thread after memory free functions in order to detect use-after-frees from another thread. If a policy detects a vulnerability, it produces a customized report. For example, the report for a use-after-free includes the invalid memory region, the most recent allocation and free sites of that memory, etc. Additionally, the core module stores recently emulated 500 instructions in a ring buffer to facilitate pattern-based vulnerability detection (e.g., information leaks) and offline debugging of a reported vulnerability.

The policy module can be extended with policies to detect specific vulnerabilities. In our prototype, we included eight policies to detect common vulnerabilities, such as heap/stack information leaks, use-after-frees, double-frees, stack/heap overflows, and null-pointer de-references. We will describe them in detail in the next section.

Table 1. List of selected SGX projects from GitHub

Project	Description	Enclave LoC	# of Bugs
mbedtls-SGX [11]	Crypto and SSL/TLS support for embedded systems.	59, 228	11
SGX-Tor [17]	Tor anonymity network.	316, 962	9
TaLoS [21]	Secure TLS termination.	183, 958	7
Bolos-enclave [22]	Trusted environment for blockchain applications.	8, 463	6
Intel-SGX-SSL [15]	SSL cryptographic library from Intel.	6, 508	5
SGX_SQLite3 [43]	Secure SQLite query.	118, 997	4
SGX-Migration [34]	Live migration VMs.	2, 829	3
SGX-Wallet [1]	Trusted password-wallet.	252	3
SGX-Reencrypt [19]	Symmetric reencryption.	1, 772	2
SGXCryptoFile [32]	Encrypting and decrypting HLS chunks.	157	2

5 Evaluation

In this section, we present the eight policies implemented in our prototype and evaluate their effectiveness and performance over ten open-source SGX projects from GitHub. Our prototype was implemented based on the QEMU instruction emulator and the Triton symbolic engine. Triton uses the z3 theorem prover as its constraint solver. The API parser is implemented based on the LLVM libtool. The policy module supports eight policies to detect heap/stack information leaks, ineffectual conditions, use-after-frees, double-frees, heap/stack overflows, and null pointer dereferences. We call vulnerabilities where a critical conditional check is under the attacker's control as an ineffectual condition. These policies do not cover all the cases of these vulnerabilities but are still very effective. The prototype currently works for the x86-64 Linux system, but it could be ported to other platforms (e.g., the Windows system). The main code-base has 4K+ lines of code with additional 1K+ lines of source code for the policies.

5.1 Effectiveness

In this subsection, we first describe the overall effectiveness of our system, and then the details of each policy.

5.1.1 Overview. We picked ten popular open-source SGX projects from GitHub as test subjects, as listed in Table 1. They can be categorized as either library wrappers (e.g., Intel-SGX-SSL) [1] or SGX based applications (e.g., SGX-Wallet). Their code sizes range from hundreds to about 320K lines of source code. Table 1 also lists the number of verified vulnerabilities we discovered from these projects. Table 2 further breaks down the vulnerabilities of each project by the policies. For example, we found from Intel-SGX-SSL, an OpenSSL wrapper developed by *Intel*, five vulnerabilities: three ineffectual conditions, one stack overflow, and one null-pointer-dereference. We note that these vulnerabilities are not a part of the original OpenSSL library. They instead were a part of the wrapper.

This demonstrates the difficulty in writing secure code for the SGX environment, particularly for wrappers of existing libraries: libraries generally defer the responsibility to correctly call their functions to their users. This is a reasonable assumption since libraries are a (trusted) part of user programs. As a result, libraries are often designed without extensive validation of user inputs. By wrapping a library in SGX, the user inputs become untrusted by nature. Retrofitted input validation, as shown by the vulnerabilities we discovered, is hard to get right. For instance, ineffectual conditions, where both sides of a critical condition check are under the attacker's control (or one side is a constant), are common among the SGX projects we tested.

In total, our system found 52 vulnerabilities in these ten projects. As mentioned earlier, our system implements three attack models: input manipulation, call permutation, and concurrent calls. We attribute a vulnerability to concurrent calls if it is identified only in the multi-threaded mode; furthermore, we attribute a vulnerability to input manipulation if it is identified in a regular/expected sequence of ECALLs (e.g., `opendb` is called before `execute_sql`). Table 3 categorizes the discovered vulnerabilities by attack models. Input manipulation is the most effective attack model with 41 vulnerabilities; while call permutation and concurrent calls found most vulnerabilities related to resource management (e.g., double-frees), as expected.

5.1.2 Policies. Heap information leak: The SGX SDK provides a function to allocate heap memory in an enclave, similar to `malloc` [26]. Data stored on the heap are accessed/retained across different ECALLs. We have observed a common practice of storing public and secret data in the same data structure, e.g., the private key to a TLS certificate may be stored side by side to the IP address, which provides an OCALL interface to send the non-sensitive IP address to the I/O buffer for debug purposes. If an attacker manages to

[1]Library wrappers are often developed by third-party developers. e.g., Intel-SGX-SSL is a library wrapper for OpenSSL developed by Intel.

