

Fuzzing SGX Enclaves via Host Program Mutations

Arslan Khan, Muqi Zou, Kyungtae Kim, Dongyan Xu, Antonio Bianchi, Dave (Jing) Tian
{khan253, zou116, kim1798, dxu, antoniob, daveti}@purdue.edu
Purdue University

Abstract—Intel Software Guard eXtension (SGX) is the cornerstone of Confidential Computing, enabling runtime code and data integrity and confidentiality via enclaves. Unfortunately, memory-unsafe and type-unsafe programming languages, such as C/C++, are commonly used to develop enclave implementations. As a result, a memory corruption or a data race within enclaves could lead to different attacks against the enclaves, such as Return-Of-Programming (ROP) and data leakage, breaking the hardware security guarantee provided by SGX. To automatically identify these issues in existing enclave implementations, in this paper, we propose FUZZSGX, an input and program mutation-based fuzzer for Intel SGX enclave implementations. FUZZSGX provides an enclave fuzzing runtime, FUZZSGX RUNTIME, a drop-in library for Intel SGX SDK, enabling code coverage and sanitization within enclaves. To explore the host app-enclave boundary, FUZZSGX conducts static analysis and symbolic execution on existing host apps and enclave implementations to generate promising fuzzing programs, fuzzing both ECALLs and OCALLs. We evaluate FUZZSGX using 30 popular SGX applications and enclave implementations and find 93 bugs among these SGX projects, including data races, null pointer dereferences, out-of-bound accesses, division-by-zero, etc. FUZZSGX achieves 3.2x higher code coverage and finds 48.2% more bugs by directly targeting the host app-enclave boundary by using program mutations, compared to state-of-the-art fuzzers.

1. Introduction

Intel Software Guard eXtension (SGX) is a microcode and hardware extension to Intel CPUs after Skylake [1] providing runtime memory integrity and confidentiality via CPU-protected memory regions called enclaves. Traditional programs can be partitioned into a trusted part, *Enclave* and an untrusted part, *Host App*. While the host app can be compromised and goes rogue, the enclave is expected to be secure. Since its introduction, SGX has been used to secure data analytics and machine learning applications [2], [3], [4], databases [5], [6], [7], [8], [9], filesystems [10], [11], multi-party computation [12], and others [13], [14], [15]. SGX has become the cornerstone of Confidential Computing [16], [17] allowing users to enjoy the power of cloud computing without trusting cloud vendors other than Intel CPUs.

Unfortunately, SGX applications and enclave implementations are commonly written in memory-unsafe and type-unsafe programming languages such as C/C++, partially due to the wide adoption of the Intel SGX SDK [18] and C/C++ in legacy applications. Furthermore, SGX

applications can employ multi-threading to achieve the benefits of multiple cores, which leaves SGX enclaves prone to race conditions, as C/C++ are not inherently thread-safe languages. The situation is worsened by the strong threat model of enclaves; i.e., only the enclave is considered the trusted computing base (TCB) whereas both the Operating System (OS) and the host app can be potentially malicious. As a result, enclaves need to secure themselves against every possible input from outside the enclaves. SGX developers often misplace the trust in the host apps and sanitize inputs inside the host app, instead of the enclave, leaving enclaves susceptible to malicious inputs. A memory corruption bug or a data race within enclaves could lead to different attacks against enclaves, including Return-Oriented-Programming (ROP) [19], [20] and secret data leakage [21], [22], defeating the security guarantees enforced by SGX.

To detect these vulnerabilities within enclave implementations, state-of-the-art solutions use different program analysis techniques and focus on different aspects of SGX applications. TeeRex [23] uses symbolic execution to analyze binary enclaves but faces the path explosion when investigating complex enclaves. Emilia [24] leverages fuzzing to detect Iago vulnerabilities [25] targeting the syscall-to-enclave boundary. However, Emilia fails to find memory corruption vulnerabilities within enclaves.

To find memory corruption vulnerabilities, SGXFuzz [26] uses code coverage-guided fuzzing SGX enclave binaries built for the hardware mode. However, as advised by Intel, binary enclaves should run in the hardware mode and exercise remote attestation to ensure that the enclave is running on a genuine Intel processor [27]. Due to this limitation, only the subset of enclaves running without remote attestation can be used for enclave binaries (See section 7). Moreover, due to the nature of binary fuzzing, SGXFuzz could not leverage source-level instrumentation and uses methods such as Intel Process Trace (PT) [28] to extract the code coverage from inside the enclave, resulting in up to a 90% performance degradation in fuzzing throughput [29] and very limited program sanitizations [30], [31]. Lastly, SGXFuzz does not support simultaneously fuzzing host apps and enclaves to fuzz complete SGX applications, causing it to miss bugs resulting from the integration of the host app and enclaves.

Based on our observations, we note that we still need a comprehensive fuzzing solution for Intel SGX applications and propose FUZZSGX, a coverage-guided input and program mutation-based fuzzer for Intel SGX enclave implementations. FUZZSGX provides FUZZSGX RUNTIME, a library OS (libOS) for the Intel SGX SDK enabling high-throughput coverage feedback, address sanitization, and

thread sanitization automatically. FUZZSGX fuzzes both user-host app boundary and host app-enclave boundary by conducting static analyses and symbolic execution on the host app and enclave implementations to generate programs directly fuzzing ECALLs and OCALLs of an enclave.

We evaluate FUZZSGX on 30 SGX applications and enclave implementations and find 93 previously unknown bugs within these SGX projects. In summary, the contributions of this paper are as follows:

- We design and implement FUZZSGX RUNTIME, a novel mechanism to extract code coverage from enclaves using a specialized libOS for Intel SGX SDK, enabling code coverage reporting, address sanitization, and thread sanitization within enclaves.
- We design and implement FUZZSGX, an orchestrated SGX fuzzer targeting enclave implementations. FUZZSGX uses static analysis and symbolic execution to reason about internal dependencies of ECALLs and their arguments to generate promising host fuzzing programs.
- We evaluate FUZZSGX using 30 SGX applications and enclave implementations and find 93 previously unknown bugs among these projects. FUZZSGX achieves 3.2x higher code coverage and finds 48.2% more crashes compared to the state-of-the-art fuzzers, such as SGXFuzz [26].

We have reported all our findings to the respective parties. The source code for FUZZSGX is available at: <https://github.com/purseclab/FuzzSGX>.

2. Background

In this section, we first introduce the high-level technical concepts of Intel SGX and the Intel SGX SDK. Then, we survey existing vulnerability discovery solutions for SGX enclave implementations to motivate our work. Finally, we present the security model we consider in the paper.

2.1. Intel SGX

Intel SGX [32] enables userspace software to create isolated execution environments (i.e. enclaves), running in a dedicated processor mode called SGX mode. These enclaves are created from the Enclave Page Cache (EPC), a physical memory region reserved by the CPU during system boot-up. The code and data inside an enclave are shielded from the rest of the system, including the application creating that enclave, the operating system, and any other software, regardless of the privilege level in which it runs. To use SGX, userspace software, which we will refer to as the *"Host App"*, asks the kernel to create an enclave within its own address space with provided code and data.

Figure 1 shows the memory map of an SGX application with its enclave loaded within its address space. The enclave is compiled as a shared object signed by the corresponding vendor. The Intel SGX kernel driver [18] uses the EADD instruction to add this shared object to the enclave memory. Furthermore, it measures the enclave code and data for checking their integrity using the EEXTEND instruction. EEXTEND stores this intermediate

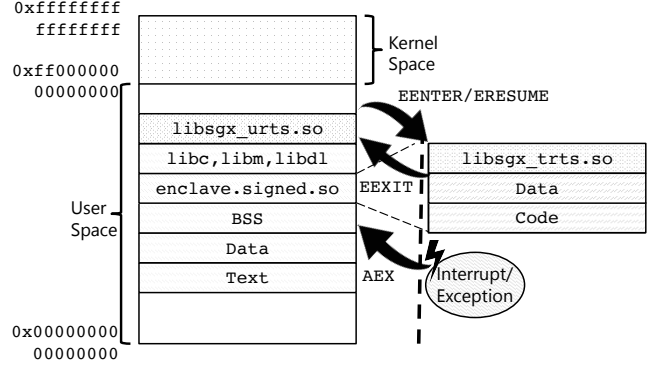


Figure 1: Address layout of an SGX application. The host app can load an enclave into its address space using `libsgx_urts.so` and the SGX kernel driver. Once the enclave is initialized, the host app can only enter the enclave using the `EENTER` instruction. Similarly, all exits from enclaves must be explicit, using the `EEXIT` instruction.

measurement in the `MRENCLAVE` register. Finally, the driver executes the `EINIT` instruction completing the initialization process. `EINIT` finalizes the verification process by comparing the signature in the `SIGSTRUCT` certificate to the measurement in the `MRENCLAVE` register. If the enclave passes its integrity checks, the enclave's init status is set to true, making it available for use by the host app.

