EnclaveFuzz: Finding Vulnerabilities in SGX Applications

Liheng Chen[1,2,3*], Zheming Li[3*], Zheyu Ma[3], Yuan Li[3,4], Baojian Chen[1,2], Chao Zhang[3,4†]
[1]Institute of Information Engineering, Chinese Academy of Sciences.
[2]School of Cyber Security, University of Chinese Academy of Sciences.
[3]Institute for Network Sciences and Cyberspace of Tsinghua University. [4]Zhongguancun Laboratory.
[2]{chenliheng21,chenbaojian20}@mails.ucas.ac.cn, [3]{lizm20, mzy20}@mails.tsinghua.edu.cn, [4]lydorazoe@gmail.com

Abstract— **Intel's Software Guard Extensions (SGX) offers an isolated execution environment, known as an enclave, where everything outside the enclave is considered potentially malicious, including non-enclave memory region, peripherals, and the operating system. Despite its robust attack model, the code running within enclaves is still prone to common memory corruption vulnerabilities. Moreover, such an attack model may introduce new threats or amplify existing ones. For instance, any direct memory access to untrusted memory from within an enclave can lead to Time-of-Check-Time-of-Use (TOCTOU) bugs since attackers are capable of controlling the whole untrusted memory. Moreover, null-pointer dereference may have a more severe security impact since the zero page controlled by the operating system is also considered malicious. Current fuzzing solutions, such as SGXFuzz and FuzzSGX, have limitations detecting such SGX-specific vulnerabilities.**

In this paper, we propose EnclaveFuzz, a multi-dimension structure-aware fuzzing framework that analyzes enclave sources to extract input structures and correlations, then generates fuzz harnesses that can produce valid inputs to pass sanity checks. To conduct multi-dimensional fuzzing, EnclaveFuzz creates data for all three input dimensions of an enclave, including both parameters and return values that enter an enclave, as well as direct untrusted memory access from within an enclave. To detect more types of vulnerabilities, we design a new sanitizer to detect both SGX-specific vulnerabilities and typical memory corruption vulnerabilities. Lastly, we provide a custom SDK to accelerate the fuzzing process and execute the enclave without the need for special hardware. To verify the effectiveness of our solution, we applied our work to test 20 real-world open-source enclaves and found 162 bugs in 14 of them.

I. Introduction

The Trusted Execution Environment (TEE) is an important security mechanism to protect modern systems. With the help of hardware, TEEs could provide enforced isolation for critical code and data. SGX is one of the most popular TEEs developed by Intel, which provides a secure enclave running within a host application. The enclave is protected by the CPU and could remain secure even if the host application is compromised. More importantly, the attack model of SGX even assumes that everything except the CPU itself could be malicious, including the operating system, peripherals, etc. Such assumptions make SGX attractive for users seeking top-level security. For example, Signal [1] application utilizes SGX for contact discovery service to protect private contact information.

Although SGX provides such a robust attack model at the system level, many SGX applications are developed in C/C++, which are prone to memory vulnerabilities like buffer overflow, use-after-free, etc. What is worse, the special attack model may bring new vulnerabilities to SGX applications or make some vulnerabilities more prevalent. For example, null pointer dereference, which usually considered as a bug, can potentially transform into a vulnerability within the SGX environment. The reason is that the zero address falls under the control of an untrusted operating system. Such a subtle, yet crucial difference in SGX's attack model amplifies the security risks of such common bugs.

Consequently, researchers and developers are actively pursuing automated techniques for bug detection within SGX applications. Fuzzing is one of the most powerful techniques to find bugs in software. However, current fuzzing solutions meet several roadblocks while testing enclaves. To test an enclave, SGXFuzz [2] tests enclave binaries as a black box and tries to recover the input structure from page-fault feedback. However, it may fail for the following reasons. First, the SGX SDK will encapsulate the developer's code and automatically add sanity checks for enclave inputs. More specifically, the SGX SDK generates bridge code according to Enclave Definition Language (EDL) files provided by developers for both host application and enclave. The bridge code contains input sanity checks and will reject illegal inputs, which makes fuzz inefficient. A fuzzer without knowledge of the exact input structure may waste time to bypass those checks or even worse never reach the developer's code. Also, although SGXFuzz could guess input parameter structures with page fault feedback, it may still lead to insufficient fuzzing due to a lack of input dimensions. As one SGX application is consisted of two components, host application and enclave, those two parts rely on two special types of function calls to communicate with each other, which are `ECalls` (call from host appli-

[*]The first two authors contributed equally to this work.
[†]Corresponding author: chaoz@tsinghua.edu.cn

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24819
www.ndss-symposium.org

cation to enclave) and `OCalls` (call from enclave to host application). More importantly, both parameters and return values of `ECalls` and `OCalls` are directional. Specifically, both parameters of `ECalls` and return values of `OCalls` are considered as inputs of enclave. Besides that, the enclave shares the same process memory space with host application and any direct untrusted memory access is also considered as input of enclave. FuzzSGX [3] tries to extract input structures and test enclaves via host mutations but both of them do not consider untrusted memory access.

Second, the whole fuzzing process heavily relies on bug oracles to monitor the program execution state and identify potential bugs. However, SGXFuzz relies solely on page fault signals for error detection, which inevitably introduces false negatives. And the sanitizers utilized by FuzzSGX cannot detect SGX-specific vulnerabilities well.

Lastly, although the enclave is loaded by the host application during startup, the enclave maintains its own separate heap and stack memory regions. To make it possible, each enclave contains redundant memory management routines as well as context-switching mechanisms, which will greatly slow down the fuzzing process.

To address these challenges, we propose EnclaveFuzz, a multi-dimension structure-aware fuzzing framework, which extracts input structures from enclave sources and performs fuzzing across multiple input dimensions of an enclave. Specifically, to achieve more code coverage, EnclaveFuzz extracts input structures and automatically generates harnesses capable of creating valid inputs that bypass sanity checks. To trigger more bugs, EnclaveFuzz analyzes enclave's trust boundaries to identify input dimensions of enclaves and feeds fuzzing to all three input dimensions of enclaves, including `ECall` input, returned value of `OCall`, and untrusted memory access from enclave. Also, to detect more types of bugs, EnclaveFuzz incorporates a sanitizer that could detect both memory corruption bugs and SGX-specific vulnerabilities, including null pointer dereference and Time-of-Check-Time-of-Use (TOCTOU). Moreover, to speed up fuzzing, EnclaveFuzz provides an optimized SGX SDK which is compatible with Intel's official one and eliminates the redundant memory management and context switch routines, and it also encapsulates the enclave as a standard shared object (.so) to perform fuzzing without special SGX hardware requirements.

Our experiment shows structure-aware inputs of EnclaveFuzz can enhance the success rate of reaching developers' code, achieving an average success rate of nearly 99%, which is 2.94 times higher than that of SGXFuzz. Additionally, it leads to a 3.62 times increase in code coverage compared to SGXFuzz. Moreover, the optimized SDK could speed up fuzzing by 6.91 times on average. As a result, EnclaveFuzz finds 162 bugs across 14 SGX applications, including both memory corruption bugs and SGX-specific vulnerabilities.

In summary, we have made the following contributions:

1) We propose a multi-dimensions structure-aware fuzzing framework EnclaveFuzz, that can effectively find the vulnerabilities for the SGX applications.
2) We optimize the SGX SDK to speed up fuzzing, improving the efficiency of fuzzing in detecting vulnerabilities.
3) We design and implement an SGX-specific sanitizer that can detect both memory corruption and SGX-specific vulnerabilities.

II. Background

With the rapid growth of cloud computing services, people tend to build their applications on shared resources. Nevertheless, potential attacks from other tenants or even the service provider can raise concerns. A TEE is an isolation solution to such problems. SGX is Intel's implementation of the TEE, which provides users with a hardware-protected user mode enclave to process sensitive data. SGX holds a robust threat model that anything except the enclave itself and CPU is untrusted, including the operating system (OS), peripheral devices, et al.

This section will briefly explain how SGX works and introduce the fuzzing technique to help understand EnclaveFuzz.

A. Intel Software Guard Extensions (SGX)

Fundamentally, SGX is a set of instructions that can establish a CPU-protected memory area for the execution and storage of critical code and data. To streamline the development of SGX applications, Intel provides an SGX SDK for developers, which contains enclave management routine code, C/C++ runtimes, and other auxiliary tools. A developer needs to specify interfaces using Enclave Definition Language (EDL) and generate bridge code for both the host application and enclave. Subsequently, the host application and enclave are compiled into two separate binaries. The host application binary is a standard executable file depending on the OS, for example, ELF (executable and linkable format) under the Linux platform. The enclave binary image will be signed cryptographically and the booted enclave instance can be attested remotely, which means developers can verify the enclave's executing environment is trusted by querying Intel's attestation servers or their own ones. Those mechanisms ensure the security of enclaves while transferring and loading. While enclave creating, it is loaded into Enclave Page Cache (EPC), which is in Processor Reserved Memory (PRM). The contents of EPC memory are encrypted and can only be decrypted within the CPU, which enforces the enclave security during runtime, as Figure 1 shows.

SGX memory model and runtime routines: The memory model of an SGX application differs from a standard user-space program. As Figure 1 shows, the enclave shares the same virtual address space with the host application during runtime. The whole enclave, including its control structures, sensitive data et al., is placed in a contiguous virtual memory space named Enclave Linear Address Range (ELRANGE). The SGX Enclave Control Structure (SECS) represents the identification of each enclave, the number of Thread Control Structures (TCS) decides the maximum number of concurrent threads running at the same time. More importantly, the enclave holds

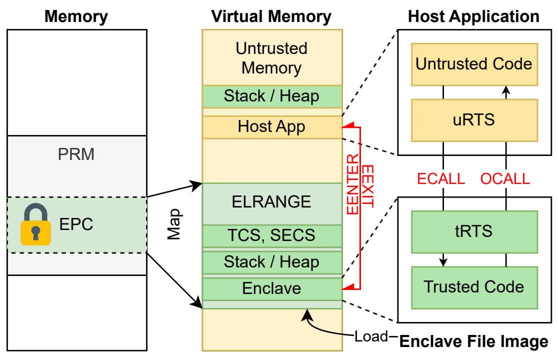


Fig. 1: SGX memory model and runtime routines

```
1  enclave {
2      trusted {
3          public int ecall_demo(
4              [in, count=10] int* arg1,
5              [out,size=arg3] char* arg2,
6              size_t arg3);
7      };
8      untrusted {
9          int ocall_demo([user_check] struct* arg1);
10     };
11 };
```

Listing 1: EDL Example

its own stack and heap memory regions since the host memory is untrusted and unprotected.