Table 2. Reported vulnerabilities by types in evaluated projects

Project	Heap info leak	Stack info leak	Ineffectual condition	UAF	Double-free	Stack overflow	Heap overflow	Null ptr deref	Total
mbedtls-SGX	2	3				3	1	2	11
SGX-Tor			2			2	1	4	9
TaLoS	1	1	1	2		1		1	7
Bolos-enclave		1	1		1		1	2	6
Intel-SGX-SSL			3			1		1	5
SGX_SQLite				4					4
SGX-Migration				2	1				3
SGX-Wallet			1		2				3
SGX-Reencrypt			1					1	2
SGXCryptoFile								2	2
Total	3	5	9	8	4	7	3	13	52

Table 3. Reported vulnerabilities by attack model

Policy	Input Manipulation	Call Permutation	Concurrent Calls
Heap info leak	3		
Stack info leak	5		
Ineffectual condition	9		
Use after free		5	3
Double free	1	1	2
Stack overflow	7		
Heap overflow	3		
Null ptr deref	13		
Total	41	6	5

modify a pointer from the public data to the secrets, this OCALL can silently leak the sensitive data.

Another common pattern that may lead to a heap information leak happens when a loop is bounded by a symbolic constraint (i.e., ECALL/OCALL inputs). For example, we have found cases where the loop calls an OCALL to write data to the outside. As the enclave could not determine how many bytes have been written, it depends on the OCALL return to calculate the next byte to write. In some of such cases, we can cause the loop to dump the entire enclave heap memory by manipulating the enclave inputs.

As such, we define a policy to detect heap information leak where the attacker can control both the iteration of the loop and a parameter passed to an OCALL in that loop. If an enclave code matches this pattern, the attacker could potentially invoke the OCALL on a large block of the heap memory. This policy has the following steps:

1. The core module triggers an event to notify the policy module about an infinite loop it encounters. We consider a loop as an infinite loop if its loop condition contains at least one free symbolic variable (i.e., a symbolic variable without constraints).
2. The policy then checks if the loop condition is symbolic.
3. If the loop condition is symbolic, the policy extracts the loop body and analyzes if it contains an OCALL or not.
4. If there is an OCALL, the policy uses the definition of the OCALL (from the API parser) to identify memory pointers in the parameters.
5. The policy reports a potential heap information leak if a pointer points to the enclave heap and can be modified in every iteration of the loop.

```
1  int
2  mbedtls_ssl_flush_output(mbedtls_ssl_context *ssl){
3    ...
4    while(ssl->out_left > 0){  // size_t type
5      buf = ssl->out_hdr + mbedtls_ssl_hdr_len(ssl) +
6                    ssl->out_msglen - ssl->out_left;
7
8      //an indirect call to OCALL
9      ret = ssl->f_send(ssl->p_bio,
10                           buf, ssl->out_left);
11
12     if(ret <= 0)           // ret > ssl->out_left
13        return(ret);
14
15     ssl->out_left -= ret;  // integer overflow
16   }
17   ...
18 }
```

Figure 4. Example heap information leak from mbedTLS-SGX. Comments are added to help understand the bug.

Currently, our policy could not determine sensitivity of the leaked memory and leave this task to human analysis. We plan to introduce a source-code annotation system for developers to mark sensitive data in SGX programs. Our policy found seven potential heap information leaks. We manually confirmed that three of them are critical. A case study from the mbedtls-SGX project is shown in Fig. 4, which allows the attacker to *dump the entire enclave memory*. The `while` loop at Line 4 is bounded by `ssl->out_left`, which has a type of `size_t` (i.e., `unsigned integer`). The symbolic engine shows that `ssl->out_left` is initialized from an ECALL input and is modified by an OCALL return during each iteration (Line 15). This creates an infinite loop if we ensure `ssl->out_left` is not zero. In addition, `ssl->f_send` is a function pointer that points to an OCALL and it has a memory pointer parameter modified in each iteration (buf). i.e., an OCALL is executed in every iteration (Line 9). This piece of code thus satisfies all the requirements of the policy. We note that the return value of `ssl->f_send()`, `ret`, is only validated against `ret <= 0` (Line 12). Any positive value is accordingly legitimate, including values larger than `ssl->out_left`. Because `ssl->out_left` is unsigned, this will lead to an integer overflow at Line 15. Furthermore, because `buf` is modified by `ssl->out_left` in every iteration (Line 5,6), the attacker can point it to arbitrary memory and leak the whole enclave memory.

Stack information leak: In this policy, we focus on the pattern where out-of-bound stack memory is copied to the heap, usually with `memcpy`. This does not necessarily lead to a stack information leak as the data remains inside the enclave. However, the destination data structure on the heap sometimes has a public interface that writes data out of the enclave. Sensitive (stack) data, such as stack canaries and return addresses, could be leaked to an attacker, allowing them to de-randomize the enclave and launch a ROP (return-oriented programming [23]) or other attacks.

Our stack information leak policy tries to locate potentially vulnerable calls of `memcpy` that involve the stack and heap. It currently does not search further to verify whether the destination heap data structure has a public interface to dump its content. We leave this task to manual analysis. That is, this policy can detect stack memory over-reads but needs to manually verify whether the stack memory can be actually leaked (we can apply static points-to analysis to automate it). This policy has the following steps:

1. Check if the length of `memcpy` is symbolic, i.e., the attacker controls how many bytes to copy, leading to a stack memory over-read.
2. Check if the source address of `memcpy` points to the stack. The core module keeps track of the enclave memory allocation and thus can easily do this.
3. Check if the destination address points to the heap.