To enter the enclave, the host app needs to execute the `EENTER` SGX instruction, which switches the CPU from user mode to enclave mode. Inside an enclave, only ring-3 instructions are allowed, therefore to achieve any ring-0 functionality enclaves must collaborate with host apps. To exit the enclave synchronously, the enclave needs to execute the `EEXIT` instruction, switching the CPU back to user mode. While the CPU is in enclave mode, any interrupts and exceptions result in Asynchronous Enclave Exits (AEX). The host app can resume the interrupted enclave execution by reentering the enclave via the `ERESUME` instruction.

To abstract the low-level SGX mechanics, Intel SGX SDK [18] provides a mechanism to define the interface between the enclave and the host app using a C-like language called Enclave Description Language (EDL). The EDL file of an enclave defines both entry functions from the host app to the enclave (ECALLs) and functions provided by the host app but callable by enclave (OCALLs). Furthermore, the SDK provides a runtime for SGX applications to provide common functionalities, such as transition routines between the host app and the enclave, cryptographic functions, etc. Due to the partitioned nature of SGX applications, the runtime is also split into two libraries: the Untrusted Runtime System (URTS) and the Trusted Runtime System (TRTS). The URTS and the TRTS are linked to the app and enclave respectively as shown in Figure 1. The URTS provides app functionality to manage the enclave lifecycle and utilize enclave functionalities, whereas the TRTS is used to validate the input from the URTS and provides a trusted C library to aid in enclave development.

2.2. Security Model

The trust model of an enclave implementation includes Intel SGX-enabled CPUs and the code running within the enclave. Depending on the enclave development environment, code within the enclave contains both the trusted runtimes provided by an SDK and other trusted libraries linked together. For instance, a crypto-related enclave implementation developed using the Intel SGX SDK consists of not only the business logic but also the TRTS and the SGX crypto library. Intel SGX also provides provisions to verify if the desired enclave implementation is running on a genuine Intel SGX CPU.

The threat model of SGX assumes malice from all the possible hardware components other than the CPU and all the available firmware and software other than the code within the enclave, including e.g., host app, OS, hypervisor, etc. While we do not consider Denial-of-Service (DoS) attacks from the system software, e.g., the OS refused to load an enclave, our threat model includes DoS attacks against enclaves during runtime via memory corruptions. Moreover, we consider various attacks that can be achieved via memory corruption within enclaves, such as ROP and data leakage.

3. Enclave Fuzzing Challenges

Fuzzers work by mutating program state (input and code) and inferring any changes seen in the code execution paths to quantify if the mutation was helpful. However, the SGX model poses the following Runtime Challenges (RC):

- **RC1: Limited Resources.** SGX enclaves provide a limited amount of resources. Most SGX implementations only provide 128MB of EPC which is shared globally among all enclaves in the system. Since fuzzers run many instances of the target programs, a high-throughput fuzzer would quickly run out of this memory budget.
- **RC2: Unavailable OS Services.** Enclaves are unable to utilize any operating system services such as system calls. As existing fuzzing solutions, like AFL [33], are highly dependent on OS services, they cannot be applied to SGX out of the box.
- **RC3: No Code Coverage Export.** As the primary purpose of SGX application partitioning is to hide information, existing fuzzing solutions are not able to see any updates from inside the enclave.
- **RC4: Missing Sanitizers.** Fuzzers rely on program sanitization to find common spatial and temporal bugs. Due to RC1 and RC2, existing sanitizers cannot work within enclaves.

A naive solution to these challenges could be the strategy adopted by Intel SGX-Fuzzer [34], i.e. using the fuzzer and program sanitization on the host application while considering the enclave as a black box. Such configuration can effectively find bugs in the host apps, however, this approach fails at effectively fuzzing enclaves and struggles to find bugs in the enclave as the fuzzer is not able to learn the code execution changes inside the enclave. While bugs in host apps are a threat, the SGX threat model already assumes the host app is untrusted. Hence, it makes more

sense to fuzz the app-enclave interface directly, instead of the app itself. Therefore, a fuzzer unaware of the SGX threat model may spend a lot of time exploring the host app whereas the real target is the enclave. Hence, efficient fuzzing becomes highly dependent on the host app. This cohesion between the host app and enclave can result in the following challenges for fuzzing:

- **FC1: Deep Code Paths.** ECALLs could be hidden under the deep paths of the host app code. As a result, existing fuzzers could spend hours of mutation to find inputs that reach ECALL invocation sites.
- **FC2: Unused ECALLs.** As mentioned before enclaves can provide multiple ECALLs, however, a host app might not require all of these ECALLs, making it impossible for fuzzers to find bugs in the unused ECALLs.
- **FC3: Limited ECALL Input and OCALL Output Coverage.** To uncover Iago [25] and COIN [35] vulnerabilities, fuzzers should be able to mutate the ECALL inputs and OCALL outputs. However, a host app might sanitize the inputs in such a manner that it is not possible to invoke ECALLs with arbitrary inputs by merely mutating inputs at the user interface, i.e. the command line arguments or standard input. Similarly, host apps can have similar sanitization on the output of OCALLs. Although this sanitization works for the specific host app and enclave combination, this misplaced sanitization goes against the SGX threat model. Enclaves are entirely responsible for input validation for all inputs they receive, as these inputs come from the untrusted runtime. Hence, such host app sanitization hinders the bug-finding capabilities of existing fuzzers.
- **FC4: Unavailable Host Apps.** When enclave implementations are used as secure libraries. Developers might not provide a host app at all and provide enclaves only to be used as a secure library rendering normal fuzzer futile without a host program. Furthermore, a host app could be incompatible with C/C++ program fuzzers. For instance, Signal Enclave [36] is written in C/C++ while the host app is written in Java, making it incompatible with C/C++ fuzzers.

4. Design

In this section, we discuss the design of FUZZSGX RUNTIME and show how it can be used to extract coverage information and enable sanitization from within enclaves. Next, we consider the design of FUZZSGX and show how FUZZSGX can tackle the fuzzing challenges described in Section 3.

4.1. FUZZSGX RUNTIME.

Coverage Reporting Code coverage-guided fuzzers extract the code coverage information by assigning distinct identifiers to basic blocks and instrumenting each basic block, such that every jump between basic blocks is reported to the fuzzer. This instrumented code uses OS services such as IPC to communicate this information to the fuzzer. Furthermore, to help find bugs such as stack overflows, buffer overflows, etc., target programs

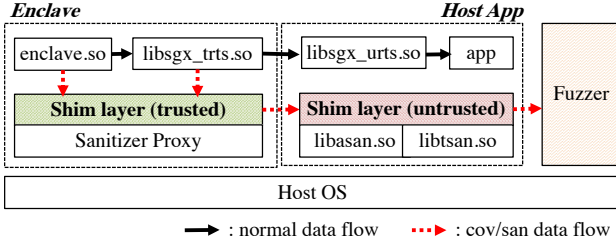


Figure 2: FUZZSGX RUNTIME architecture. The enclave is linked with the trusted part of the shim layer, while the host app is linked with the untrusted part.

are commonly instrumented with sanitizing code, which also depends on OS services. Since these mechanisms are not available in enclaves, existing approaches to code coverage measurement and bug detection do not work with SGX programs.

To overcome this limitation, a naive solution could be to use existing libOS [37], [38] approaches. However, these projects are designed to run userspace applications inside SGX enclaves, hence, they do not work well with SGX-aware applications. Furthermore, these frameworks result in high overhead, preventing high throughput execution of test cases. Furthermore, this solution does not help with *RC1: Limited Resources*. The high overhead and limited resource availability in hardware mode makes fuzzing in hardware mode using existing libOS infeasible.