Although the host application and the enclave share the same virtual address space, they do have several limitations while accessing each other's memory regions. An enclave could access the whole host application's memory, but could not execute any code within the host memory. The host application cannot access anything within enclave's memory. For the host application, `EENTER` and `ERESUME` instructions are the only way to execute the code of enclave. All classical subroutine instructions, like `call` or `jump`, will fail if the target is within enclave.

SGX provides several instructions for enclave creation, execution, and return and the Intel SGX SDK [4] wraps those instructions into runtime routines to provide C-compatible interfaces for developers. A developer needs to specify 2 sets of interfaces using Enclave Definition Language, named enclave call (`ECALL`) and outside call (`OCALL`). As one SGX application is split into two parts, host application, and enclave, `sgx_edger8r` generates bridge code for them according to EDL, named untrusted bridge (`uBridge`) and trusted bridge (`tBridge`), respectively. The bridge functions are wrapped interfaces that will be invoked by the host application and enclave.

To be more specific, the host application firstly invokes the `uBridge` function, then the untrusted runtime system (`uRTS`) switches context and transfers control to the trusted runtime system (`tRTS`) within the enclave. The `tBridge` will simply check the input and pass all data to the real `ECALL` function if valid. Similarly, enclave calls the `tBridge` firstly to execute functions in the host application.

In summary, an SGX application has a special memory model and highly relies on the routine provided by SDK to manage its lifecycle and execute trusted code.

Enclave Interface Definition: As mentioned before, SGX SDK will generate bridge code for both the host application and enclave according to the EDL from the developer. More specifically, the bridge code helps to transfer data between enclave and the host application.

List 1 shows an example of an EDL file. Line 2 and Line 8 indicate interfaces defined within the curly brackets are `ECALL`s and `OCALL`s. Take the `ECALL` in Line 3 as an example, its definition is based on C language function declaration, plus with the attributes defined within square brackets. Those attributes are the most important part of an EDL definition, which indicates the direction and size of the pointer parameter. It may be safe to pass a pointer as a parameter in traditional programs, but that is not the case for enclaves. Enclave can only use C language to declare interfaces in EDL, and parameters only will be shallow copied, which means for pointers, only the address value will be passed into the enclave. The pointer still points to the untrusted memory region belonging to the host application. It makes the enclave prone to TOCTOU attacks if the pointer is used after being checked without deep copy since content outside the enclave can be modified by the untrusted host at any time. One possible solution is to deep copy each pointer, which will inevitably slow the execution due to a large number of memory operations. As a result, Intel's SGX SDK provides additional attributes to handle pointers. Developers can determine the size and direction of copied data. For `ECALL`s, attribute `in` indicates that the pointed-to memory content needs to be copied into the enclave and `out` for out to the enclave. For `OCALL`s, attribute `in` means to copy buffer to the host application's memory and vice versa.

List 2 shows the bridge code of the `ECALL` from List 1 generated by SGX SDK, the listed code is simplified for better understanding. The bridge code uses a generated structure `ms_ecall_t` to pack data transferred between enclave and the host application, including all parameters and the return value. The `uBridge` (Line 9 to 17) packs data into structure `ms` and calls `uRTS` function `sgx_ecall` to enter enclave. The `uRTS` then executes `EENTER` instruction and passes the pointer of `ms` to `tRTS`, followed by several `tRTS` routines, and finally passes to the `tBridge` (Line 20-42). Next, the `tBridge` performs a sanity check of the input data, shown in Line 32 and will throw errors if failed. In the end, the tBridge executes the real `ECALL` function (Line 41) from the developer.

Additionally, the `count` attribute can be used to denote the number of elements, which means the total copied bytes are `count * sizeof(*ptr)`. The EDL also supports pointers with dynamic length, as Line 5 in List 1 shown. The size of `arg2` is dynamically determined by the value of

```
typedef struct ms_ecall_demo_t {
    int ms_retval;
    int* ms_arg1;
    char* ms_arg2;
    size_t ms_arg3;
} ms_ecall_demo_t;

/* Enclave_u.c */
sgx_status_t ecall_demo(sgx_enclave_id_t eid,
                        int* retval, int* arg1,
                        char* arg2, size_t arg3) {
    // marshall inputs
    ms_ecall_demo_t ms;
    ms.ms_arg1 = arg1;
    // call uRTS to enter enclave
    sgx_ecall(eid, 0, &ocall_table_Enclave, &ms);
}

/* Enclave_t.c */
static sgx_status_t SGX_CDECL sgx_ecall_demo(
                                          void*pms)
    // check marshalled data outside enclave
    CHECK_REF_POINTER(pms, sizeof(ms_ecall_demo_t)
    );
    // unmarshall inputs
    ms_ecall_demo_t* ms =
        SGX_CAST(ms_ecall_demo_t*, pms);
    int* _tmp_arg1 = ms->ms_arg1;
    size_t _len_arg1 = 10 * sizeof(int);
    // check size
    if (sizeof(*_tmp_arg1) != 0
        && 10 > (SIZE_MAX / sizeof(*_tmp_arg1))) {
        return SGX_ERROR_INVALID_PARAMETER; }
    // check parameter 1 outside enclave
    CHECK_UNIQUE_POINTER(_tmp_arg1, _len_arg1);
    // allocate enclave memory
    _in_arg1 = (int*)malloc(_len_arg1);
    // copy data into enclave memory
    memcpy_s(_in_arg1,_len_arg1,_tmp_arg1,
    _len_arg1);
    // call uRTS to execute the real ECALL
    function
    ms->ms_retval =
        ecall_demo(_in_arg1,_in_arg2,_tmp_arg3);
}
```

Listing 2: Generated Bridge Code

arg3. For other complicated pointer types, such as structures with nested pointers, the developer can explicitly state using user_check attribute. This indicates that only the pointer will be passed and developers need to perform manual checks, otherwise will introduce potential security risk.

In summary, developers can use in or out to denote the direction and specify the length of copied data using count and size. Also, developers can use user_check and then perform manual checks.

Intel SGX Software Development Kit (SDK): Besides the above-mentioned bridge code, SGX SDK contains context switch routines. While initializing an enclave, SGX SDK will copy enclave code and data from the enclave image to EPC, arrange the layout of enclave memory, measure the legality of loaded code and data, and set metadata. Also, the SDK needs to recover EPC and prune metadata while destroying an enclave.

Specifically, SGX SDK executes EENTER and EEXIT instructions while entering and exiting the enclave, which costs about 3,800 and 3,300 cycles according to Eleos [5]. Moreover, SDK needs to clean the register status and perform context switches every time executing ECALL/OCALL functions, as well as manage the separated enclave memory, including both stack and heap. Such routines may tamper the efficiency of fuzzing.

Additionally, the SDK generates a table g_ecall_table that maps each ECALL function ID to its respective function body, as well as ocall_table that maps OCALLs functions. When the host applications makes an ECALL, it specifies the ECALL's function ID. The SGX runtime will use this ID to look up the corresponding function pointer, and then call the corresponding function inside the enclave and vice versa.

B. *Finding Bugs Using Fuzzing Technique*

Fuzzing is an efficient and automatic technique to discover vulnerabilities in software. A naive fuzzer executes a Program Under Test (PUT) with randomly generated data and observes the execution status. AFL [6] utilizes code coverage as feedback to retain test cases that trigger new code as good seeds. Tools like SanitizerCoverage [7] are widely used in fuzzing. Subsequent works [8]–[11] improve the fuzzing process by guiding fuzzers with data-flow feedback. To enable fuzzing on targets with complex inputs, structure-aware fuzzing has been proposed, which constructs well-formed data instead of a buffer of raw bytes as test cases. Unstructured fuzzers have to rely on brute force to bypass input validations or sanity checks within PUT, while structure-aware fuzzer [12], [13] could generate more valid inputs.

However, AFL-like fuzzers are unsuitable to fuzz API-based targets, including library code, system calls, ECALLs of an enclave and et al. A fuzz harness is needed to connect the fuzz engine and the functions under test. A harness can be either generated from templates or manually written. Syzkaller [14] is a system call based fuzzer that uses pre-defined system call templates to generate harness for the Linux kernel. Difuze [12] and KSG [15] automatically extract system call interfaces and generate templates to facilitate kernel fuzzing. LibFuzzer [16] is a fuzzing engine to test user space library code and can be easily linked to the target library with a manually written harness. OSS-Fuzz project [17] from Google has already found 28,000 bugs from 850 projects with libFuzzer and harnesses from developers. Further, some researchers also propose solutions to automatically generate fuzz harnesses. FUDGE [18] tries to construct a harness from existing code by code slicing. FuzzGen [19] automatically abstracts an API dependence graph from libraries and then synthesizes the harness for testing.

To efficiently detect bugs, fuzzing is often coordinated with sanitizers. AddressSanitizer (ASan) [20] is one of the most widely used sanitizers to detect commonly seen memory corruption bugs, including stack/heap overflow, use-after-free and et al. ASan instruments memory operations within the target program, surround memory objects with red zone, and constructs a shadow memory map to check if the visited memory region is valid during runtime. Any memory operations on the red zone will result in an ASan alert to notify potential bugs. UBSan [21] can detect undefined behavior per C/C++ standards, and thread sanitizer [22] can detect data race bugs in concurrent systems. In addition to ASan, fuzzing can also be combined with other sanitizers [23]–[25] that can detect various types of bugs. SGXBOUNDS [26] proposes a sanitizer for SGX, which can detect out-of-bound bugs. However, it only supports 32bit memory access, which greatly limits its usage.

In order to fuzzing enclave, SGXFuzz [2] tries to recover input structures from black-box enclave binaries and may fail on complex structures. Also, SGXFuzz may generate invalid test cases that fail to pass the sanity checks within the enclave and thus wastes fuzzing time. Besides that, SGXFuzz only focuses on a single dimension of input during fuzzing process, which can lead to false negatives. FuzzSGX [3] shares the same limitation with SGXFuzz that is not aware of the boundary between trusted and untrusted memory and does not consider untrusted memory access as an input dimension to trigger TOCTOU. Moreover, both of them lack a sanitizer for SGX-specific vulnerabilities.

Generally, there are three challenges that need to be addressed to perform efficient fuzzing on SGX enclaves.