This policy marked 16 `memcpy` as suspicious, and we confirmed that five of them have a public interface. Fig. 5 shows an example from the mbedtls-SGX project (This vulnerability was reported independently by a GitHub user [10].) Function `ssl_server` uses a fixed size stack buffer, `client_ip[16]`, to store the client's IP address accepted by an OCALL, `mbedtls_net_accept_ocall`. The OCALL also returns the length of client IP in `cliip_len` (Line 7-9). Next, `ssl_server` calls `mbedtls_ssl_set_client_transport_id` and passes both

```
1  int ssl_server(){
2    // vulnerable stack buffer
3    unsigned char client_ip[16] = {0};
4    ...
5    // enclave receives unsafe client ip and its length
6    // in client_ip and  cliip_len
7    if((ret = mbedtls_net_accept_ocall(&listen_fd,
8                    &client_fd, client_ip,
9                    sizeof(client_ip), &cliip_len ))
10                                               != 0){
11     ...
12   }
13   if(opt.transport ==
14               MBEDTLS_SSL_TRANSPORT_DATAGRAM){
15     if((ret =
16             mbedtls_ssl_set_client_transport_id(
17               &ssl, client_ip, cliip_len)) != 0){
18       ...
19     }
20   }
21 }
22
23 int mbedtls_ssl_set_client_transport_id(
24                         mbedtls_ssl_context *ssl,
25                         const unsigned char *info,
26                         size_t ilen){
27   ...
28   mbedtls_free(ssl->cli_id);
29   if((ssl->cli_id = mbedtls_calloc(1, ilen))
30                                          == NULL)
31     return(MBEDTLS_ERR_SSL_ALLOC_FAILED);
32
33   // stack memory overread if ilen > sizeof(info)
34   memcpy(ssl->cli_id, info, ilen);
35   ssl->cli_id_len = ilen;
36   ....
37 }
```

Figure 5. Example stack info leak from mbedTLS-SGX.

`client_ip` and `cliip_len` to it (Line 16-17). In `mbedtls_ssl_set_client_transport_id`, `ilen` (i.e., `cliip_len`) is used to allocate memory for `ssl->cli_id` and later copied data from `info` (i.e., `client_ip`) to it (Line 28-35). Therefore, an attacker can cause a stack memory over-read by manipulating the inputs from `mbedtls_net_accept_ocall`. Moreover, `ssl->cli_id` has a public interface to retrieve its content, including the stack memory.

Ineffectual condition: A conditional check in the enclave becomes ineffectual if the attacker can control its outcome. Therefore, an ineffectual condition allows an attacker to bypass validation, avoid authentication, etc. Ineffectual conditions could be easily identified by checking whether one side or both sides of a conditional check contain symbolic variables. However, we further narrow them down to more probable cases by checking whether the condition is followed by an unconditional control transfer, which is likely an error handling exit. This policy has the following steps:

```
1  int
2  reencrypt(client_id *clid, uint8_t *request,
3             size_t requestlen, uint8_t *response,
4                           size_t *responselen) {
5    ...
6      // keyin originates from the unsafe ECALL param clid
7    if((ret = check_policy(&keyin, &keyout, *clid,
8                           keyIDin, keyIDout))
9                                != REENCRYPT_OK) {
10     ...
11   }
12   // OCALL to get (unsafe) timestamp
13   if(ret = unsafe_timestamp(&timestamp)
14                               != REENCRYPT_OK){
15     ...
16   }
17   // both sides of the conditional statement
18   // contain symbolic variables
19   if (timestamp > keyin->expiration_date ||
20     timestamp > keyout->expiration_date) {
21     ret = REENCRYPT_KEY_EXPIRED;
22     goto err;
23   }
24
25   if ((ret = decrypt(&m, &mlen,c,clen,
26                      keyin)) != REENCRYPT_OK) {
27     ...
28   }
29 }
```

Figure 6. Ineffectual condition from SGX-Reencrypt

```
1  sqlite3* db; // database object
2
3  int sqlite3SafetyCheckOk(sqlite3 *db){
4    u32 magic;
5    if( db==0 ){
6     return;
7    }
8    magic = db->magic; // use
9  }
10 void sqlite3_close(sqlite3 *db){
11   if( sqlite3GlobalConfig.bMemstat){
12     sqlite3_mutex_enter(mem0.mutex);
13     sqlite3GlobalConfig.m.xFree(db);
14     sqlite3_mutex_leave(mem0.mutex);
15   }
16 }
17 void ecall_opendb(const char *dbname){
18   rc = sqlite3_open(dbname, &db);
19 }
20 void ecall_execute_sql(const char *sql){
21   rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
22 }
23 void ecall_closedb(){
24   sqlite3_close(db);
25   // forget to set db = 0
26 }
```

Figure 7. Use-after-free from SGX-SQLite

1. An ineffectual condition is identified if both sides of the condition contain symbolic variables or if one side contains symbolic variables and the other side is a constant.
2. It further checks if the conditional check is followed by an unconditional control transfer within a basic block of no more than five instructions.