To tackle this challenge, we propose using the simulation mode for fuzzing SGX applications, hence tackling *RC1: Limited Resources*. However, the simulation mode does not solve *RC2: Unavailable OS Services*, as the simulation mode only emulates SGX leaf instructions and structures without impacting enclave implementations. As a result, enclaves built for simulation mode still lack OS services. Therefore, to emulate OS services, we implement a *shim layer* that allows code coverage collection and vulnerability detection within enclaves by resolving all of the dependencies, such as system calls, required by the instrumented code with minimal overhead. As shown in Figure 2, the shim layer is split into trusted and untrusted components. The trusted component links with the enclave and the untrusted part links with the host app. The trusted part emulates any OS services required by the sanitization and code coverage. These services include Inter-process communications (IPC), memory management, file handling, etc. To get code coverage from all enclave components (i.e., *libsgx_trts.so* and *enclave.so*), we place the shim layer right at the bottom of the enclave and the host app. Thanks to the division of the shim layer into the trusted and the untrusted part, SGX-aware applications can use it without any modification in the source code of either the enclave or the host app. The shim layer emulates essential OS services required for fuzzing and sanitization, with a minimalist overhead, hence meeting *RC2: Unavailable OS Services* challenge. Furthermore, current fuzzers, such as AFL/AFL++, can extract code coverage from inside enclaves using the shim layer, hence bypassing *RC3: No Code Coverage Export*.

Sanitizers. The shim layer can emulate OS services inside enclaves. However, even with these services, current sani-

tizers do not work with the partitioned SGX programs. For instance, Thread Sanitizer (TSAN) [39] does not support sanitization in statically linked libraries [40], whereas enclaves are distributed as statically linked libraries making them incompatible with TSAN. To tackle this problem, we modify the address and thread sanitizer to make them aware of SGX programs’ partitions.

Address Sanitizer (ASAN): ASAN finds spatial bugs by tracking memory accesses and keeping the access metadata in shadow memory [30]. We use the shim layer to export the updates to the shadow memory and bug reports. Furthermore, to enable direct updates of the shadow memory, we use the simulation mode of Intel SGX SDK. For ASAN, statically linked enclaves can utilize the initialization done by the host app, hence the initialization routines run inside the enclave without any modifications.

Thread Sanitizer (TSAN): TSAN finds temporal bugs by conducting a Happens-Before [39] analysis. TSAN requires information about memory accesses. Similar to ASAN, we use the shim layer to export this information from within the enclave. However, unlike ASAN, TSAN’s initialization does not take care of statically-linked libraries, and re-initializing the sanitizer from a statically-linked enclave overwrites the previous initialization, crashing the sanitizer. To adapt TSAN for enclaves we need to either modify TSAN to have a different dynamically allocated metadata or modify TSAN to merge the operations from the static enclaves to the host app metadata. The first approach does not scale, as for each statically linked library we would have to allocate a new set of metadata and reinitialize the sanitizer each time a new library is loaded. We used the second design for our modified TSAN. Using the customized SGX-aware sanitizers we meet *RC4: Missing Sanitizers*.

4.2. FUZZSGX

To tackle the challenges mentioned in Section 3, we design FUZZSGX as an enclave interface-aware fuzzing suite. FUZZSGX uses traditional input mutation techniques for fuzzing host apps. Moreover, FUZZSGX employs program mutations to directly fuzz the host app-enclave interface. In a nutshell, FUZZSGX uses the host app and enclave source to synthesize custom-generated host apps to connect with the target enclave and it feeds inputs targeting the enclave functions through the generated host apps. At the same time, FUZZSGX employs program mutation to create mutated host apps that can efficiently fuzz the host app-enclave interface. To create these mutated host apps, FUZZSGX analyzes the source code of both the host app and the enclave. Moreover, the source is used to instrument the enclave to enable both code coverage feedback and the usage of different sanitizers. Figure 3 shows the fuzzing pipeline for FUZZSGX. We discuss each phase of the pipeline in the following subsections.

Program Mutation. The fuzzing challenges (FC) presented in Section 3 cannot be tackled using only input mutation. For instance, if a host app does not invoke an ECALL, any duration of input mutation will not be able to invoke this ECALL. To this end, FUZZSGX generates custom host apps that exploit the SGX threat model to find vulnerabilities where current fuzzers fail. For program synthesis, FUZZSGX identifies enclave interfaces and input

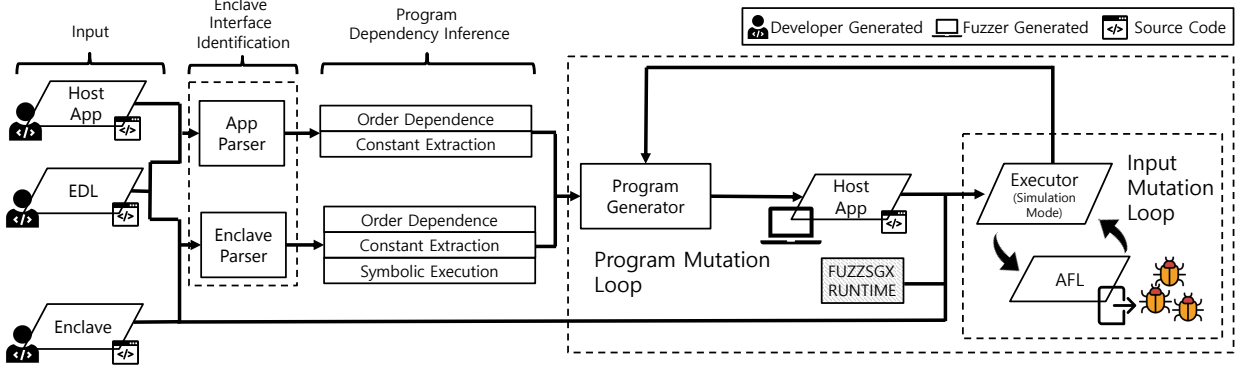


Figure 3: FUZZSGX pipeline. Using the enclave metadata and code FUZZSGX learns about the program to effectively generate new test programs. FUZZSGX fuzzes the newly generated test programs using input mutation.

formats using static and symbolic program analysis. First, it parses all interfaces to an enclave, such as ECALLs and OCALLs, using the EDL and the enclave source files (in the *Enclave Interface Identification* phase). Then, it reasons about call dependencies for enclave entry functions, as well as input values leading to in-depth enclave code paths (*Enclave Dependency Inference*). Based on this information, it generates test cases for fuzzing (*Program Generation*). Lastly, FUZZSGX fuzzes the newly generated programs in a continuous loop (*Main Fuzz Loop*).

Enclave Interface Identification. Given an SGX enclave, FUZZSGX needs to figure out its public interfaces. To achieve this, FUZZSGX uses the *Interface Parser*, which understands basic information concerning the enclave code. Specifically, the interface parser parses the EDL files, extracting ECALL/OCALL functions and EDL-specific annotations, such as interface information about arguments, specified within trusted and untrusted sections, respectively. Furthermore, FUZZSGX also creates a database of symbols from source code which is used to find any type information in the EDL files and mutate the OCALL definitions in place. The custom parsing extracts all interfaces provided by an enclave, regardless of any host app dependency, hence, overcoming the *FC2:Unused ECALLs* challenge. The interface parser saves the prototypes and the type information as an Abstract Syntax Tree (AST). FUZZSGX uses this information in the following program analysis, program generation, and input mutation phases.

Program Dependency Inference. ECALLs can have multiple dependencies among each other. This phase is responsible for finding these dependencies between different ECALLs. Existing work [41], [42] has tried to infer the dependencies between different APIs for a library by parsing users of that particular library. However, unlike normal libraries, enclaves rarely have more than one host app. In some cases, some enclaves do not provide a host app making existing inference techniques inapplicable. To tackle these challenges, FUZZSGX uses static analysis and symbolic execution to extract this information. These analyses are conducted on both host apps and enclaves, however, host apps are considered optional. FUZZSGX augments the analysis results with the EDL information for generating SGX applications without a host app.