1) **Insufficient understanding of the input structures and dimensions:** Since SGX SDK encapsulates the developer's code and automatically adds sanity checks for enclave inputs, a fuzzer without knowledge of the exact input structure may waste time bypassing these checks or might not even reach the developer's code. Also, the special communication pattern between the host application and the enclave exposes new input dimension.
2) **Limited Bug Oracle Capabilities:** The effectiveness of the fuzzing process greatly depends on bug oracles, which check program execution states and identify potential bugs. However, current solutions [26] can only detect limited types of vulnerabilities and overlook SGX-specific ones.
3) **Slow fuzzing process due to redundant management routines:** Each enclave contains redundant routine code for memory management and context switching, which hampers the efficiency of the fuzzing process.

III. Design

To overcome the aforementioned challenges in fuzzing enclaves, we propose a multi-dimension structure-aware fuzzing framework EnclaveFuzz, with a fuzzing-optimized SGX SDK, and an SGX-specific sanitizer, as Figure 2 shows.

EnclaveFuzz firstly extracts the security boundaries of enclaves from EDL, then automatically generates a harness for multi-dimension structure-aware fuzzing. Second, EnclaveFuzz provides an optimized SGX SDK to seamlessly build enclave under test into a `Virtual Enclave`, which aims to speed up fuzzing. Third, to detect more vulnerabilities, EnclaveFuzz is embedded with a sanitizer to detect SGX-specific vulnerabilities and commonly seen memory corruption

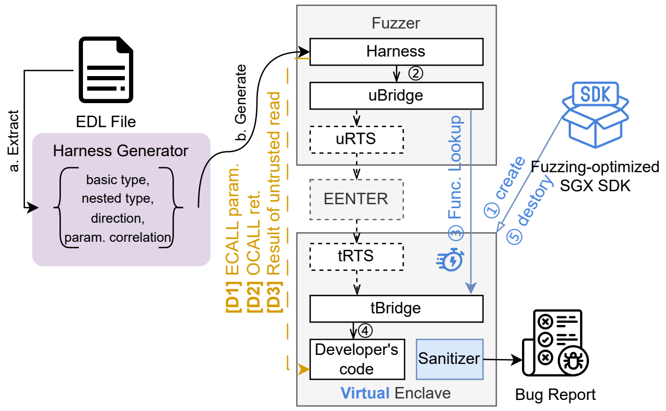


Fig. 2: Overview of Enclave

TABLE I: Data flow direction of pointer parameters

Type	Dir. Attr.	Size Attr.	**Direction**	**Bytes** allocated
ECALL	IN	**Fixed:** `size	count = val.`	enter enclave ✓	**Fixed:** `value specified`
	OUT		exit enclave ✗	
OCALL	IN	**Dynamic:** `size = param.` `user_check`	exit enclave ✗	**Dynamic:** `runtime decided`
	OUT		enter enclave ✓	

bugs. Next, we elaborate on the design of each part in the following subsections.

A. Multi-dimension Structure-aware Fuzzing

As there are sanity checks in the enclave trusted bridge, such as memory range check, size check, etc., fuzzers that directly feed random data may fail to pass those checks.

As Figure 2 shows, EnclaveFuzz firstly extracts the security boundaries of the enclave, as described in the EDL file, by analyzing attributes of the arguments and return values of `ECALLs` and `OCALLs`. During the extraction process, EnclaveFuzz parses the EDL file per enclave to get basic type information of each parameter and return value. For improved structure-awareness, EnclaveFuzz iteratively analyzes nested type, like pointer, structure and array. For opaque pointers, EnclaveFuzz tries to search for a matched structure definition.

More importantly, EnclaveFuzz needs to determine how those data will be transferred since the data flow between the host application and the enclave can be directional. For parameters passing by value, it is relatively straightforward to determine the direction according to the type of interface, parameters of `ECALLs` are copied into the enclave, while return values of `OCALLs` are also copied into the enclave.

When dealing with pointer parameters, EnclaveFuzz utilizes the direction attributes (`in/out`) in EDL. Specifically, EnclaveFuzz follows the rules from Table I to allocate data for pointer parameters. Only pointers that will be deep copied into the enclave are assigned data by EnclaveFuzz, as allocating data that will not enter the enclave is meaningless. As for size, EnclaveFuzz uses the explicitly defined value from EDL for pointers with a fixed size. For pointers specified as `size = parameter`, the length of copied data is dynamically determined by the value of `parameter`, and it forms a correlation between the pointer parameter and `size` parameter. In such cases, EnclaveFuzz records the correlation of two parameters and prepares the `size` parameter firstly, and then allocates data accordingly during run-time.

A special case is `user_check`, as Line 9 from List 1 shown, which means only the pointer itself will be transferred into the enclave. Under such circumstances, it is hard to infer the direction and size needed from EDL since the developer could use the pointer freely. EnclaveFuzz treats those pointers as input since they are potential inputs of an enclave and allocates data with a randomly generated `count` value. That means totally `count * sizeof(*ptr)` bytes are prepared for the parameter. Also, to test if the enclave is prone to the null pointer dereference vulnerability, EnclaveFuzz also passes the null value to pointer parameters with a manually set probability.

More importantly, EnclaveFuzz needs to prepare data for all three input dimensions: `ECALL`, `OCALL`, and untrusted memory access. As for `OCALLs`, EnclaveFuzz employs a hook-based strategy, which intercepts the return value of an `OCALL` function and parameters used in the enclave. The reason is that typically an enclave relies on the `OCALL` function to interact with the untrusted OS, and may check data from `OCALLs`. The randomly generated data may make those checks fail and thus make the fuzzer incapable of exploring deeper code. As the host application and the enclave share the same virtual process memory, there is a chance that the enclave directly loads data from the untrusted host memory regions. EnclaveFuzz alters specific host memory addresses that reported by the TOCTOU sanitizer to trigger potential bugs.

In summary, EnclaveFuzz extracts the security boundary from EDL and automatically generates a structure-aware fuzzing harness, which feeds data from all three input dimensions to test the target enclave.

B. Fuzzing-optimized SGX SDK and Virtual Enclave

As aforementioned, an enclave runs within the same virtual process memory as the application on the host while holding its own stack and heap. To make it possible, SGX SDK needs to manage the isolated memory and switch contexts while entering and leaving the enclave, which inevitably slows down the fuzzing. Such a special memory layout is optimal for hardware enclaves since the enclave must be placed within the separated EPC memory to protect sensitive information. However, such a design is unnecessary for fuzzing and the management routines also slow down the fuzzing. To eliminate redundant routines and speed up fuzzing, EnclaveFuzz provides a fuzzing-optimized SGX SDK that packs tBridge code as well as the enclave code from the developer into a standard shared library (.so), named `Virtual Enclave`, shown in Figure 3(a). Since the fuzzing-optimized SGX SDK does not rely on any SGX hardware instructions, EnclaveFuzz can test enclaves without specific hardware requirements.

Specifically, EnclaveFuzz utilizes one bit from the shadow map used by SGX-specific sanitizer to distinguish host memory and enclave memory within the same heap and stack memory regions. With the help of such a design, there is no need to manage isolated memory regions, enabling faster creation and destruction of `Virtual Enclaves`. More importantly, EnclaveFuzz does not execute any expensive context switch routines while entering and leaving the enclave, which directly looks up the function tables and transfers the control flow to the corresponding code. The `tBridge` is retained in the `Virtual Enclave`, as it contains sanity checks generated from EDL and may lead to false positives if removed.

In summary, the `Virtual Enclave` shares the same heap and stack with the host application (fuzzer) and EnclaveFuzz relies on shadow maps to distinguish between them.

C. Vulnerability Detection

A well-designed sanitizer could save a lot of human effort to analyze crashes and identify the root causes of bugs, as well as report vulnerabilities that do not trigger crashes. However, previous work [26] has several limitations, supporting only 32-bit memory space and incapable of detecting SGX-specific vulnerabilities.

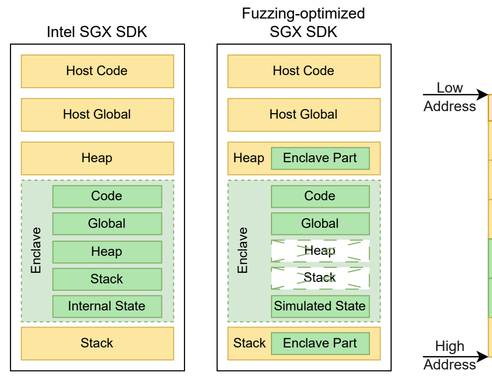
EnclaveFuzz provides a sanitizer that supports four types of vulnerabilities, as shown in Figure 3(b). In general, EnclaveFuzz supports detecting both memory corruption bugs and SGX-specific vulnerabilities, which are out-of-bound memory access, dangling pointer dereference, null pointer dereference and TOCTOU. We elaborate on the policies used to detect each type of vulnerability in the following subsections.

Detect out-of-bound and dangling pointer dereference. EnclaveFuzz follows a similar design with ASan to detect memory corruption bugs, which utilizes a shadow map and red zones to detect invalid memory access. EnclaveFuzz also utilizes an 8-bit shadow memory to record memory allocation status and checks during memory access operations. Besides, EnclaveFuzz uses the higher bit of the shadow byte to differentiate between the memory associated with the host application and that of the enclave.

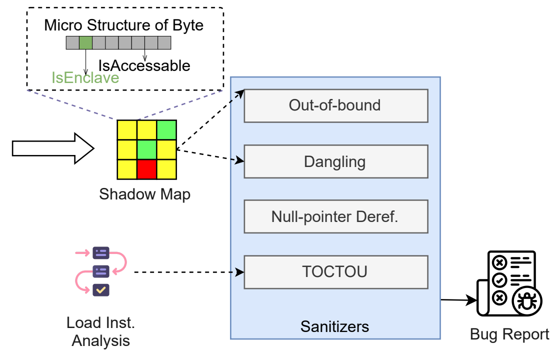
Detect null pointer dereference. As null pointer dereferences in a regular program typically results in undefined behavior, developers may consider it as a bug rather than a vulnerability. However, the zero address is under the control of the untrusted OS and can be mapped to place malicious content. To detect such a problem, EnclaveFuzz marks the zero address page inaccessible as a guard page and registers a signal handler to report any zero address access.

Detect TOCTOU vulnerability. Time-of-Check-Time-of-Use (TOCTOU) is a temporal flaw often originating from race conditions in multi-process programs. Existing solutions [27]–[30] focus on identifying race conditions that operate on the same memory and rely on task scheduling to trigger potential TOCTOU bugs, overlooking the issue of untrusted memory access in SGX applications. Midas [31] mitigates double-fetch bugs in system-calls via a snapshot-based strategy which does not work for SGX applications since OS is untrusted.

However, the unique threat model of SGX applications makes such vulnerability more commonly seen in enclaves.



(a) Memory layout.



(b) EnclaveFuzz Sanitizer.