This policy detected nine ineffectual conditions. A case study from the SGX-Reencrypt project in Fig. 6 clearly demonstrates the consequences of ineffectual conditions. The condition in Line 19 and 20 is ineffectual because `timestamp`, `keyin->expiration_date`, `keyout->expiration_date` all come from untrusted inputs. `timestamp` is updated by an OCALL `unsafe_timestamp`; the rest two variables originate from an ECALL input, passed in the parameter `clid`. This condition is followed by a `goto` statement, which returns an error code if the key has expired. (Line 21-22). As such, this vulnerability allows an attacker to reuse an expired key.

Use-after-free: Use-after-free is one of the most common memory vulnerabilities in software [28]. In SGX, access to freed memory can cause an enclave to crash, use unexpected values, or even execute arbitrary code. The conventional method to detect use-after-frees is to maintain the allocated/free status of the memory and verify each memory access against it. Use-after-free is generally caused by faulty memory management. It becomes more notable with the call permutation and concurrent calls attack models.

This policy leverages the multi-threaded execution support in the core module. Specifically, it hooks the `free` function and each memory dereference, and performs the following actions in the callback:

1. If a *free* function is called, the policy requests the core module to pause the associated thread until other threads have completed N instructions.
2. If a memory dereference event is triggered, the policy validates raises an alert if the addressed memory has been freed.

This policy detected eight use-after-frees, five by call permutation, and three by concurrent calls. Fig. 7 shows one of the use-after-frees in the SGX-SQLite project. The original SQLite library lets its user maintain its main data structure, `sqlite3*`. The SGX-SQLite library is considered a user (wrapper) of SQLite. It creates an `sqlite3` object in `ecall_opendb`, uses that object in `ecall_execute_sql`, and destroys that in `ecall_closedb`. Because the enclave forgets to reset db to null (Line 25), the following call sequence will lead to a use-after-free vulnerability: `ecall_opendb`, `ecall_closedb`, `ecall_execute_sql`. We note that a proper fix to this problem must use locks to correctly update the db pointer, as well as synchronizing the access to the database.

Double-free: Double free is another type of common resource management vulnerability. Our double-free detection policy follows a similar approach as the use-after-free policy except that it checks whether a block of memory has been

```
1  int
2  ecall_change_master_password(const char* old_password,
3                               const char* new_password) {
4    if (strlen(new_password) < 8
5          || strlen(new_password)+1 > MAX_ITEM_SIZE)
6      ...
7    call_status = ocall_load_wallet(&ocall_ret,
8                              sealed_data, sealed_size);
9    ...
10   sealing_status = unseal_wallet(
11                 (sgx_sealed_data_t*)sealed_data,
12                              wallet, plaintext_size);
13   ...
14   if (strcmp(wallet->master_password,
15                       old_password) != 0) {
16         ...
17   }
18   strncpy(wallet->master_password, new_password,
19                              strlen(new_password)+1);
20   ...
21   sealing_status = seal_wallet(wallet,
22                  (sgx_sealed_data_t*)sealed_data,
23                              sealed_size);
24   free(wallet); // first free
25   if (sealing_status != SGX_SUCCESS) {
26     free(wallet);   // second free
27     ...
28   }
29   ...
30 }
```

Figure 8. Example double-free from SGX-Wallet

```
1  int ssl_client(client_opt_t opt, char* headers[],
2                 int n_header, unsigned char* output,
3                                      int length){
4
5    unsigned char buf[16385]; // var for SSL certificate
6    unsigned char psk[32];   // vulnerable local buffer
7    ...
8
9    if(strlen(opt.psk)){
10     ...
11     psk_len = strlen( opt.psk ) / 2;
12
13     // if len(opt.psk)>64,  overflow buffer psk
14     for(j = 0;j < strlen( opt.psk );j += 2){
15       c = opt.psk[j];
16       ...
17       psk[j/2] = c << 4;
18       ...
19       c = opt.psk[j + 1];
20       ...
21       psk[j/2] |= c;
22     }
23   }
24 }
```

Figure 9. Example stack overflow from mbedtls-SGX

freed more than once. We pick an example from the SGX-Wallet project to demonstrate the vulnerability (Fig. 8). SGX-Wallet aims at securely storing sealed data on the untrusted system by sealing the data in the enclave. Function `ecall_change_master_password()` allows a user to change the master password of the wallet after validating the old password against the sealed wallet. Specifically, it loads the sealed wallet (stored in the untrusted world) with an OCALL (Line 7, 8), unseals it (Line 10-12), and then compares it to the old password (Line 14, 15). Clearly, an invalid new password or errors in `seal_wallet` will free the wallet twice, leading to a double-free vulnerability (Line 24-26).

Stack overflow: Stack overflow is a common software vulnerability. It is still a critical issue for SGX-based software. An attacker can exploit stack overflows to overwrite control data on the stack, as well as sensitive enclave data, such as secret keys. Our policy leverages `__stack_chk_fail` to detect stack overflows, which is a compiler intrinsic inserted by the compiler into programs to detect stack overflows. The policy uses symbolic execution to actively trigger the `__stack_chk_fail` function by calculating path constraints leading to them. Normally, stack overflows can be exploited to hijack return addresses on the stack. However, that might be blocked by the stack canary. We found some interesting cases where an attacker could break the enclave trust model without additional information leaks to bypass the canary. Such an example is shown in Fig. 9.