Order Dependence Analysis: One possible dependency among ECALLs is the Happens-Before dependency (i.e. "ECALL A must be called before ECALL B"). Figure 4

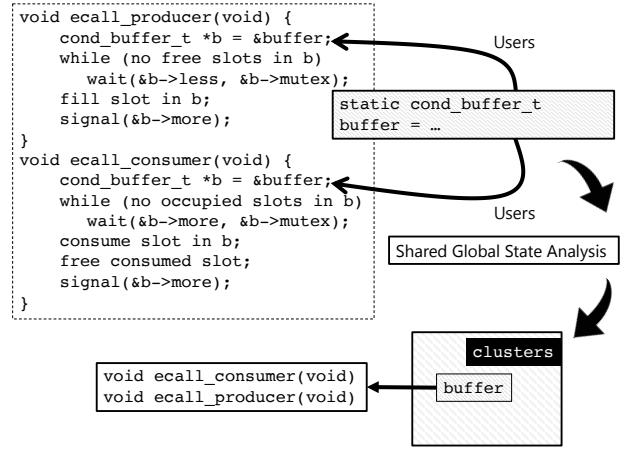


Figure 4: Order Dependence on enclaves: We find common users of variables used as arguments for function calls. `ecall_consumer` and `ecall_producer` are clustered together because of the shared variable `buffer`

shows an example of one such dependency. The producer function must be called before the consumer function. Otherwise, the program will indefinitely block inside the consumer function. FUZZSGX uses the *order dependence analysis* on enclave code to infer such dependencies, as shown in Algorithm 1.

The pass works depending on the availability of the host application. If a host application is available, this pass tracks different program values (variables) used in the host app. If the host app issues different ECALLs from the same context in the host app, this pass finds whether there is some dependency over return value or input arguments among those ECALLs. Specifically, this pass goes through all the users of a value. If a value is used in multiple function calls, we register those functions as an order dependency chain. Figure 5 shows partial results of running this pass on a host app. `str3` is used as the second argument and its length is used as the third argument. Running the pass, gives us a chain of function calls, hinting the fuzzer that the third argument of `ecall_pointer_size` should be the return value of `strlen` applied to the second argument.

In case the host app is missing, the pass goes through all the global state, such as global variables, and find the users of the global state. If FUZZSGX finds any ECALLs in the users of the global state, it adds them to the user for

Algorithm 1: Order Dependence Analysis

```

Result: DepChain
if  $\exists$  hostApp then
  ECALL  $\leftarrow$  PUBLIC_ECALLS
  DepChain  $\leftarrow \emptyset$ 
  foreach funcs  $\in$  FUNCTIONS do
    foreach call  $\in$  func.calls do
      if call.callee  $\in$  ECALL then
        temp.insert([return.def, -1, call])
        foreach arg  $\in$  call do
          temp.insert([arg.def, arg.Position, call])
        end
        if count(temp[value, *, *]) > 1 then
          foreach [arg, call]  $\in$  temp[value, *, *] do
            DepChain[value]  $\leftarrow$  [arg.Position, call]
          end
        end
      end
    end
  end
else
  clusters  $\leftarrow \emptyset$ ;
  foreach var  $\in$  GLOBAL_VARIABLES do
    if size(var.users) > 1 then
      foreach user  $\in$  var.users do
        clusters[var].insert(user)
      end
    end
    foreach [cluster, var]  $\in$  clusters do
      permutations  $\leftarrow$  permutation(cluster);
      bestCoverage  $\leftarrow$  0;
      foreach order  $\in$  permutations do
        currentCoverage  $\leftarrow$  run(order);
        if currentCoverage > bestCoverage then
          clusters[var]  $\leftarrow$  order;
          bestCoverage  $\leftarrow$  currentCoverage;
        end
      end
    end
    DepChain  $\leftarrow$  bestCoverage
  end

```

```

char str3[] = "1234567890";
ret = ecall_pointer_size(global_eid, (void*)str3, strlen(str3));
if (ret != SGX_SUCCESS)
  abort();

```

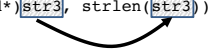


Figure 5: Order dependence analysis on host apps. The pass goes through the host app to extract the dependency between different arguments. `str3` registers a dependency between argument 1 and 3 for `ecall_pointer_size` function.

these global states. We call a set of users for a global state cluster. While the clusters show a possible dependency among different ECALLs, they do not tell anything about the correct order. To infer the correct order, FUZZSGX uses the code coverage as a metric. The observation here is that a misplaced ECALL will either block or fail, resulting in lower code coverage. Motivated by this observation, FUZZSGX evaluates all permutations and picks the one with the highest coverage score as the inferred correct order. These clusters are used to generate programs with different permutations, while the code coverage is dynamically measured. For Figure 4, the pass results show that these two functions have a shared global variable buffer, indicating a possible dependency between the two functions.

Constant Extraction: This pass parses the initialization statements and conditionals inside enclaves and the host app code to extract all constant literals. We supply these arguments as a dictionary for input mutation. This dictionary helps to bypass hard-coded constraints to find new paths.

```

int bBckHMeFDd_1;
temp = (char *)&bBckHMeFDd_1;

int* kXGyTnqXvk_1 = &bBckHMeFDd_1;
long int KMgIidYzwI_2;
temp = (char *)&KMgIidYzwI_2;
fgets(temp,8, stdin);

char IgvAVOkRvU_3;
temp = (char *)&IgvAVOkRvU_3;
fgets(temp,1, stdin);

const char* VTxDUTIuaW_3 = &IgvAVOkRvU_3;
ret = enc_wolfSSL_CTX_set_cipher_list(global_eid,
  kXGyTnqXvk_1, KMgIidYzwI_2, VTxDUTIuaW_3);

```

Listing 1: Code Snippet generated for Wolf SSL Enclave. The generated program expects inputs directly using input mutation.

Program Generation FUZZSGX generates new host apps in the program generation phase. The generated programs take input from the input fuzzer and directly feed it to the enclave. The program generator requires a template program that consists of the boilerplate code for enclave initialization and any project-specific setup. The boilerplate code ensures that the program generator can issue ECALLs, without knowing any semantics about the initialization of the enclave. In most cases, the template file simply creates the target enclave. FUZZSGX modifies the template program to generate code for fuzzing ECALLs and OCALLs. For ECALLs, FUZZSGX adds code for mutating inputs to the enclave and randomly reorders their invocation. Similarly, for OCALLs, FUZZSGX generates code for mutating the return value of OCALLs and may insert code for calling nested ECALLs based on an unfair coin toss¹. The nested ECALLs invocations, if any, are also randomly reordered in each fuzzing iteration.

During program generation, FUZZSGX follows the information inferred in the previous stages. For instance, based on the results of shared global state analysis, FUZZSGX emits the code for all ECALLs in a dependency cluster if one of the ECALL is present in the generated test case. Listing 1 shows a snippet of generated code. Using this mechanism, we can tackle *FC4:Unavailable Host Apps*.

Seed Value Generation With the newly generated program, FUZZSGX can start the fuzzing campaign. However, fuzzers usually require seed inputs as the starting point for better mutation. A simple design is starting the fuzzer without any seed inputs. However, existing work [43] shows that fuzzing is highly dependent on initial seed inputs. For instance, without input format information, random mutations struggle to bypass comparisons against magic numbers. Therefore, to get good seed input values for input mutation, FUZZSGX provides two mechanisms to generate seed inputs for an enclave: 1) Symbolic Execution and 2) Mutation-based inference.

- **Symbolic Execution:** In this configuration we use symbolic execution on the enclave code and use the interfaces identified by the *Interface Parser* to symbolize input arguments to ECALLs. To tackle the path explosion problem, instead of symbolically executing the SGX SDK libraries, FUZZSGX concretely

1. The unfairness of the coin is a configurable parameter.

executes them and skips functions that do not alter the final state of the ECALL. In this way, FUZZSGX can reduce the number of symbolic states. Furthermore, FUZZSGX limits the symbolic execution to a predefined step count. The step count can be configured according to the target project. This design enables FUZZSGX to fuzz projects where symbolic execution does not scale.

- *Mutation-based Inference:* Existing work proposes different mutation strategies to find interesting inputs without prior information about the program. GRIMOIRE [44] infers the structure of the input based on code coverage feedback, whereas REDQUEEN [45] establishes input to program state correspondence by conducting static analysis. In this configuration, FUZZSGX generates programs and fuzzes them with REDQUEEN and GRIMOIRE mutation strategies without providing initial seeds.

We compare the efficacy of these two approaches in terms of input seed generation in Section 6.

Main Fuzzing Loop FUZZSGX mutates the target program in a continuous loop, setting up a test harness for each generated program for input fuzzing. FUZZSGX works in two configurations to explore ECALL order:

Exhaustive Configuration: If the number of ECALLs is less than a configurable threshold number, FUZZSGX generates all possible permutations of ECALL sequences. These programs can be fuzzed in parallel using input mutation indefinitely.