Fig. 3: Enclave memory layout and design of sanitizer

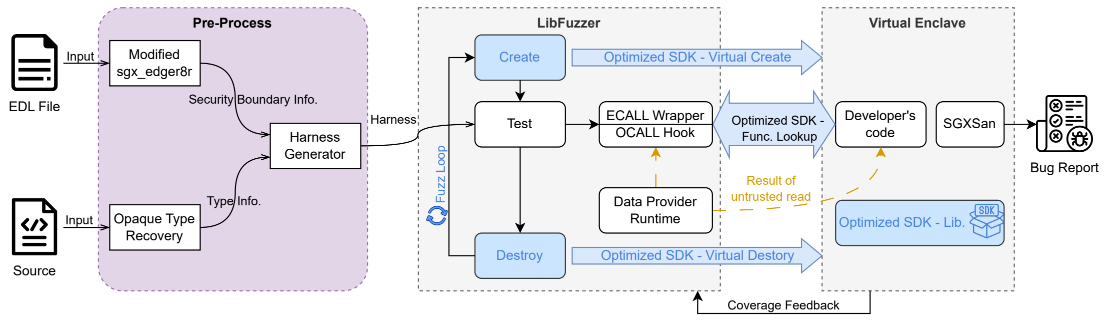


Fig. 4: EnclaveFuzz Architecture

The reason is that the attacker is assumed to control the whole untrusted memory, as well as manipulate the OS's thread scheduling. For an enclave, any direct use of host memory within checks may lead to security concerns. To detect TOCTOU, EnclaveFuzz performs a two-stage analysis. Firstly, we perform compile time analysis. EnclaveFuzz collects all load instructions and utilizes define-use chains to determine if a load instruction could be used by a subsequent compare instruction. If so, those load instructions are marked as comparison-loads. Secondly, at run-time, EnclaveFuzz hooks all load instructions and maintains a FIFO queue to record memory addresses visited by comparison-loads. When other load instructions are executed, EnclaveFuzz checks for any overlapping between the visited address and those recorded in the queue. Any overlaps are considered as a potential TOCTOU vulnerability. More importantly, to validate those potential bugs, EnclaveFuzz employs a fuzzing strategy. Specifically, EnclaveFuzz checks whether the overlapped address belongs to the untrusted host application's memory region and mutates its value if so before the second load instruction executed.

IV. Implementation

The software architecture of EnclaveFuzz is shown in Figure 4. In the pre-processing stage, EnclaveFuzz analyzes both the EDL file and the source code to extract the security boundary and type information, and then generates the harness including `ECALL` wrappers and `OCALL` hook functions. During fuzzing run-time, the fuzzer loads the enclave as a dynamic shared library and provides data for all three input dimensions of the enclave. We will open-source EnclaveFuzz[*] to facilitate further research in this domain.

A. *Pre-process*

Modified `sgx_edger8r` To accurately extract security boundary information of an enclave from the EDL, we extended the Intel SGX Edger8r tool (`sgx_edger8r`) to analyze both the EDL and source code to gather attributes of both `ECALL`s and `OCALL`s, including the type information, nested type, size, direction, the correlation between arguments, as well as array dimensions.

Opaque Type Recovery As developers can declare and use incomplete types (e.g. `typedef struct Struct* StructPtr;`) in `ECALL`s, making it impossible to extract structure layout information directly from these interface. To address this, EnclaveFuzz uses an LLVM IR pass to extract all structure layouts from the enclave source code during compile-time. The host application(fuzzer) then queries these results when encountering an opaque pointer to get the layout of the pointed structure.

Harness Generator We implemented an LLVM IR Pass to generate the harness for fuzzing, including `ECALL` wrappers and `OCALL` hooks. EnclaveFuzz firstly creates a function

[*]https://github.com/LeoneChen/EnclaveFuzz

declaration in IR and then inserts instructions to prepare data for a specific interface. For parameters with basic type, EnclaveFuzz allocates as much data as the size of the type. For structures, EnclaveFuzz recursively allocates data for each field of the structure, where the maximum recursion depth can be set by the user. As for opaque data types, EnclaveFuzz utilizes saved opaque type information and attempts to match them by name. Additionally, EnclaveFuzz provides the option to specify a probability to modify the return value of `OCALL`.

B. libFuzzer

We used `libFuzzer` as fuzz engine. In the fuzz loop, EnclaveFuzz begins by using `dlopen` to create a `Virtual Enclave` and then tests enclave interfaces within the generated harness. The `ECALL` wrapper retrieves data from the data provider run-time and subsequently invokes the `ECALLs`. This process includes input packing in uBridge, function table lookup, and finally entering the `Virtual Enclave`. The `OCALL` hooks are executed when the enclave makes `OCALLs` to modify return values. Last, EnclaveFuzz calls `dlclose` to destroy `Virtual Enclave` and unmap its memory. This step ensures that the start of each fuzz round is stateless and deterministic, which is crucial for reproducing crashes. We constructed data provider run-time based on LLVM `FuzzedDataProvider` to generate structure-aware inputs. As enclave can access host memory, EnclaveFuzz instruments load instructions and relies on the data provider to generate data to manipulate untrusted memory region, guided by the TOCTOU sanitizer.

C. Virtual Enclave

We developed a fuzzing-optimized SGX SDK to construct a `Virtual Enclave`. The SDK is based on Intel SGX SDK simulation mode but eliminates isolated enclave memory management, context switches, and internal state management. Instead, it incorporates standard shared library creation and destruction procedures, along with function table lookup for entering and exiting `Virtual Enclave`. The rest of the SGX SDK remains unchanged to ensure compatibility and consistency.

The optimized SDK has the same interface abstraction as Intel's SDK, the user only needs to modify the build system, such as the Makefile, to link SGX applications with the optimized SDK, then recompile their project to apply EnclaveFuzz.

In addition, we implement a sanitizer of EnclaveFuzz named SGXSan to detect the vulnerabilities aforementioned. We made modifications to the AddressSanitizer LLVM pass and developed a sanitizer run-time to perform out-of-bound, dangling pointer dereference checks within the `Virtual Enclave`. As for TOCTOU, EnclaveFuzz analyzes load and compare instructions and then instruments load instructions to perform untrusted memory mutation during run-time. To distinguish between memory associated with the enclave and the host application, we redesigned the layout of the shadow byte in ASan, which utilizes the higher bit of the shadow byte to mark memory ownership.

V. Evaluation

In this section, we evaluated EnclaveFuzz on real-world SGX applications to answer the following research questions:

- **RQ1:** Can EnclaveFuzz find new vulnerabilities from real-world enclaves, especially for SGX specific vulnerabilities, like TOCTOU or null pointer Dereference?
- **RQ2:** Does the structure-aware fuzzing strategy contribute to code coverage improvement? Specifically, can EnclaveFuzz generate more valid inputs to pass the sanity checks and explore more unique code than existing tools?
- **RQ3:** Does the multi-dimension inputs contribute to trigger more bugs?
- **RQ4** Does the optimized SDK increase the speed of fuzzing enclaves?
- **RQ5:** What is the overhead of the sanitizer?

A. Evaluation Setup

1) Target Enclaves: We select 20 open-sourced SGX applications from both Intel and third-party developers. The chosen applications have a wide range of functionalities and complexities to provide a comprehensive and diverse evaluation of our solution. The specific applications used in the evaluation are detailed in Table II, both the number of `ECall` and basic blocks of each enclave are listed to help understand the scale and complexity of tested applications. Also, we include the specific version used for evaluation.

2) Experiment Setup: We conducted evaluation using a server equipped with an Intel Xeon(R) 8358 CPU and 1024GB of RAM. Each enclave was tested for a duration of 24 CPU core hours and the experiment was repeated three times. The evaluation started with an empty seed and each fuzzing process is bind to a single CPU core per target.

B. Vulnerability Findings in Real-world Enclaves

1) Overall Result: In summary, EnclaveFuzz reported hundreds of crashes among 14 enclaves during test. Since a single bug can potentially cause multiple crashes, it was necessary to deduplicate these crashes. We deduplicated crashes by comparing the PC value, stack trace, and error message from sanitizer and then carried out a manual analysis of the unique crashes. Through this analysis, we were able to identify a total of 162 distinct bugs. Table III shows detailed information about bugs found by EnclaveFuzz.

A significant number of vulnerabilities were discovered in TaLoS [40], which ports libressl [49] into enclave and exposures 207 `ECALL` interfaces to the host application. However, it annotates most of the pointer parameters of `ECALL` and `OCALL` as `user_check` and thus the SGX SDK does not generate any sanity checks for those pointers. But only few security checks are implemented by the developer and thus results in a huge number of bugs.

Among all bugs detected by EnclaveFuzz, null-pointer dereference and TOCTOU contributes the most cases, we

TABLE II: Target Enclaves

Enclave	Version	#ECALLs	#BBs	Publisher	#Bugs
intel-sgx-ssl [32]	c2c75f	2	86k	Intel	$1^{\dagger}$+ $2^{\ddagger}$
Launch Enclave (LE) [33]	8abc6d	2	4.6k	Intel	0
Prov.Cert.Enclave (PCE) [33]	8abc6d	3	6.8k	Intel	0
Prov.Enclave (PVE) [33]	8abc6d	4	10.5k	Intel	0
Quoting Enclave (QE) [33]	8abc6d	2	10.1k	Intel	0
ehsm [34]	190b9a	32	89k	Intel	$11^{\dagger}$+ $1^{\ddagger}$
trusted-function-framework [35]	1c5ab9	3	116k	Ant Group	3^{*}
wasm-micro-runtime [36]	5fb511	2	32k	Bytecode Alliance	$15^{\dagger}$+ $1^{\ddagger}$
wolfssl [37]	2b670c	22	33k	wolfssl	0
sgxwallet [38]	22d6c9	22	9.9k	SKALE Network	$3^{\dagger}$
SGX_SQLite [39]	c470f0	3	33k	3rd party	3^{*}
TaLoS [40]	9c9599	207	75k	3rd party	96^{*}
mbedtls-SGX [41]	eab8e3	6	30k	3rd party	4^{*}
sgx-wallet [42]	8d15df	5	3.5k	3rd party	$10^{\sharp}$
sgx-dnet [43]	0fe09c	3	6.6k	3rd party	2^{*}
plinius [44]	a25162	3		3rd party	2^{*}
BiORAM-SGX [45]	d86dab	10	10k	3rd party	2^{*}
bolos enclave [46]	573464	10	18k	3rd party	0
SGXCryptoFile [47]	92f3cd	2	2.8k	3rd party	2^{*}
sgx-reencrypt [48]	6f0659	5	4.6k	3rd party	4^{*}
Total					162

* reported to developers
† already confirmed by developers
‡ already fixed by developers
♯ project has been archived, bugs cannot be submitted

believe this trend indicates a general lack of awareness among developers regarding the unique security assumptions of SGX, especially when handling pointers that across the enclave security boundaries. This underscores the importance of both a comprehensive understanding of security boundaries and an automatic detection tool to catch those bugs before they become a part of the production codebase.