Function `ssl_client()` in Fig. 9 takes `opt`, an unsafe object originated from an ECALL input. The SGX SDK's basic memory sanitizer ensures the `opt` object is of the correct size. However, its field, `opt.psk`, is an unconstrained string, i.e., it can have arbitrary length. In this function, the loop (Line 14-22) copies the string's bytes to a local buffer, `psk`, without checking whether the string fits in the buffer, leading to a stack overflow. Even worse, `psk` is stored below another buffer, `buf`, which is used to temporarily store the SSL certificate. As such, an attacker can overwrite the certificate with malicious input. This example demonstrates the limitation of the SDK's annotation-based basic sanitization: it can only provide basic protection to top-level data structures. Even though the developer could use `user_check` to sanitize lower-level data structures, it is not convenient and usually incompletely applied.

Heap overflow: This policy is similar to AddressSanitizer [36]: the core module maintains the heap bounds. The policy hooks memory write functions. When triggered, the policy first locates the destination heap buffer and checks if the memory write function overruns the buffer. For performance reasons, we hook two memory write functions, `memcpy` and `memset`.

An example heap overflow we detected is shown in Fig. 10. The function allocates memory for `cfg` to accommodate the unsafe `unix_socket_path`, if `parse_port_config` receives `defaultaddr` as NULL. A `smartllist` generator uses `addrport` to define the `unix_socket_path`. There is a *potential* integer overflow in Line 5 that may lead to smaller memory being allocated [2] and further a heap overflow in Line 21.

[2]The SGX hardware/software may put a limit on the size of the parameter. Our emulator assumes there is no such limit.

```
static port_cfg_t* port_cfg_new(size_t namelen)
{
  ...
  // integer overflow causes less memory allocated
  port_cfg_t *cfg = tor_malloc_zero(
                  sizeof(port_cfg_t) + namelen + 1);
  ..
  return cfg;
}
static int
parse_port_config(smartlist_t *out, ..,
                        const char *defaultaddr, ..){
  ...
  size_t namelen = unix_socket_path ?
                        strlen(unix_socket_path) : 0;
  port_cfg_t *cfg = port_cfg_new(namelen);
  ..
  if (unix_socket_path) {
    ...
    // heap overflow if integer overflow triggered
    memcpy(cfg->unix_addr, unix_socket_path,
                                namelen + 1);
    ...
    tor_free(unix_socket_path);
  }
}
```

Figure 10. Example heap overflow from SGX-Tor

```
1  static sgx_status_t SGX_CDECL sgx_sgxEncryptFile(void* pms){
2    ...
3    // _tmp_encMessageOut could be NULL, results in
4    // _in_encMessageOut to be NULL
5    if (_tmp_encMessageOut != NULL && _len_encMessageOut != 0) {
6      if ((_in_encMessageOut = (unsigned char*)
7                  malloc(_len_encMessageOut)) == NULL) {
8      }
9      ...
10   }
11   sgxEncryptFile(_in_decMessageIn, _tmp_len,
12                  _in_encMessageOut, _tmp_lenOut);
13
14   if (_in_encMessageOut) {
15     if (memcpy_s(_tmp_encMessageOut, _len_encMessageOut,
16               _in_encMessageOut, _len_encMessageOut)) {
17     }
18   }
19 }
20
21 void sgxEncryptFile(unsigned char *decMessageIn, size_t len,
22                  unsigned char *encMessageOut, size_t lenOut){
23   uint8_t p_dst[lenOut];
24   ...
25   // encMessageOut should be checked for NULL
26   memcpy(encMessageOut, p_dst, lenOut);
27 }
```

Figure 11. Null pointer dereference from SGX-CryptoFile

Null pointer dereference: Null pointer dereference often results in a segmentation fault unless the exception is handled. This policy uses the symbolic engine to query whether the address of a memory access is 0 or not. Fig. 11 shows an example from the SGX-CryptoFile project. In this example, `_tmp_encMessageOut` originates from the unsafe `pms` and can be NULL. This leads to a null pointer dereference in Line 26. We note that the auto-generated enclave interface function, `sgx_sgxEncryptFile`, uses `memcpy_s` to avoid this problem. However, the internal function `sgxEncryptFile` fails to do the same. It seems that the developer was unaware that the interface could set the `memcpy` destination to NULL.

5.2 Performance

We evaluated our prototype on an Ubuntu server (18.04 LTS) with an Intel Core i7 processor and 32 GB of memory. We built the test SGX projects with Intel SGX Linux SDK (v2.5) and Clang/LLVM (v9.0). We allocated 30 hours for each test project. Projects like SGX-wallet and SGX-CryptoFile finished within 4 hours. Overall, the multi-thread mode runs 6.5*x* times longer than the single-thread mode because of the frequent locks and delays. Our prototype is compatible with most of the enclave code we tested except instructions for some hardware features (e.g., AES-NI instruction set), which are not supported by the underlying Triton and QEMU engines. Due to the limited memory size, we could apply at most 3 policies simultaneously at a time. Symbolic engines are known to consume a large amount of memory.