Iterative Configuration: For enclaves with a large number of ECALLs, it might be infeasible to run all of the permutations in parallel. In this configuration, FUZZSGX randomly mutates the ECALL order in a continuous loop and it can be configured to terminate each iteration after one of the conditions: 1) A predefined amount of time has elapsed. 2) No new path is seen for a predefined amount of time or 3) A predefined number of input mutation cycles have been executed.

By directly fuzzing the host-enclave boundary and using the information extracted via the various interfaces, we can tackle *FC1:Deep Code Paths* and *FC3:Limited ECALL Input and OCALL Output Coverage*.

Input Mutation. Once the host app is generated using program generation, FUZZSGX sets up a test harness to drive the input mutation process. For input mutation, FUZZSGX needs to instrument enclave code so that it can export code coverage and sanitize programs to assist in finding bugs. This instrumented code normally relies on OS services, however, the programming environment inside SGX enclaves is severely limited which restricts users from using the instrumented code. FUZZSGX utilizes FUZZSGX RUNTIME to extract the code coverage information from within enclaves.

5. Implementation

We implement FUZZSGX as a modular system. For FUZZSGX RUNTIME, the shim layer is implemented as a shared library for the Intel SGX SDK. As described in Section 4, the library is split into a trusted (*libsgx_tsgxfuzz*) and an untrusted part (*libsgx_usgxfuzz*). *libsgx_tsgxfuzz* consists of 1.1K SLOC, whereas *libsgx_usgxfuzz*

Component	SLOC
FUZZSGX RUNTIME	1.5 K
FUZZSGX	2.1K

TABLE 1: SLOC for different components of FUZZSGX.

consists of 185 SLOC. We tried to keep the modifications to address sanitizer and thread sanitizer to a minimum and accommodated the changes as much as we could inside the shim layer. To this end, we were able to run both sanitizers with minor modifications in the Intel SGX SDK. For address sanitizer, all of the changes were contained in the SDK and the shim layer whereas for thread sanitizer we added merely 74 SLOC to the project.

The main implementation of FUZZSGX is a Python program consisting of 1.2K SLOC. This program is responsible for *Program Dependency Inference*, *Program Generation*, and orchestrating the *Main Fuzz Loop*. The analyses for *Program Dependency Inference* are implemented as different LLVM passes, including the Order Dependence Analysis and Constant Extraction. These passes consist of 875 SLOC. Since the Intel SGX SDK does not support LLVM, we convert the application and the enclave to separate LLVM bitcode files. We then run different LLVM passes on the application bitcode file and the enclave bitcode file.

For symbolic execution, we use angr [46], a well-known binary analysis and symbolic execution framework. Furthermore, to simulate the SGX SDK library functions (e.g., `sgx_is_within_encl()`), we utilize angr’s *symprocedures*. Table 1 summarizes the SLOC of each component.

6. Evaluation

To evaluate FUZZSGX, we conducted our experiments on an octa-core Intel(R) Core(TM) i7-7700 CPU with 16GiB of physical memory. Our data set includes the top 50 SGX projects on GitHub based on the project star rating. Out of these projects, we fuzzed 26 projects compatible with FUZZSGX. The projects left out were either outdated (i.e. they did not work with the latest SGX SDK version) or had compilation errors. Furthermore, we pick projects from existing SGX vulnerability discovery papers such as TeeREX [23] and Emilia [24]. Lastly, since libOS projects, such as Graphene-SGX [37] and SGX-LKL [47], do not use the Intel SGX SDK EDL files to describe the interface between enclave and host app, we manually created the EDL and fuzz them using program mutation only. Overall, we fuzzed a total of 30 projects as described in Appendix A.

The rest of this section is structured as follows: Section 6.1 summarizes the bugs found by FUZZSGX. Section 6.2 evaluates the overhead incurred by FUZZSGX RUNTIME. Section 6.3 evaluates the seed generation mechanisms provided by FUZZSGX. Section 6.4 evaluates FUZZSGX in terms of its bug-detection capability and efficacy in fuzzing SGX programs. Section 6.5 evaluates efficacy of program mutation generated by FUZZSGX. Section 6.6 discusses case studies regarding the vulnerabilities found by FUZZSGX.

Project	Type	Revision	Bugs
webasm	App	2281ac9	30
Graphene	LibOS	v1.0.1/v1.1	2
SGX-LKL	LibOS	c8cb0b8	6
secure-analytics-sgx	App	43576a2	37
linux-sgx	SDK	2.11	2
sgx-kmeans	App	8ab6035	1
SGX-SQLite	App	c470f0a	1
Signal Contact Discovery	App	1.13	1
Total			80

TABLE 2: Summary of spatial bugs found in different projects.

Project	Unprotected Objects	Revision
SGX-SQLite	5	c470f0a
secure-analytics-sgx	1	43576a2
webasm	1	2281ac9
SGX-DFC	6	2281ac9

TABLE 3: Summary of temporal bugs found in different enclaves. Objects are the unprotected resources with data races.

6.1. Reported Bugs

We categorize the bugs found into spatial and temporal bugs. Spatial bugs are bugs related to invalid memory accesses, whereas, temporal bugs are caused by race conditions on valid memory accesses. To find these vulnerabilities, we ran the fuzzer for 96 hours, in the exhaustive configuration (section 4).

Spatial Bugs. Table 2 summarizes the spatial bugs found by FUZZSGX. Appendix B gives the full list of the bugs with the corresponding details. In general, we found that although SGX projects are security-oriented, they still suffer from bugs such as null pointer dereferences, out-of-bound accesses, divide-by-zero, etc. The majority of the bugs consisted of null pointer dereferences and out-of-bound accesses.

Temporal Bugs. Table 3 summarizes the temporal bugs found by FUZZSGX. Similar to spatial bugs, we only list projects where we were able to find any data races. We found that most of the enclaves with races are not designed to be multi-threaded, however, their TCSNum configuration parameters are set to be greater than 1 allowing concurrent enclave accesses from TCSNum threads. To mitigate this issue, we write a simple GCC analysis pass that parses through the enclaves to search for synchronization API calls and warns the user if the enclave is configured to allow multithreading without synchronization API usage.

6.2. Runtime Benchmark

In this section, we evaluate the overhead incurred by FUZZSGX RUNTIME. Since no existing benchmark is available for libOS approaches, we write our custom benchmark program that executes different system calls with randomized parameters. We run the benchmark natively on Linux, inside the enclave using FUZZSGX RUNTIME and, Graphene-SGX [37], a full-fledged security-oriented library OS (libOS) designed for enclaves. The overhead is averaged over ten executions to remove noise in the results. While FUZZSGX RUNTIME emulates 49 system calls and library functions. We focus on the functions

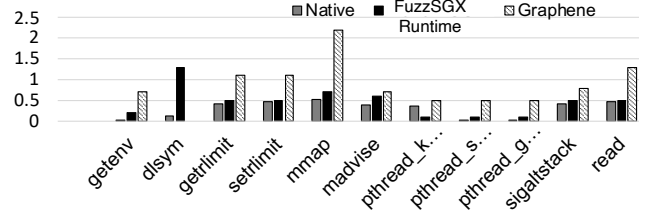


Figure 6: Average execution runtime in milliseconds of different system calls and library functions using native execution, FUZZSGX RUNTIME and Graphene-SGX. While native execution is the fastest in most cases, FUZZSGX RUNTIME incurs less overhead compared to Graphene-SGX.

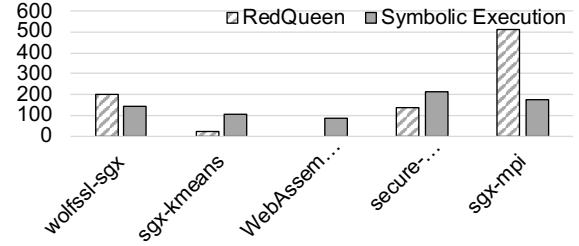


Figure 7: Comparison of basic block edge coverage based on the seed generation method. The y-axis shows the number of unique basic block edges, while the x-axis describes different projects.

that are always invoked while fuzzing an application. For instance, all target programs call `getenv` to find if the application is running under a fuzzer.

Figure 6 shows the results of this evaluation. In most cases, native execution is the fastest. For *pthread keys* related functions, if the enclave creates the key, FUZZSGX RUNTIME emulates these functions without leaving the enclave. Otherwise, the call is forwarded to the *pthread* library in the host app. Furthermore, to emulate *pthread keys*, FUZZSGX RUNTIME uses hashing instead of linear search, resulting in faster execution. In addition, we cache the results of system calls to emulate them without leaving the enclave. In all cases, Graphene-SGX’s emulation is slower compared to FUZZSGX RUNTIME, which stems from the fact that Graphene-SGX works within the threat model of SGX. Whenever Graphene-SGX transmits data to the untrusted runtime, it conducts CPU-intensive operations such as encryption to ensure confidentiality and integrity.