Additionally, to compare the bug detection capabilities of EnclaveFuzz with the current state-of-the-art SGX fuzzing tool, SGXFuzz, we conducted a 24-hour experiment. We used the default setting shipped with SGXFuzz and also deduplicated crashes using its own analysis script. The results are listed in Table IV.

The results show that EnclaveFuzz can find vulnerabilities from real-world enclaves. More importantly, EnclaveFuzz successfully found SGX-specific vulnerabilities and commonly seen memory corruption bugs.

2) Responsible Disclosure: When we detect and manually verify a bug, we follow a 90-day disclosure period and notify the affected vendor through email or repository issues. The detailed disclosure status is listed in Table II.

3) Case Study: Next, we elaborate on several bugs to show the effectiveness of EnclaveFuzz.

Case Study 1: Stack Overflow in SGX_SQLite As List 3 shows, a stack variable `zDbHeader` is allocated by `sqlite3BtreeIoeb` and used in `unixRead`. And the value of variable `got` comes from untrusted `OCALL osRead`. An attacker could manipulate the return value of `seekAndRead` to trigger a stack overflow.

Case Study 2: Double Fetch in TaLoS List 4 demonstrates

```
SQLITE_PRIVATE int sqlite3BtreeOpen(...) {
  unsigned char zDbHeader[100];
  rc = sqlite3PagerReadFileheader(...,zDbHeader);
//unixRead is called, zDbHeader is passed to pBuf
}
static int unixRead(..., void *pBuf, ...) {
  got = seekAndRead(...);
  memset(&((char*)pBuf)[got],0,amt-got);//overflow
}
static int seekAndRead(...) {
  // OCALL to get got from host
  got = osRead(id->h, pBuf, cnt);
  return got;
}
```

Listing 3: Stack Overflow in SGX_SQLite

```
1  // EDL: public void ecall_EVP_MD_CTX_destroy(
2  //                 [user_check] EVP_MD_CTX *ctx);
3  void ecall_EVP_MD_CTX_destroy(EVP_MD_CTX *ctx) {
4    return EVP_MD_CTX_destroy(ctx);
5    /*ctx is annotated as user_check, thus only the
       pointer itself will be copied into enclave*/
6    // call EVP_MD_CTX_cleanup
7  }
8  int EVP_MD_CTX_cleanup(EVP_MD_CTX *ctx) {
9    if (ctx->digest && ctx->digest->cleanup && ...)
10     ctx->digest->cleanup(ctx);
11 }
```

Listing 4: Double Fetch in TaLoS

a classic double fetch bug in SGX, where a `user_check` pointer parameter, `ctx`, is sourced from the untrusted host without properly deep-copying its content. Specifically, an attack can alter `ctx->digest` its initial verification but prior to the subsequent check on Line 9, leading to a potential crash. Moreover, an attacker can manipulate `ctx->digest` after both checks on Line 9 but before the function invocation on Line 10 to hijack the control-flow.

Case Study 3: Use After Free in mbedtls-SGX An `ECALL ssl_conn_teardown` can be used to destroy the handler. The vulnerable class, `TLSConnectionHandler`, incorporates a destructor that executes `mbedtls_pk_free`. If an attacker invokes the `ECALL` consecutively, the `connectionHandler` is freed during the first invocation. On the second invocation, the destructor will be called, which accesses `ctx->pk_info`, resulting in a use-after-free bug.

C. Input Validity and Code Coverage

As discussed earlier, the bridge code generated by the SGX SDK includes sanity checks that reject invalid inputs. It is a waste of time if the inputs never reach the developer's enclave code under test. Also, it is meaningless to simply remove these sanity checks during fuzzing, which will introduce a significant number of false positives. To verify if the input generated by our fuzzer could pass sanity checks and eventually be processed by the developer's enclave code, we instrumented the target enclaves to include a log function at the beginning of each `ECALL` function. As for SGXFuzz, which runs the enclave within a minimal Linux environment nested in a

```
class TLSConnectionHandler { // vulnerable class
  private:
    mbedtls_pk_context pkey;
  public:
    ~TLSConnectionHandler( mbedtls_pk_free(&pkey)
    );
}
void ssl_conn_teardown(void){ // ECALL
  delete connectionHandler; // free
}
void mbedtls_pk_free( mbedtls_pk_context *ctx ) {
  if( ctx == NULL  ctx->pk_info == NULL ) // use
    return;
}
```

Listing 5: Use After Free in mbedtls-SGX

TABLE III: Vulnerabilities and Bugs found by EnclaveFuzz

Type	Enclave	#Bugs	Total
Null-Pointer Dereference	sgx-wallet	7	68
	intel-sgx-ssl	1	
	mbedtls-SGX	2	
	TaLoS	44	
	sgx-dnet	1	
	plinius	1	
	sgxwallet	2	
	sgx-reencrypt	4	
	trusted-function-framework	1	
	wasm-micro-runtim	4	
	BiORAM-SGX	1	
Use After Free	intel-sgx-ssl	2	6
	SGX_SQLite	2	
	mbedtls-SGX	2	
TOCTOU	TaLoS	37	38
	wasm-micro-runtim	1	
Stack Overflow	SGX_SQLite	1	5
	ehsm	1	
	BiORAM-SGX	1	
	SGXCryptoFile	2	
Heap Overflow	sgx-wallet	3	18
	TaLoS	2	
	sgxwallet	1	
	ehsm	11	
	wasm-micro-runtim	1	
Int Overflow	TaLoS	13	15
	sgx-dnet	1	
	plinius	1	
Arbitrarily Read/Write/Execute	trusted-function-framework	1	11
	wasm-micro-runtim	10	
Unchecked Size	trusted-function-framework	1	1
Total	14 Apps		162

```
public void sgxDecryptFile(
  [in,size=len] unsigned char *encMessageIn,
  size_t len,
  [out,size=lenOut] unsigned char *decMessageOut,
  size_t lenOut);
```

Listing 6: Interface with multiple-sized parameters from SGXCryptoFile

QEMU-emulated machine, we leveraged `hprintf` (hypercall printf) to log tested `ECALL` functions to the serial output of QEMU. In both cases, if any sanity check fails, execution terminates before reaching the `ECALL` function, thereby not generating the corresponding log messages. In the end, we used scripts to calculate how many times such log messages occurred and then divided by the total execution times recorded by fuzzer engines to get the successful execution rate of `ECALL`s. Table IV shows the detailed results, and all values presented in the table are arithmetic averages of three repeated experiments. For SGXFuzz, there is a significant proportion of cases in which the input does not even enter the enclave. In contrast, the input from EnclaveFuzz manages to enter the enclave almost every time. We manually analyze cases from `SGXCryptoFile` and find that SGXFuzz infers the wrong size on interfaces that have multiple-sized parameters, shown in List 6. We consider it cannot determine whether `len` is the size of `encMessageIn` or `decMessageOut` and thus infer the wrong size.

In addition to the success rate experiments, we also conducted experiments to evaluate if EnclaveFuzz could cover more code blocks compared with the existing solution, SGXFuzz. As an enclave binary contains not only the code from developers but also the SGX SDK, including the trusted bridge code, memory management routines, etc., it may not be enough to perform a comprehensive evaluation to compare only the coverage of the whole enclave. In addition, we presented three metrics of coverage evaluation results, enclave coverage, effectiveness, and interesting coverage to demonstrate the code coverage performance of EnclaveFuzz, listed in Table IV.

Enclave coverage indicates total coverage of enclave binary, which includes SGX SDK. The formula is shown as follows:

$$EnclaveCoverage = \frac{BB'_{SDK} + BB'_{Developer}}{BB_{SDK} + BB_{Developer}}$$

$BB'_{Developer}$ indicates covered basic blocks belong to the developer's enclave code, BB'_{SDK} indicates covered basic blocks belong to SGX SDK, $BB_{Developer}$ indicates all basic blocks belong to developer's enclave code, and BB_{SDK} indicates all basic blocks belong to SGX SDK.

In addition to enclave coverage, we also introduce a new measurement metric, indicated as effectiveness, which is calculated as the proportion of covered code blocks within the developer's enclave code relative to all covered code. The formula is shown as follows:

$$Effectiveness = \frac{BB'_{Developer}}{BB'_{SDK} + BB'_{Developer}}$$

Effectiveness could illuminate how much of the fuzzer attention is paid to explore the developer's enclave code, which is the primary area of interest.

Lastly, we use interesting coverage to denote the coverage of the developer's enclave code specifically. The formula is shown as follows:

$$InterestingCoverage = \frac{BB'_{Developer}}{BB_{Developer}}$$

This measure provides a clearer picture of how effectively our testing solution is at testing developer's code.