6 Related Work

Iago attacks: Iago attacks, an inspirational work by Checkoway et al., are closely related to the COIN attack model [6]. They question the security of systems such as Overshadow [8] that protect user-space applications against complex, vulnerable, or compromised operating system kernels with the help of virtualization. It demonstrates that a malicious OS kernel can compromise a user process by manipulating the syscall returns alone. Our system targets SGX enclaves. The attack surface of an enclave is more flexible than that of a user process because an enclave exposes both ECALL and OCALL functions, and the attacker can invoke these APIs in arbitrary orders. COIN attacks accordingly consist of four attack methods with the additional concurrency, order, and nested attacks that are not in Iago attacks. We also built an extensible framework to check the enclave code for common vulnerabilities.

SGX security: In recent years, there has been a number of research on the micro-architectural side-channel attacks against Intel SGX [7, 18, 38, 41]. For example, Foreshadow leverages a speculative execution bug from Intel x86 processors to reliably leak plaintext enclave secrets from the CPU cache [38]. And SgxPectre exploits the race condition between speculatively executed memory references and the latency of branch resolution to steal seal and attestation keys [7]. Compared to these systems, we focus on the software attack models common to the enclave software and build an extensible framework to evaluate the security of enclave software regarding these attack models.

On the other side, there are relatively few efforts to study the enclave software interface. Lee and Kim demonstrated that uninitialized padding bytes from the enclave could leak sensitive data [24]. Asyncshock proposed a method to exploit known race conditions in the unsafe multi-threaded code of the enclave [42]. Recently, a comprehensive study on trusted runtime code evaluates eight widely used enclave runtimes and reports ten distinct classes of sanitization flaws [39]. In comparison, our research proposes the COIN attacks, a systematic model of attacks against the enclave software, and built a framework to evaluate the enclave code with these attack methods. This framework can be used to test the enclave code before releasing. Haven and SCONE [2, 3] are two systems that host unmodified applications/containers in SGX secure enclave. However, they do not provide any confidentiality or integrity guarantees and ignores the SGX coding recommendations from Intel. So, we do not consider similar systems in our threat model. Finally, Glamdring is a source-level framework that automatically partitions an application into untrusted and enclave parts by identifying functions that may be exposed to or may affect sensitive data [25]. Our system would be useful to check the security of the interfaces of generated enclaves.

Vulnerability detection: There are many different software vulnerability detection methods, such as fuzzing and concolic/static/dynamic analysis. In this section, we could only mention work related to SGX or other TEEs (trusted execution environment). Fuzzing is popular for its effectiveness and scalability but might be lacking in completeness and compatibility. For example, because of TEE's locked-down nature, dynamic analysis is difficult to apply [13]. To address this problem, PARTEMU introduces an emulation based AFL fuzzing method specific to ARM TrustZone. As ARM TrustZone applications are tightly coupled with TrustZone operating systems, PARTEMU focuses on how to handle TZOS calls [14]. Our framework overcomes incompleteness using concolic execution and is compatible with the SGX enclave through instruction emulation.

Moat applies formal verification to prove that an enclave does not leak its secrets to a powerful adversary. It requires annotations on the sensitive data and tracks the data flow to verify if an adversary could observe them [37]. Instead, our framework only requires the source code and detects vulnerabilities of common categories such as information leaks and use-after-frees. Similar to other best-effort vulnerability detection systems, our framework cannot guarantee the absence of vulnerabilities of any specific types.

SGX-Step is a framework that allows a user to single-step (instruction-level) enclave execution [40]. This system is useful to verify vulnerabilities reported by our system. A new ASLR scheme, SGX-Shield, secretly bootstraps the enclave memory space layout with a finer-grained randomization [35]. However, there are also SGX-ROP, Guard's Dilemma, and Dark-ROP [5, 23, 33], which demonstrate that return-oriented programming attacks could be mounted against an ASLR enabled SGX enclave if there is a memory vulnerability in the enclave. SGXBounds [20] offers protection against out-of-bounds memory accesses in the SGX enclave. This protection may detect some of the memory overflow vulnerabilities detected by our system with additional cost memory and runtime.

7 Summary

We have described the COIN attacks that can be used to target SGX-based code through its exposed ECALLs and OCALLs. The COIN attacks consist of Concurrency, Order, Inputs, and Nested call attacks. We further presented the details of an extensible framework, based on instruction emulation and symbolic execution, that implements the COIN attack models with policy plugins. Each policy detects a specific type/pattern of vulnerabilities. Additional policies can be integrated into the system to improve effectiveness. Our experiments with ten open-source SGX projects demonstrate the various vulnerabilities common in these projects.

8 Availability

The prototype is available as an open-source project at https://github.com/mustakcsecuet/COIN-Attacks with the GPL-3.0 license.

9 Acknowledge

We would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This project was partially supported by National Science Foundation (NSF) under Grant 1453020 and 1738912. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF. We also thank the open-source QEMU, LLVM/Clang, and Triton projects.

References

[1] Alberto Sonnino. 2019. SGX-wallet. https://github.com/asonnino/sgx-wallet.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 689–703.