6.3. FUZZSGX Seed Generation

To evaluate the efficacy of the seed generation mechanisms employed by FUZZSGX, we measure the code coverage achieved by running the programs with the generated seed inputs. For this experiment, we run each target project for the same period of time, decided by the time taken by the symbolic execution engine (angr) to complete 100 steps for each ECALL. Figure 7 shows the basic block edge covered by the seed generated by each method. Based on our experiments, the coverage of seed inputs depends on the project. For projects with pointer input arguments, such as *sgx-mpi*, mutation-based techniques, such as REDQUEEN, seem to perform better,

Fuzzer	Executions per second
FuzzSGX	60.1K
SGXFuzz	4.03K
AFL	61.5K

TABLE 4: Executions per second averaged over a period of five minutes.

whereas for other projects, such as `sgx-kmeans`, symbolic execution generates better seed inputs. WebAssembly was an anomaly to this trend as mutation-based techniques were not able to generate any useful inputs within the time budget. Upon further investigation, we noticed that WebAssembly-SGX requires a `wasm` module, therefore, the input has to adhere to a compact binary instruction format.

6.4. Fuzzing Benchmark

We evaluate the FUZZSGX in terms of its efficacy in throughput, code coverage, and bug detection. In this evaluation, we establish SGXFuzz [26] as the baseline for our evaluation. Moreover, we also compare with AFL [48], by applying to SGX host apps directly. All fuzzers are run in a single-thread configuration.

Fuzzing Throughput: We compare the throughput of FUZZSGX with SGXFuzz and AFL. Since we want to focus on the fuzzing throughput irrespective of the target enclave complexity, we write a simple enclave that has a single ECALL. We run all of the fuzzers for a fixed duration of two minutes and get the average executions per second for each fuzzer. Table 4 shows the results of this experiment. On average, AFL shows the highest throughput, executing the target program around 62.0K times per second. Close to AFL, FUZZSGX executes the target program 60.1K times per second, whereas SGXFuzz can execute the same enclave around 4.03K times. FUZZSGX’s high throughput can be attributed to various factors, such as the low overhead of FUZZSGX RUNTIME. Moreover, since FUZZSGX RUNTIME enables compile-time instrumentation, FUZZSGX can collect the code coverage more efficiently compared to approaches such as hardware trace collection [28], emulation-based techniques [48], etc. Overall, FUZZSGX is approximately 15x faster than SGXFuzz.

Code Coverage: In this evaluation, we present ten projects from our evaluation data set to highlight the various behaviors we saw during our evaluation. We run all of the fuzzers for a fixed period of 24 hours to measure the code coverage. Figure 8 shows the measured code coverage of the evaluation. SGXFuzz was unable to fuzz SGX Kmeans as its test harness used was not able to satisfy the memory requirements for this project. More specifically, SGX Kmeans For most projects, we see a logarithmic progression shifted along the time for all fuzzers due to the fact that early mutations usually yield higher code coverage, whereas reaching deep code paths is harder, thus resulting in a flat line. Based on our results, AFL produced less code coverage for all of the evaluated projects, as it is not able to extract code coverage from within the enclave. FUZZSGX finds more code coverage for all projects, except `sgx-gmp-demo`. Upon further investigation, we note that FuzzSGX reports code coverage from the enclave implementation

and the proxy bridge between the enclave and the host app, whereas FUZZSGX only accounts for the code coverage from within the enclave. Based on this, we conjecture that because of a large number of ECALLs with simple implementations, SGXFuzz is able to find higher coverage in `sgx-gmp-demo`.

Moreover, we also note that FUZZSGX finds code coverage much faster compared to SGXFuzz and AFL. One outlier to this trend is WebAssembly SGX, where SGXFuzz initially found more code coverage. However, FUZZSGX quickly catches up with SGXFuzz within a couple of minutes. Thanks to the high throughput achieved by FUZZSGX RUNTIME and compile-time instrumentation, FUZZSGX achieves 3.2x higher coverage than SGXFuzz and 23x higher code coverage than AFL.

Bug Detection: In this evaluation, we count the number of bugs found by the fuzzer. We use the same projects as the code coverage throughput experiment and fuzz them for the same period of time of 24 hours. Figure 9 shows the number of unique crashes found in 24 hours. Similar to code coverage, the behavior varies from project to project. Except `sgx-mpi`, FUZZSGX finds more crashes compared to SGXFuzz for all other projects. Similarly to code coverage, AFL finds the lowest number of crashes, except for `sgx-gmp-demo` where it performed better than SGXFuzz. Due to the high throughput and effective sanitization achieved using FUZZSGX RUNTIME, FUZZSGX is able to find more crashes in the target programs. For the evaluated projects, FUZZSGX finds 1.48x more crashes compared to SGXFuzz and 11.6x more crashes compared to AFL.

6.5. Program Mutation Benchmark

To evaluate FUZZSGX program mutation, we pick six projects out of our evaluation data set based on the enclave configuration, such as the number of ECALLs. As `sgx-demo` has more than 40 ECALLs, besides `sgx-demo`, all of the projects were run in the exhaustive configuration of FUZZSGX. Furthermore, since `mbedtls` is intended to be used as a trusted library for enclaves, we wrote a custom EDL to expose `mbedtls` APIs. To evaluate the efficacy of program mutation, we fuzz the developer-provided host app without program mutation as the baseline configuration. Both configurations were run for a fixed amount of 96 hours.

Code Coverage: Figure 10 shows the final code coverage after running the fuzzer for 96 hours. We cannot run input-only mutation with `ContactDiscoveryEnclave`, as the corresponding host app is written in Java. For most applications, the developer-written application results in higher code coverage, since developers usually know the real dependencies between different ecalls. However, for projects such as `mbedtls`, we see higher code coverage with FUZZSGX generated programs, as the unmodified host app is only written for demonstration purposes and does not fully exercise the functionality of the enclave.

Bug Detection: Figure 11 shows the number of unique crashes found during the 96 hours. Program mutation configuration finds more crashes for every project. Especially for `secure-analytics-sgx`, FUZZSGX was able to find more than 100 crashes, whereas input mutation-only did not find a single crash. Upon further investigation, we