TABLE IV: Experiment Results

Enclave Name	Code Coverage[1]						Input Validity		Bug Findings	
	Enclave Cov.		Interesting Cov.		Effectiveness					
	SGXFuzz	EnclaveFuzz	SGXFuzz	EnclaveFuzz	SGXFuzz	EnclaveFuzz	SGXFuzz	EnclaveFuzz	SGXFuzz	EnclaveFuzz
intel-sgx-ssl	0.75%	18.04%	0.02%	18.39%	1.66%	99.66%	0%	100%	0	3
AE LE	3.85%	11.67%	14.29%	32.08%	1.98%	15.25%	26.89%	100%	0	0
AE PCE	4.10%	13.94%	22.53%	45.34%	3.49%	15.30%	17.48%	100%	0	0
AE PVE	2.36%	8.63%	10.05%	16.95%	6.32%	22.62%	33.15%	100%	0	0
AE QE	2.64%	3.20%	13.23%	6.68%	3.60%	16.13%	5.52%	100%	0	0
SGX_SQLite	2.39%	6.78%	1.45%	7.20%	26.64%	99.96%	30.39%	100%	0	3
TaLoS	5.86%	9.78%	4.66%	10.00%	36.56%	99.58%	53.50%	100%	90	96
mbedtls-SGX	6.54%	30.64%	8.16%	32.64%	53.68%	99.66%	21.23%	100%	1	4
wolfssl	3.64%	42.44%	0.38%	45.00%	7.72%	99.78%	38.27%	99.99%	0	0
sgx-wallet	8.52%	33.10%	12.68%	79.39%	1.42%	39.72%	30.06%	99.99%	1	10
sgx-dnet	5.64%	0.97%	1.13%	0.51%	7.00%	34.92%	69.15%	100%	2	2
plinius	3.07%	2.24%	1.10%	2.19%	7.41%	73.47%	68.41%	100%	2	2
sgxwallet	6.33%	51.81%	7.21%	43.50%	7.74%	25.44%	20.74%	100%	2	3
BiORAM-SGX	4.30%	17.95%	0.55%	1.08%	5.45%	1.66%	48.43%	82.95%	0	2
bolos-enclave	6.71%	7.85%	1.17%	0.48%	4.86%	4.01%	40.10%	84.09%	0	0
ehsm	3.69%	16.91%	3.81%	15.00%	76.97%	81.60%	0%	91.79%	0	12
sgx-reencrypt	8.60%	33.31%	14.92%	31.26%	20.26%	28.26%	84.38%	100.00%	2	4
SGXCryptoFile	5.85%	17.62%	15.04%	80.56%	4.15%	5.88%	0%	100.00%	0	2
trusted-function-frame	2.53%	1.97%	2.13%	1.53%	75.64%	75.22%	0%	100.00%	0	3
wasm-micro-runtime	3.95%	1.67%	2.08%	0.94%	32.64%	46.04%	78.04%	100.00%	5	15
average	4.57%	16.53%	6.83%	23.54%	19.26%	49.21%	33.29%	97.94%	5.25	8.05

[1] For SGXFuzz, the coverage is gathered by the Ghidra script shipped with kAFL fuzzer.
For EnclaveFuzz, the coverage is collected by clang's source-based code coverage feature [50].

To gather the necessary data for these metrics, for EnclaveFuzz, we utilized the `llvm-cov` tool to display the basic block coverage per file and then calculated effectiveness and interesting coverage. On the other hand, SGXFuzz is a binary fuzzer and incompatible with LLVM's coverage tool, we turned to collect coverage information from the execution traces generated by the fuzzer via Intel Processor Trace (PT). And then, we modified the analysis scripts shipped with SGXFuzz to calculate effectiveness and interesting coverage.

In Table IV, the effectiveness of EnclaveFuzz is higher than SGXFuzz on most of enclaves, indicating EnclaveFuzz wastes less time testing SGX SDK. In interesting coverage, EnclaveFuzz covers more developer's enclave code than SGXFuzz in most applications, but for sgx-dnet [43], plinius [44] and BiORAM-SGX [45], both fuzzers showed poor performance on code coverage. We manually reviewed those enclaves' sources and tried to find out the reason. Sgx-dnet is an SGX ported version of a machine learning library Darknet, which can be used to train or test neural networks. All three exposed `ECALL`s from sgx-dnet take complicated structures as input. For example, the first input of `ecall_classify` takes a `user_check` double-linked list and the third input is structural image data, which makes both fuzzers hard to recover the input structure. The coverage over time graphs are placed in Appendix A.

D. Multi-dimension Inputs

As aforementioned, the input dimension of an enclave includes not only the arguments of `ECALL` functions but also returns values of `OCALL` functions and untrusted memory access. Under the attack model of SGX, an attacker is capable to take control of all three input dimensions to perform attacks.

TABLE V: Ablation Study: sanitizer & multi-dimension inputs

Enclave Name	EnclaveFuzz	Fuzzer_NoSan	Fuzzer_1D
	unique crashes generated during experiments		
intel-sgx-ssl	3	2	2
AE LE	0	0	0
AE PCE	0	0	0
AE PVE	0	0	0
AE QE	0	0	0
SGX_SQLite	3	0	2
TaLoS	96	76	59
mbedtls-SGX	4	1	3
wolfssl	0	0	0
sgx-wallet	10	5	6
sgx-dnet	2	1	1
plinius	2	1	1
sgxwallet	3	1	1
BiORAM-SGX	2	2	2
bolos-enclave	0	0	0
ehsm	12	0	11
sgx-reencrypt	4	2	4
SGXCryptoFile	2	2	2
trusted-function-frame	3	3	3
wasm-micro-runtime	16	13	13
Total	162	109	110

Consequently, an effective fuzzing tool needs to evaluate all possible input vectors. To evaluate the effectiveness of multi-dimension inputs from EnclaveFuzz, we conducted an ablation study. This experiment sets up a one-dimension fuzzer, named `Fuzzer_1D`, which only generates inputs for `ECALL` functions, while keeping all other aspects of the system constant. The results are presented in Table V, which proves that multi-dimension inputs is helpful to find bugs. Specifically, input for untrusted memory access is necessary to trigger and detect potential TOCTOU bugs.

TABLE VI: Ablation Study: fuzzing-optimized SDK

Enclave Name	EnclaveFuzz-SIM	EnclaveFuzz-HW	EnclaveFuzz (Opt.SDK)
	ECALLs executed in 24 hours		
intel-sgx-ssl	18K	217	19K
AE LE	155M	63M	454M
AE PCE	153M	58M	483M
AE PVE	123M	44M	11M
AE QE	42M	27M	50M
SGX_SQLite	40M	15M	160M
TaLoS	448K	194K	120K
mbedtls-SGX	1M	122K	1M
wolfssl	370K	17K	23K
sgx-wallet	86M	21M	137M
sgx-dnet	354k	94k	504k
plinius	71k	54k	501k
sgxwallet	430k	218k	1.9M
BiORAM-SGX	1M	26K	9M
bolos-enclave	96M	30M	505M
ehsm	227K	163K	212K
sgx-reencrypt	14M	10M	15M
SGXCryptoFile	2M	467K	18M
trusted-function-frame	13M	3M	3M
wasm-micro-runtime	4M	1M	40M
Speedup rate	2.67×	1×	6.91×

TABLE VII: Optimized SDK Performance

Item	Opt.SDK	Sim w.	HW w.	Sim.	HW.
1K Create	0.14s	1.89s	28.27s	1.50s	25.63s
1K Destroy	0.11s	0.46s	1.38s	0.39s	1.06s
5M ECALL	0.46s	22.89s	34.30s	16.04s	25.49s
5M SwitchlessE-CALL	0.46s	1.46s	1.41s	1.28s	1.59s
5M OCALL	0.05s	3.45s	11.40s	3.37s	12.05s
5M SwitchlessO-CALL	0.06s	3.06s	4.62s	1.38s	1.42s

E. Fuzzing-optimized SGX SDK

To verify if the fuzzing-optimized SGX SDK helps to speed up fuzzing, we conducted an ablation study.

Firstly, we evaluated performance differences in creating and destroying enclave as well as ECALL and OCALL which contain an empty payload, in order to evaluate how much can fuzzing-optimized SDK speed up basic operations.

In Table VII, we evaluated the performance of basic operations run on five modes. Since fuzzing-optimized SGX SDK still relies on shadow map to determine whether the memory is located in enclave, for the purpose of fairness, we modify EnclaveFuzz and run enclave in both simulation mode and hardware mode Intel SGX SDK with a sanitizer that has same functionality as control groups, and also choose simulation mode and hardware mode Intel SGX SDK without sanitizer as another control groups. We found Intel SGX SDK simulation mode is faster than the hardware mode in creating and destroying enclave as well as ordinary `ECALL` and `OCALL`, but about the same in switchless `ECALL` and `OCALL`. Besides, Intel SGX SDK with sanitizer runs slower than without sanitizer. Importantly, fuzzing-optimized SGX SDK ran far faster than others in all basic operations.

In order to evaluate how much can fuzzing-optimized SGX SDK speed up fuzzing, we statistic number of `ECALLs` executed in 24 hour on real world SGX applications. According to Table VI, EnclaveFuzz-SIM run 2.67× faster than EnclaveFuzz-HW, and EnclaveFuzz(Opt.SDK) run 6.91× faster than EnclaveFuzz-HW. In some special enclave, EnclaveFuzz(Opt.SDK) can run slower than EnclaveFuzz-SIM, it's due to these enclaves often need `OCALL` to get untrusted time and check if a period of time has elapsed, since time can be input with random data, then enclave may always find time not expired in a loop.

In order to prove fuzzing-optimized SGX SDK is correct in functionality, we test it with Intel SGXSSL test routine, and Wolfssl benchmark, and finally show it's correctness. All crash inputs found by EnclaveFuzz with optimized SDK is reproducible in EnclaveFuzz hardware mode.

F. Sanitizer Overhead

As Figure 5 shows, we evaluated EnclaveFuzz on the WolfSSL benchmark. SGX version benchmark result is similar to the non-SGX version, SGXSan cost an extra 68.07% which is similar to ASan's 48.22%.

VI. Discussion

A. Future Work

1) Support more SDK: Those make it possible to extract security boundaries from more applications relying on different SDKs and generate harnesses for fuzzing. The implementation of EnclaveFuzz is currently related to the official Intel SGX SDK. Besides that, many third-party SGX SDKs have been published, like Microsoft OpenEnclave [51] and Google Asylo [52]. These SDKs, while being unique in their implementation, share a common design philosophy, making them potential targets for future integration with EnclaveFuzz. For instance, Asylo uses the same tool (`edger8r`) as Intel SDK to generate bridge code for enclaves and provides remote procedure calls (RPCs) for top-level applications, while OpenEnclave offers a similar tool (`oeedger8r`) for the same purpose. Our future plans include extending EnclaveFuzz's capabilities to harness these SDKs and generate effective fuzzing schemes.

2) Explore deeper code: Although EnclaveFuzz outperforms the current solution, there is still room for improvement. EnclaveFuzz primarily focuses on the interfaces to extract input structures. Meanwhile, an area of potential enhancement is in the deeper exploration of both enclave code and application code. These codes may contain valuable constraint information that can guide the generation of more meaningful inputs, enabling EnclaveFuzz to investigate deeper layers of code and uncover more intricate or elusive bugs.