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

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[5] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient code-reuse attacks against Intel SGX.. In *27th USENIX Security Symposium (USENIX Security 18)*. 1213–1227.

[6] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the system call API is a bad untrusted RPC interface.. In *ASPLOS*, Vol. 13. 253–264.

[7] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[8] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 2–13.

[9] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[10] Eadom. 2018. Stack memory leak issue from SGX-mbetls project, reported by an independent GitHub user. https://github.com/bl4ck5un/mbedtls-SGX/issues/13.

[11] Fan Zhang. 2019. SGX-mbedtls. https://github.com/bl4ck5un/mbedtls-SGX.

[12] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. Iron: functional encryption using Intel SGX.. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 765–782.

[13] G. Beniamini. 2019. FuzzZone. https://github.com/laginimaineb/fuzz_zone/tree/master/FuzzZone.

[14] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, Michael Grace, Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, Hayawardh Vijayakumar, et al. 2019. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020)(To Appear)*.

[15] Intel. 2019. Intel SSL. https://github.com/intel/intel-sgx-ssl.

[16] Jim Gordon. 2018. Microsoft* Azure confidential computing with IntelÂő SGX. https://intel.ly/2Db5x1Z.

[17] KAIST INA. 2019. SGX-Tor. https://github.com/kaist-ina/SGX-Tor.

[18] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer.. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*.

[19] Kudelski Security. 2019. SGX-Reencrypt. https://github.com/kudelskisecurity/sgx-reencrypt.

[20] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 205–221.

[21] Large-Scale Data & Systems (LSDS) Group. 2019. TaLoS. https://github.com/lsds/TaLoS.

[22] Ledger. 2019. BoLoS. https://github.com/LedgerHQ/bolos-enclave.

[23] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 523–539.

[24] Sangho Lee and Taesoo Kim. 2017. Leaking uninitialized secure enclave memory via structure padding. *arXiv preprint arXiv:1710.09061* (2017).

[25] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic Application Partitioning for Intel {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 285–298.

[26] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. 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*. ACM, 10.

[27] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *Hasp@ isca* 10, 1 (2013).

[28] mitre. 2019. 2019 CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[29] Nelly Porter, Jason Garms, Sergey Simakov. 2018. Introducing Asylo: an open-source framework for confidential computing. https://bit.ly/2YtVwof.

[30] Rafael Pires, David Goltzsche, Sonia Ben Mokhtar, Sara Bouchenak, Antoine Boutet, Pascal Felber, Rüdiger Kapitza, Marcelo Pasin, and Valerio Schiavoni. 2018. CYCLOSA: Decentralizing Private Web Search Through SGX-Based Browser Extensions0. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

[31] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB âĂŞ A secure database using SGX.. In *To appear in the Proceedings of the IEEE Symposium on Security & Privacy, May 2018*. IEEE. https://www.microsoft.com/en-us/research/publication/enclavedb-a-secure-database-using-sgx/

[32] Ricardo de Souza Costa. 2019. SGXCryptoFile. https://github.com/rscosta/SGXCryptoFile.

[33] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical enclave malware with Intel SGX. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 177–196.

[34] Secure Systems Group (SSG) at Aalto University. 2019. SGX-migration. https://github.com/SSGAalto/sgx-migration.

[35] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs.. In *NDSS*.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker.. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 309–318.

[37] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2015. Moat: Verifying confidentiality of enclave programs.. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1169–1184.

[38] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution.. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 991–1008.

[39] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. 2019. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1741–1758.

[40] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM, 4.

[41] Jinwen Wang, Yueqiang Cheng, Qi Li, and Yong Jiang. 2018. Interface-Based Side Channel Attack Against Intel SGX. *arXiv preprint arXiv:1811.05378* (2018).

[42] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves.. In *European Symposium on Research in Computer Security*. Springer, 440–457.

[43] Yerzhan Mazhkenov. 2019. SGX-SQLite. https://github.com/yerzhan7/SGX_SQLite.

A Artifact Appendix

A.1 Abstract

Our artifact contains source files of the COIN attacks framework and eight evaluated policies. The framework implements the concurrency, order, and inputs models of COIN attacks. It is based on the following open-source projects: QEMU, Triton, and Clang/LLVM. The implementation has three major parts: 1) a python EDL file parser, 2) an LLVM libtool to extract information from the LLVM bitcode, and 3) the core module to analyze the enclave code for vulnerabilities based on the three COIN attack models. The artifact also has several small benchmarks (including a clone of the SGX_SQLite project). The artifact does not require the host machine to have an SGX-enabled CPU, only Intel SGX SDK. Following is the directory tree of the artifact:

```
COIN_Attack
└─ src
│   ├─ semantics
│   │   ├─ llvm_src
│   │   │   └─ lib/Transforms/EnclaveSemantics
│   │   └─ pyedl
│   └─ core
│       └─ Triton/src/enclaveCoverage
├─ scripts
│   ├─ PoCs
│   └─ SGX_SQLite
└─ PoCs
```