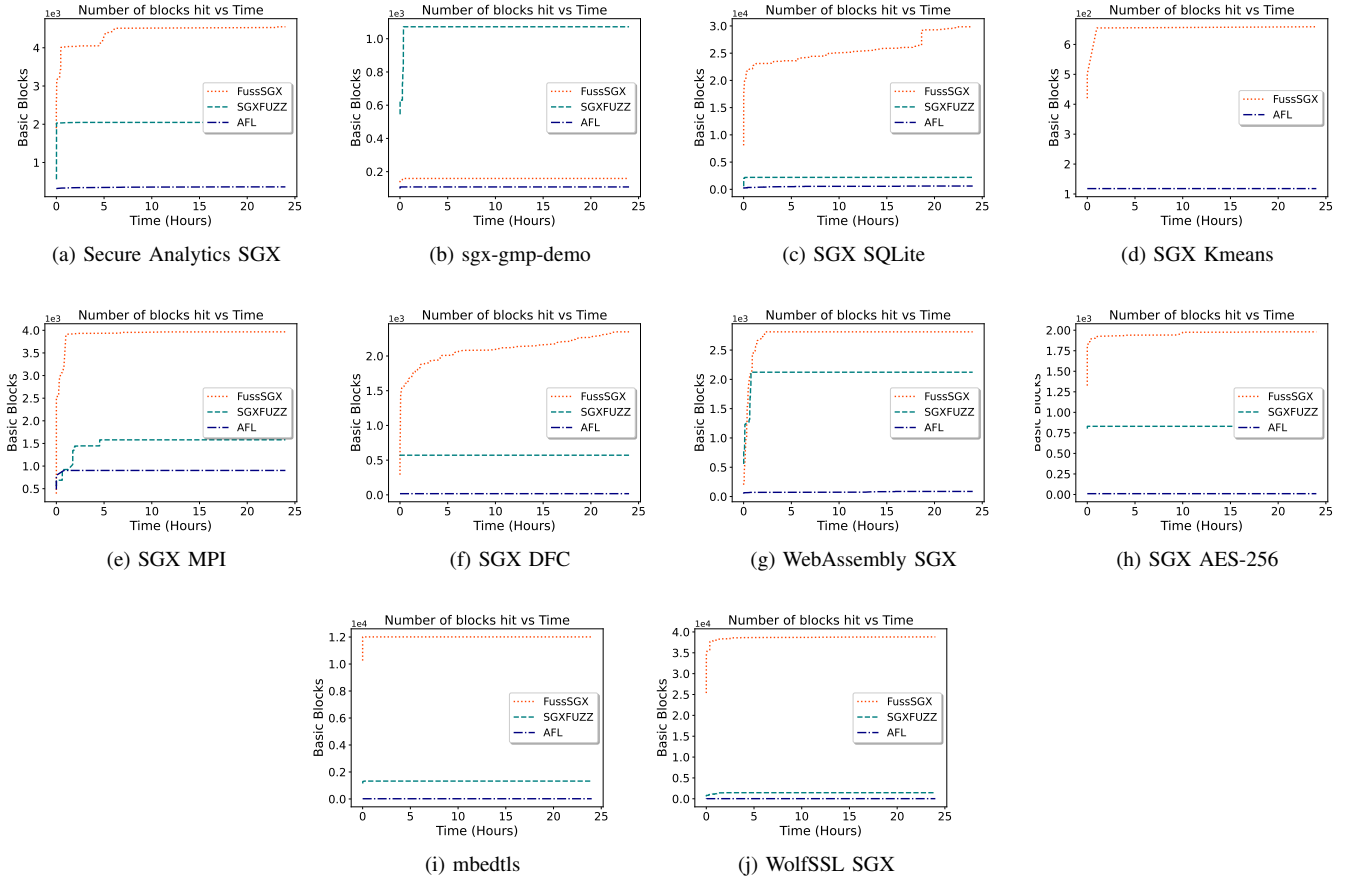


Figure 8: Basic block exploration found by different fuzzers.

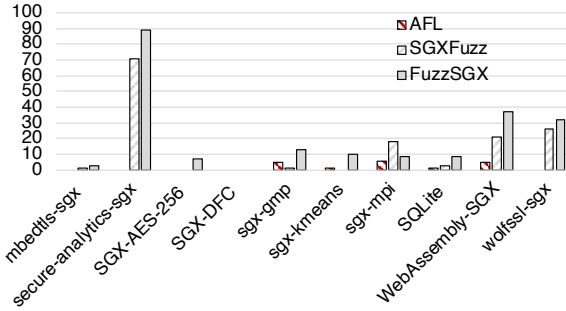


Figure 9: Number of unique crashes found in 24 hours found by different fuzzers.

found that the input sanitization logic is usually wrongly misplaced in the host apps rather than enclaves. However, as the SGX threat model indicates all parameters must be sanitized by the enclave code, this misplaced trust enables FUZZSGX to crash the enclave by fuzzing the ECALL interface directly. Due to this reason, even though we find lower code coverage, the unsanitized inputs trigger more crashes within the enclave, even before penetrating deep code paths.

6.6. Case Studies

SGX-LKL. The Linux Kernel Library (LKL) [47] leverages the Linux kernel such that it could be linked against

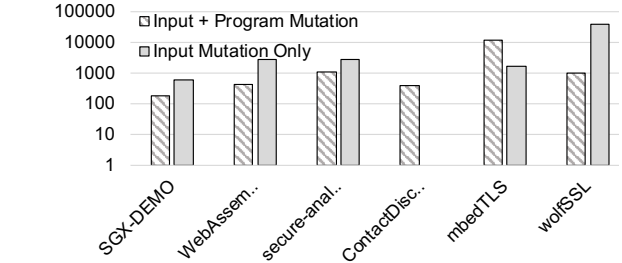


Figure 10: Number of blocks explored by input mutation compared to program and input mutation in 96 hours (log scale).

user applications while removing any dependence on the underlying architecture. This enables userspace applications to reuse features from the Linux kernel outside the kernel itself. SGX-LKL [49] provides the same functionality as LKL but inside an SGX enclave.

For communication between the enclave and the host app, SGX-LKL utilizes *VirtIO* [50], a lockless queuing and communication mechanism used for emulating devices. VirtIO users set up shared queues, and communicate over metadata specific to the device emulated. Because of this mechanism, SGX-LKL shares various pointers from the host app with the enclave. While fuzzing the host-enclave boundary, we found that the enclave uses such memory pointers without proper validation.

Listing 2 shows one such pointer. During the block

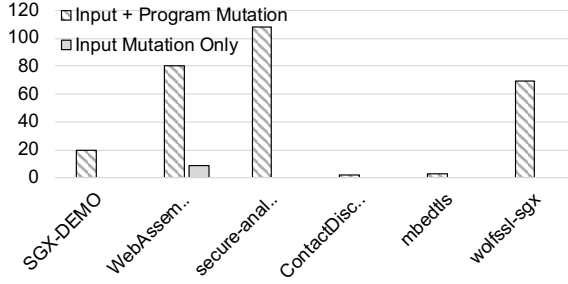


Figure 11: Number of bugs found by input mutation compared to program and input mutation in 96 hours.

```

293 for (size_t i = 0;
      i < enc->num_virtio_blk_dev;
      i++)
294 {
295   enc->virtio_blk_dev_mem[i] =
      host->virtio_blk_dev_mem[i];
296   const char* name =
      host->virtio_blk_dev_names[i];
297   size_t name_len = oe_strlen(name) + 1;
298   enc->virtio_blk_dev_names[i] =
      oe_alloc_or_die(
299     name_len,
300     sizeof(char),
301     "Could not allocate memory \n");
302   memcpy(enc->virtio_blk_dev_names[i],
      name, name_len);
303 }

```

Listing 2: VirtIO block initialization in SGX-LKL.

device initialization, the SGX-LKL enclave directly reads the data from untrusted memory without sanity checks, resulting in a null pointer dereference on line 295. Although adding sanity checks should fix the issue, this could still result in Time-of-check-to-time-of-use (TOCTTOU) [51] bugs, since the pointers are passed to the enclave using untrusted runtime and can be modified later, e.g., line 302, causing unintended behaviors inside the enclave.

These issues have been confirmed as security threats by the developers. This disclosure led to an overhaul for reviewing the shared pointers usage in the SGX-LKL project². Note that, in the original host app, it is not possible to call the ECALL with a null pointer. However, thanks to the program mutation employed by FUZZSGX, we are able to bypass the checking done in the host app to trigger this vulnerability.

WebAssembly Enclave. WebAssembly (wasm) [52] is a binary instruction format designed for a virtual machine running inside web browsers to achieve near-native performance. WebAssembly Enclave [53] runs the WebAssembly interpreter inside an SGX enclave. This enclave requires two inputs: 1) the path to a compiled wasm file to be interpreted, and 2) modifiers used to modify the interpreter behavior. We discovered various bugs while fuzzing the WebAssembly Enclave, including null pointer dereferences and out-of-bound accesses.

The WebAssembly Enclave uses the open syscall to access wasm files. However, in doing so, it uses the return value of the system call without any checking. This behavior can be exploited to conduct Iago attacks [25] enabling

2. Link omitted for anonymity

information side-channels, control-flow hijackings, and memory corruption [35], [54].

```

#define FATAL(...) { \
    printf(" Error(%s:%d):" \
        , __FILE__, __LINE__); \
    printf( __VA_ARGS__); \
}

```

Listing 1: FATAL macro used for failure handling.

Furthermore, the handling of the failure condition shown in Listing 1 is inadequate. In the original WebAssembly interpreter, this macro is used to halt the execution of the program and is called when the program faces an unmanageable error. However, in the WebAssembly Enclave, instead of halting or aborting the program, the program simply returns after printing the error, resulting in code execution with unexpected inputs. These unexpected inputs cause memory corruption in nine different execution paths within the enclave.

```

2013 if (fidx == -1) {
2014   entry = "_main";
2015   fidx = get_export_fidx(m, entry);
2016 }
2017 if (fidx == -1) {
2018   FATAL("no exported function"
2019     "named '%s'\n", entry);
2019 }
2020 type = m->functions[fidx].type;

```

Listing 3: WebAssembly enclave code for getting function index.