B. Limitation

The reliability of the optimized SGX SDK may lead to false positives. To accelerate the fuzzing process as more as possible, EnclaveFuzz only simulates the necessary control structures required to make the `Virtual Enclave` work, such as the thread control structure (TCS) used for concurrent

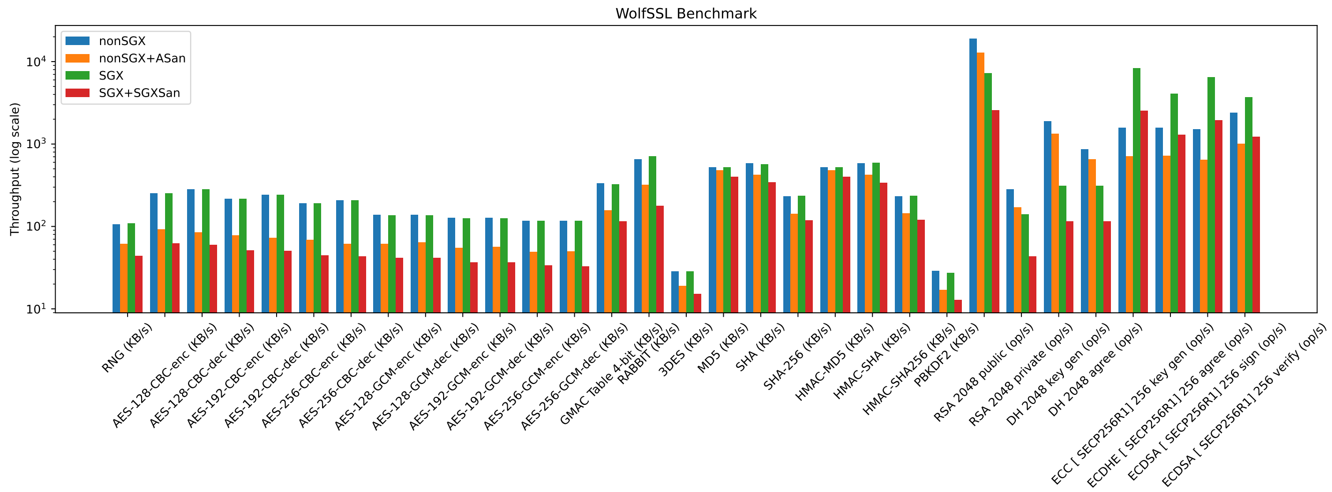


Fig. 5: WolfSSL Benchmark Results

```
sgx_status_t sgx_create_report(
  const sgx_target_info_t *target_info,
  const sgx_report_data_t *report_data,
  sgx_report_t *report) {
   /* call EREPORT Instruction */
}
```

Listing 7: SGX Instruction Wrapper Functions

executions within an enclave. However, some SGX functionalities may depend on structures that are not yet supported by EnclaveFuzz or are even impractical to simulate. For instance, List 7 shows an example of a wrapper function from SGX SDK, which actually wraps an SGX hardware instruction that can be used to verify the authenticity of enclave. EnclaveFuzz is incapable and unnecessary to simulate such functions. However, the developer may use `sgx_verify_report` to verify the report and abort execution if failed. To make fuzz run continuously, EnclaveFuzz patches `sgx_verify_report` and always return true. That breaks the security mechanism of SGX and may lead to false positive results.

C. Related Work

1) Fuzzing enclaves: Several works have applied fuzzing to find bugs within enclaves. SGXFuzz [2] targets enclave binaries and applies a binary fuzzer [53] with snapshot supported [54], [55], which tries to recover the expected layout of input incrementally with the feedback information from page faults but may fail on some targets with multiple-sized parameters as aforementioned. Also, SGXFuzz can only detect specific types of vulnerabilities due to the lack of sanitizers for the enclave binary.

FuzzSGX [3] generates mutated host applications to test the enclaves, which incorporates program(C/C++) mutations within the fuzzing loop, leading to considerable overheads. The lack of untrusted memory input as well as SGX-specific sanitizer also limits its capability to uncover more bugs. More importantly, FuzzSGX still runs in simulation mode, which includes redundant routines detrimental to fuzzing efficiency.

2) Memory safety of enclaves: Memory safety is crucial for enclaves, and several works have shed light on this topic. DarkROP [56] presents the first memory corruption attack against SGX applications based on return-oriented programming (ROP). SGX-Shield [57] proposes an address space layout randomization (ASLR) mechanism for SGX applications. Guard's Dilemma [58] then shows that even ASLR protection can be compromised. Smashex [59] leverages SGX asynchronous exceptions, leading to enclave memory disclosures and ROP attacks. In order to automatically find vulnerabilities in enclaves, TeeRex [60] utilizes symbolic execution to analyze enclave binary code and checks symbolic states to report potential vulnerabilities. Coin Attacks [61] summarizes attack surfaces of an enclave and applies concolic execution to detect bugs with manually written policies. These solutions are incapable of analyzing large-scale programs due to intrinsic limitations of symbolic execution, including state explosion and unsolved constraints.

3) Fuzz other TEEs: TEEzz [62] focuses on fuzzing trusted application in ARM TrustZone. It employs dynamic binary instrumentation and captures trace information from multiple layers of interfaces to deduce the relationships between low-level interfaces and high-level APIs, as well as recovering the essential type information to achieve type-awareness. However, there's a caveat when applying this solution to SGX applications: if the host application doesn't engage all of the available `ECALLs` of an enclave, it may not capture representative run-time traces. This limitation is critical as attackers could potentially exploit these untouched `ECALLs` using a malicious host application.

SGXBOUNDS [26] uses high 32 bits to store boundary information and causes only 32-bit programs can run on 64-bit machines, and it can't detect dangling pointers.

As for the attack model, Iago [63] reveals it's hard to protect applications from malicious OS. Emilia [64] fuzz enclave OCALL to find possible Iago bugs.

VII. Conclusion

The present focus of fuzzing solutions is predominantly on testing enclave binaries, often bypassing the potential to harness information from the source code to enhance fuzzing effectiveness. EnclaveFuzz addresses this issue by systematically extracting interface structures from bridge code, thus automating the generation of more precise fuzzing harnesses.

In addition, EnclaveFuzz implements an optimized version of the original Intel SGX SDK to expedite fuzzing, contributing to the efficiency of the security testing procedure. EnclaveFuzz also integrates sanitizers to detect a wide spectrum of vulnerabilities, reinforcing its comprehensive approach to the testing of enclave binaries.

Acknowledgement

We sincerely thank all the anonymous reviewers for their valuable feedback. This work is supported in part by the National Key Research and Development Program of China (2021YFB2701000), the National Natural Science Foundation of China (61972224) and E3Z0201112.

References

[1] Signal. [Online]. Available: https://signal.org/en/

[2] T. Cloosters, J. Willbold, T. Holz, and L. Davi, "{SGXFuzz}: Efficiently synthesizing nested structures for {SGX} enclave fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3147–3164.

[3] A. Khan, M. Zou, K. Kim, D. Xu, A. Bianchi, and D. J. Tian, "Fuzzing sgx enclaves via host program mutations," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023, pp. 472–488.

[4] Intel, "Intel Software Guard Extensions SDK for Linux OS." [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html

[5] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 238–253.

[6] M. Zalewski, "American fuzzy lop," 2017.

[7] LLVM, "LLVM's SanitizerCoverage." [Online]. Available: https://clang.llvm.org/docs/SanitizerCoverage.html

[8] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[9] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata: Fuzzing with path aware taint analysis," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1–17.

[10] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "Greyone: Data flow sensitive fuzzing." in *USENIX Security Symposium*, 2020, pp. 2577–2594.

[11] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "Mopt: Optimized mutation scheduling for fuzzers." in *USENIX Security Symposium*, 2019, pp. 1949–1966.

[12] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[13] J. Choi, K. Kim, D. Lee, and S. K. Cha, "Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 677–693.

[14] Google, "Syzkaller." [Online]. Available: https://github.com/google/syzkaller

[15] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "{KSG}: Augmenting kernel fuzzing with system call specification generation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 351–366.

[16] LLVM, "libfuzzer - A Library For Coverage-guided Fuzz Testing." [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[17] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," in *USENIX Security symposium*. USENIX Association, 2017.

[18] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985.

[19] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 2271–2287.

[20] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[21] LLVM, "Undefined Behavior Sanitizer (UBSan)." [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[22] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*, 2009, pp. 62–71.

[23] Google, "AddressSanitizerLeakSanitizer." [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer

[24] LLVM, "Hardware-assisted AddressSanitizer." [Online]. Available: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html

[25] Google, "MemorySanitizer." [Online]. Available: https://github.com/google/sanitizers/wiki/MemorySanitizer

[26] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "Sgxbounds: Memory safety for shielded execution," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 205–221.

[27] G. M. Jurczyk and G. Coldwind, "Bochspwn: Identifying 0-days via system-wide memory access pattern analysis," *Black Hat USA Briefings (Black Hat USA)*, 2013.

[28] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 587–600.

[29] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How {Double-Fetch} situations turn into {Double-Fetch} vulnerabilities: A study of double fetches in the linux kernel," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1–16.

[30] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 661–678.

[31] A. Bhattacharyya, U. Tesic, and M. Payer, "Midas: Systematic kernel {TOCTTOU} protection," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 107–124.

[32] Intel, "Intel Software Guard Extensions SSL." [Online]. Available: https://github.com/intel/intel-sgx-ssl

[33] ——, "Intel SGX for Linux." [Online]. Available: https://github.com/intel/linux-sgx

[34] ——, "An End-to-End Distributed and Scalable Cloud KMS (Key Management System) built on top of Intel SGX enclave-based HSM (Hardware Security Module), aka eHSM." [Online]. Available: https://github.com/intel/ehsm

[35] SOFAEnclave, "A TEE programming framework based on trusted functions." [Online]. Available: https://github.com/SOFAEnclave/trusted-function-framework

[36] bytecodealliance, "WebAssembly Micro Runtime (WAMR)." [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime

[37] wolfSSL, "The wolfSSL library is a small, fast, portable implementation of TLS/SSL for embedded devices to the cloud. ." [Online]. Available: https://github.com/wolfSSL/wolfssl

[38] skalenetwork, "sgxwallet is the first-ever opensource high-performance hardware secure crypto wallet that is based on Intel SGX technology." [Online]. Available: https://github.com/skalenetwork/sgxwallet

[39] yerzhan7, "SQLite database inside a secure Intel SGX enclave (Linux)." [Online]. Available: https://github.com/yerzhan7/SGX_SQLite

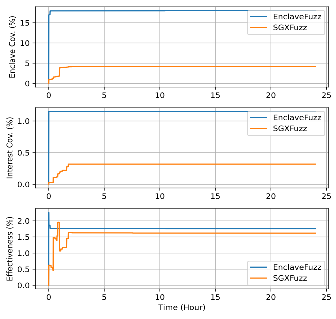
[40] P.-L. Aublin, F. Kelbert, D. O'Keffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch, "Talos: Secure and transparent tls termination inside sgx enclaves," 2017.