A.2 Description

- **Algorithm:** COIN attacks (concurrency, order, and inputs) to evaluate the security of SGX enclaves.
- **Program:** SGX enclave built with Intel SGX SDK (v 2.5) in a Linux distribution.
- **Compilation:** Clang (v 10.0.0), included with the artifact.
- **Data set:** Eight micro-benchmarks and a clone of the SGX-SQLite project from https://github.com/yerzhan7/SGX_SQLite.
- **Run-time environment:** Our artifact has been developed and tested on Linux environment (Ubuntu 18.04.3 LTS server OS). The software dependencies are either included in the artifact or installed with `apt`.
- **Hardware:** We used Intel Core i7 (32 GB memory) and Intel Xeon E3-1275 (64 GB memory) for testing. Complex SGX test projects (e.g. mbedTLS) may consume upto 48 GB memory.
- **Execution:** The micro-benchmarks takes about 30 mins each and the SGX-SQLite analysis takes 2+ hours on our test machines.
- **Output:** The artifact outputs a report containing one or more vulnerabilities (each report includes 200 instructions and input mapping for debug purpose).
- **Experiments:** Experiment results are explained at https://github.com/mustakcsecuet/COIN-Attacks/blob/master/scripts/EXPERIMENT.md.
- **How much disk space required:** The artifact after build requires around 45 GB disk space (35 GB to host the compiler).
- **How much time is needed to prepare workflow:** It takes about 1 hour to build the artifact.
- **Publicly available:** `DOI: 10.5281/zenodo.3605266`. The latest version (preferred) will always be available at https://github.com/mustakcsecuet/COIN-Attacks. Use the Zenodo project for reproduce purpose only.
- **Code licenses:** GNU General Public License v3.0.

A.3 Installation

The setup is divided into three steps: install Intel SGX SDK at `/opt/intel/sgxsdk/`; build the Clang/LLVM compiler from the included source code; install the Triton symbolic engine along with the z3 solver from the source too. Installation depends on the Ubuntu packages such as gcc, g++, python, binutils, automake, etc. A more detailed instruction is available at https://github.com/mustakcsecuet/COIN-Attack/blob/master/README.md.

A.4 Experiment workflow

First run the python EDL parser to parse the EDL file; then use Clang to compile (`SGX_Mode=SIM`) the SGX project to generate both the enclave bitcode (using the gold plugin) and enclave shared object (.so); next, use the LLVM libtool to extract the memory layout information from the bitcode; finally, use the python code coverage tool to test the enclave interface against COIN attacks. The code coverage tool can explicitly set the number of seeds to try for each ECALL sequence and the number of instructions to emulate.

```
python edlParse.py Enclave/Enclave.edl
opt -load LLVMEnclaveSemantic.so < enclave.bc
python coverage.py enclave.so N_SEED N_INST
```

Our artifact includes eight small benchmarks to test an enclave for eight different vulnerabilities. The source of benchmarks is available under `COIN-Attacks/PoCs/`. It also includes the SGX-SQLite project (real benchmark) because some of the vulnerabilities we have reported have been patched in the latest code.

A.5 Evaluation and expected result

Our artifact includes eight different types of vulnerability detection policies. They can be categorized as memory vulnerabilities, information leaks, and control-flow hijacks. Each vulnerability report follows a common format:

```
<vulnerability report header>
<last 200 executed instructions>
<the memory to input mappings>
```

In the following, we describe the vulnerability report header of each policy.

- **Use-after-free:**
  - Memories that are used after free.
  - Where are the memories allocated.
  - Where are the memories freed.
  - Where are the memories used after freed.
- **Double free:**
  - Memories that are freed twice.
  - Where are the memories allocated.
  - Where are the memories freed at first.
  - Where are the memories freed in the second time.
- **Stack overflow:**
  - The function of the overwritten return address.
- **Heap overflow:**
  - Memories that are written out of bound.
  - Where is the heap memory overflowed.
- **Heap memory leak:**
  - The name of OCALL that leaks the information.
  - Where is the OCALL used.
- **Stack memory leak:**
  - Where is the memcpy called.
  - Memories that have been overread.
  - Memories that hold the leak data.
- **Ineffectual condition:**
  - The weak conditional statement.
  - Where is the error code following the conditional statement.

The report shows the vulnerability at the assembly code level. We plan to integrate a debug plugin to translate that to the source code location. The detail of vulnerability reporting is available at https://github.com/mustakcsecuet/COIN-Attacks/blob/master/scripts/EXPERIMENT.md.

A.6 Experiment customization

The artifact assumes the target SGX project follows a standard directory structure (application code in App/ and enclave code in Enclave/). The bash scripts and the PoCs are useful resources to guide the customization of an experiment.

A.7 Notes

There are a few known issues with the framework, mostly due to the limitation in the dependencies:

- **Instruction not recognized:** QEMU does not support some recently introduced instructions, such as `endbr64` from Intel CET. Currently, we handle them by throwing an exception.
- **ISA too complex:** Some instructions (e.g., AES-NI) are too complex to support in the Triton symbolic engine, a well-known limitation (https://github.com/JonathanSalwan/Triton/issues/793). Our framework skips these instructions as well.
- **Nested Calls:** Our artifact does not include the fourth model of COIN attacks. We plan to support it in future.

If you find other bugs in our project (on GitHub), please raise an issue in the GitHub or email to the first author.