[41] bl4ck5un, "mbedtls-SGX: a SGX-friendly TLS stack (ported from mbedtls)." [Online]. Available: https://github.com/bl4ck5un/mbedtls-SGX
[42] asonnino, "Simple password-wallet application based on Intel SGX for linux." [Online]. Available: https://github.com/asonnino/sgx-wallet
[43] anonymous-xh, "SGX-Darknet: SGX compatible ML library." [Online]. Available: https://github.com/anonymous-xh/sgx-dnet
[44] ——, "Plinius: Secure ML model training with Intel SGX and PM for fault tolerance." [Online]. Available: https://github.com/anonymous-xh/plinius
[45] cBioLab, "A Practical Privacy-Preserving Data Analysis for Personal Genome by Intel SGX." [Online]. Available: https://github.com/cBioLab/BiORAM-SGX
[46] LedgerHQ, "BOLOS community enclave for SGX simulator." [Online]. Available: https://github.com/LedgerHQ/bolos-enclave
[47] rscosta, "SgxCryptoFile - App for Encrypting and Decrypting Files using Intel SGX." [Online]. Available: https://github.com/rscosta/SGXCryptoFile
[48] kudelskisecurity, "PoC of an SGX enclave performing symmetric reencryption." [Online]. Available: https://github.com/kudelskisecurity/sgx-reencrypt
[49] B. Beck, "LibreSSL–An OpenSSL replacement. The first 30 days, and where we go from here. BSDCAN 2014," 2014.
[50] LLVM, "Source-based Code Coverage." [Online]. Available: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html
[51] Microsoft, "Open Enclave SDK." [Online]. Available: https://github.com/openenclave/openenclave
[52] Google, "An open and flexible framework for developing enclave applications." [Online]. Available: https://github.com/google/asylo
[53] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.
[54] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *USENIX Security Symposium*, 2017.
[55] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo
[56] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 523–539.
[57] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in *NDSS*, 2017.
[58] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: efficient code-reuse attacks against intel {SGX}," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1213–1227.
[59] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, "Smashex: Smashing sgx enclaves using exceptions," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 779–793.
[60] T. Cloosters, M. Rodler, and L. Davi, "{TeeRex}: Discovery and exploitation of memory corruption vulnerabilities in {SGX} enclaves," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 841–858.
[61] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "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*, 2020, pp. 971–985.
[62] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer, "Teezz: Fuzzing trusted applications on cots android devices," in *2023 2023 IEEE Symposium on Security and Privacy (SP)(SP)*, 2023, pp. 220–235.
[63] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.
[64] R. Cui, L. Zhao, and D. Lie, "Emilia: Catching iago in legacy code." in *NDSS*, 2021.

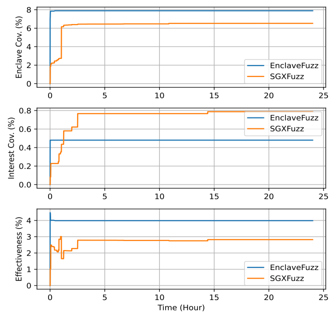
Appendix

A. Coverage over time

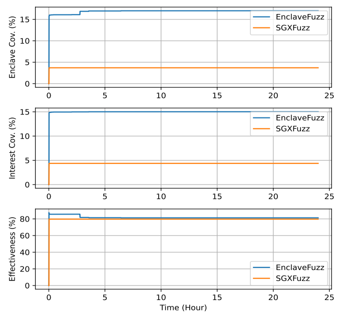
We collect enclave coverage, interesting coverage, and effectiveness over 24 hours for 20 enclaves, and demonstrate them in Figure A.6.



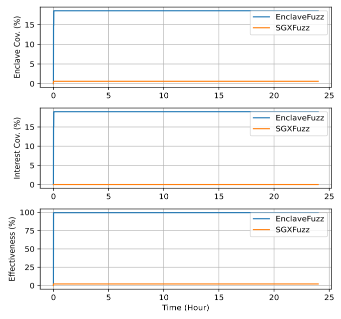
(a) BiORAM-SGX



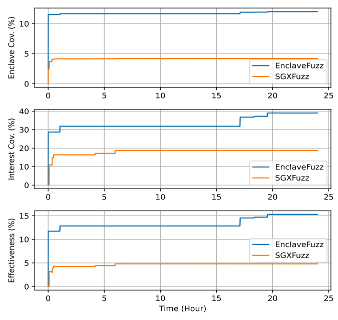
(b) bolos-enclave



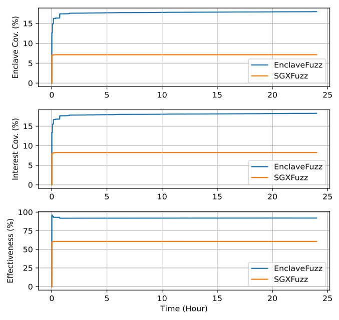
(c) ehsm



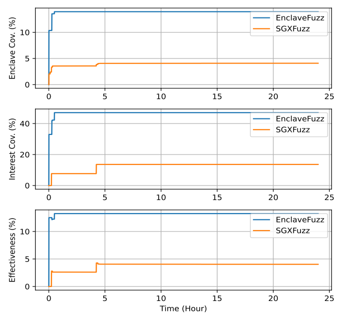
(d) intel-sgx-ssl



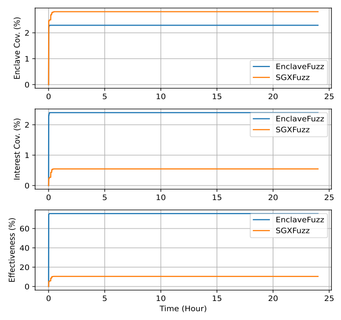
(e) AE LE



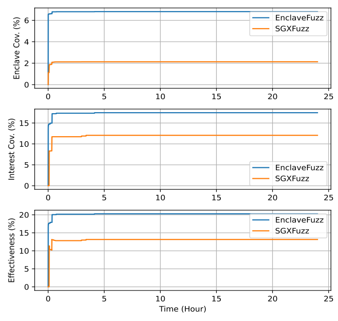
(f) mbedtls-SGX



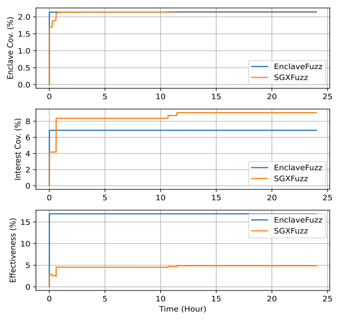
(g) AE PCE



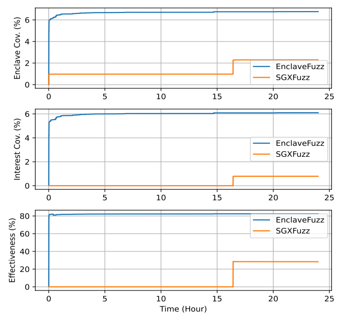
(h) plinius



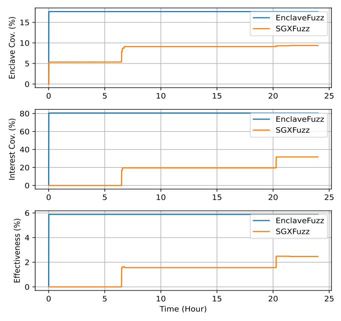
(i) AE PVE



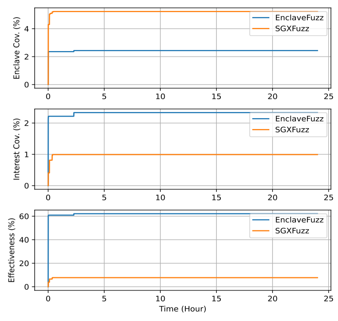
(j) AE QE



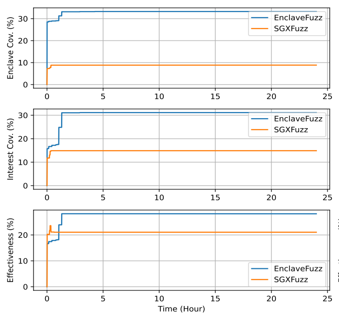
(k) SGX_SQLite



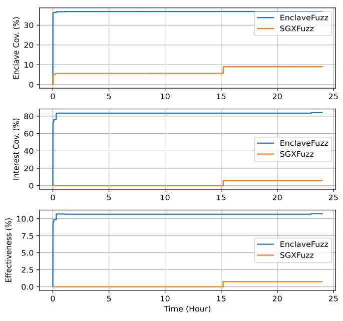
(l) SGXCryptoFile



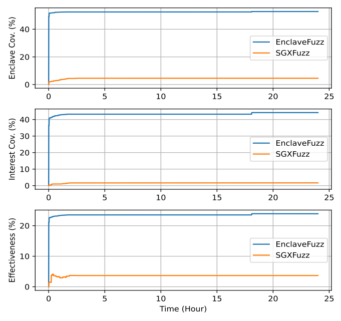
(m) sgx-dnet



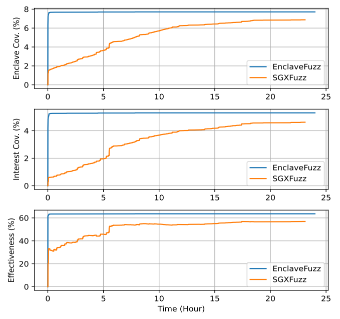
(n) sgx-reencrypt



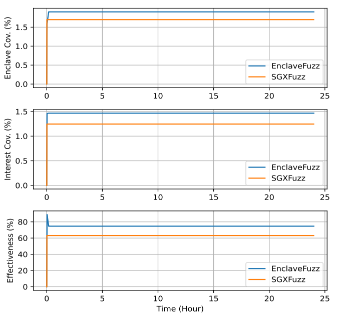
(o) sgx-wallet



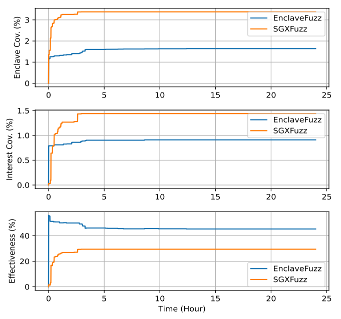
(p) sgxwallet



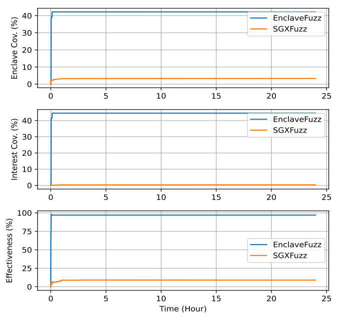
(q) TaLoS



(r) trusted-function-frame



(s) wasm-micro-runtime



(t) wolfssl

Fig. A.6: Code coverage Over Time