For instance, in Listing 3, the enclave tries to find the function index, `fidx`, based on the input arguments. However, if the function is not found, the enclave calls the FATAL macro at line 2018. The subsequent code is written with the assumption that `fidx` has a sane value and continues execution with an invalid `fidx`, resulting in memory corruption inside the enclave.

```

1520 Module *load_module(uint32_t len,
      uint8_t* file_contents, char *path, Options options){
1521   uint32_t mod_len;
1522   uint8_t *bytes;
1523   uint8_t vt;
1524   uint32_t pos = 0, word;
1525   Module *m;
1526
1527   // Allocate the module
1528   m = (Module*) aalloc(1, sizeof(Module), "Module");
1529   m->path = path;
1530   m->options = options;
1531   ...
1991   _wa_current_module_ = m;
1992

```

Listing 4: WebAssembly enclave code for interpreting instructions.

Listing 4 shows a data race in the WebAssembly enclave that can lead to data leakage. Whenever wasm code is executed in the WebAssembly Enclave, the enclave creates a module for processing the code. At the end of each execution, the module is cleaned so that the computation's secrecy is preserved. During fuzzing, we found that the variable `_wa_current_module_` pointing to the current module is not protected by any locks, enabling data leakage

from within the enclave. Moreover, the variable can be easily controlled using the input arguments of the ECALL. To do so, a malicious thread can preempt an ECALL mid-processing after the instruction at line 1991. Next, the attacker waits for a user to invoke the same ECALL. Once the ECALL reaches line 1991 in the victim thread, we can suspend the victim thread and resume our malicious thread. Since there is only one copy of the current module, the current module will point to the module of the victim thread. Therefore, in the attacker thread, we can read the information contained in the victim module thread using the malicious thread. To achieve this fine-grained execution control, attackers can use SGX-Step [55].

Case Study: SGX-KMEANS. K-means is a clustering algorithm used in various applications such as data mining [56], data compression [57], and pattern recognition [58]. SGX-KMEANS [59] trains a k-means model inside an SGX enclave protecting the Intellectual Property (IP) of the k-means model. While fuzzing SGX-KMEANS, we find that it fails to sanitize the input parameters in the code path shown in Listing 5. The value of k is supplied by the untrusted runtime and used without checking. Furthermore, after lines 18, 24, and 23, there are no checks to see if the dynamic memory allocations were successful. Hence, calling the ECALL with a large value of k fails the memory allocation, resulting in a null pointer dereference on line 29.

```

18 float ** c1 = (float ** ) calloc(k*m,
    sizeof(float * )); // temp centroids
19
23 for (h = i = 0; i < k; h += n / k, i++) {
24   c1[i] = (float * ) calloc(m, sizeof(float));
25   if (!centroids) {
26     c[i] = (float * ) calloc(m, sizeof(float));
27   }
28   // pick k points as initial centroids
29   for (j = m; j-- > 0; c[i][j] = data[h][j]);

```

Listing 5: Source Code

7. Discussion

FUZZSGX security model feasibility SGX threat model considers only the enclave implementation as trusted, as described in subsection 2.2. Due to this design choice, only the enclave and the hardware are checked for integrity measurements, leaving the malicious host app to freely control the inputs to an enclave. An attacker can concretely exploit the found bugs by modifying host apps to send arbitrary inputs to the enclave, including malicious inputs triggering the vulnerabilities found by FUZZSGX. We showcase two different mechanisms to control the input to the enclaves: 1) rebuilding the host app, and 2) binary rewriting.

Rebuilding Host App: Enclaves are loosely coupled to their host apps. Due to this design, an attacker can rebuild the host app with malicious code to freely control the inputs to the enclave. FUZZSGX is a running demonstration of this mechanism, as we mutate the host app to generate a "malicious" copy such that FUZZSGX uses to find inputs to crash the program.

Binary Rewriting: Binary rewriting patches the behavior of an existing binary by rewriting instructions in a binary.

To demonstrate how binary rewriting can control the inputs to enclaves, we patch the code using both dynamic and static binary rewriting techniques [60]. Dynamic binary rewriting modifies the code during execution. On a high level, it patches the code by obtaining writing permission to code memory and modifies the prologue of a function or a function pointer to redirect code execution to malicious code. For our purposes, we use a simple detour-based patch. We use the funchook [61] library to create a detour that modifies the input arguments to the enclave. After applying the patch we can fully control the value transmitted to the enclave. On the other hand, static binary rewriting patches the binary before it runs. While this mechanism is more stealthy, it requires manual effort to find the addresses of the functions and variables. As a demonstration, we use Patcherex [62] with the detour-based backend to control the input to the enclave.

Applying FUZZSGX to other SDK. FUZZSGX has been implemented with the Intel SGX SDK, however, our design is generic to retrofit other TEE SDK implementations. As a demonstration, we cover a few SDK implementations to describe how FUZZSGX can work with these implementations. In general, to port FUZZSGX, users need to: 1) link the trusted library (libsgx_tsgxfuzz.so) to the trusted partition, 2) link the untrusted library (libsgx_usgxfuzz.so) to the untrusted partition, and 3) add glue logic to enable communication between the libsgx_tsgxfuzz.so and libsgx_usgxfuzz.so.

OpenEnclave [63] (OE): OE provides a TEE architecture-agnostic way to make enclave applications. The workflow is similar to the Intel SGX SDK and the applications are partitioned into a host app and an enclave. Furthermore, the interface is defined using an EDL file. Moreover, the EDL format used by OE is similar to the Intel SGX SDK's EDL format. Users can link libsgx_tsgxfuzz.so with the enclave and libsgx_usgxfuzz.so to the host app. Lastly, users can write the OE EDL based on the Intel SGX SDK EDL to enable communication between the libraries.

Keystone [64]: Keystone-SDK enables building applications for the Keystone (RISC-V) TEE architecture. While this SDK is developed for a completely different architecture, the SDK provides similar facilities as Intel SGX SDK. Keystone applications use *Edge* libraries to communicate between the host and enclave³. For enclave manipulation purposes, the host app is linked with *Host* libraries, whereas the programming environment in the enclave is defined by the *Enclave Application* libraries.⁴ Keystone also requires a trusted runtime or kernel to run in the S-Mode of the trusted world. The trusted runtime used by Keystone-SDK is called *Eyrie*. Users can port FUZZSGX to Keystone-SDK by linking libsgx_tsgxfuzz.so with either the *Eyrie runtime* or *Enclave Application*, and linking libsgx_usgxfuzz.so to the host application. Lastly, users will need to implement the glue logic to enable communication between the trusted and untrusted libraries. However, unlike OE and the Intel SGX SDK, Keystone-SDK does not provide an automated way to generate the glue code. To this end, users will need to manually write the glue code to enable communication between libsgx_tsgxfuzz.so and libsgx_usgxfuzz.so.

3. Currently, only OCALLs are supported. However, ecalls can be emulated using a polling mechanism.

4. Keystone applications are also linked with verification libraries, but for our purposes, we do not consider the verification procedure.

Conclusion: In general, each SDK splits the application into a secure (enclave) and non-secure (host app) part. Since we design FUZZSGX with this partitioned model in mind, we apply FUZZSGX to most SDKs with a partitioned application model. Note that, FUZZSGX does not work with un-partitioned TEE applications, i.e., SDKs that allow running unmodified applications inside an enclave, such as Graphene [37], SGX-LKL [47], etc., or SDKs providing Function as a Service (FAAS), such as Enarx [65], Teaclave [66], etc.

Fuzzing hardware-mode enclaves Due to the limited resources available for enclave mode, fuzzing enclaves in hardware mode does not scale. A naive way to tackle this problem could be emulating the enclave inside an emulator [67]. However, the SGX attestation process includes hardware integrity measurements and it should be impossible for fuzzers to bypass the SGX remote attestation when instrumented with fuzzer and sanitizer instrumentation [23]. If a release mode enclave does not properly implement attestation or enables debug support, there are much simpler mechanisms to thwart the security guarantees of SGX.

Different mutation strategies. FUZZSGX uses both program and input mutation for fuzzing. Each mutated program is fuzzed using input mutation until a pre-defined condition is met. As mentioned in Section 4, these are naive conditions, such as fixed timeout. However, the optimal strategy for initiating program mutation is dependent on the target program and could be a combination of different conditions. For instance, while fuzzing programs with several order dependencies, focusing on program mutations would be more beneficial as only input mutations will not be able to resolve these dependencies. We leave this exploration of optimal mutation strategies as future work.

Dynamically provisioned enclaves. SGX enclaves can dynamically provision code for confidentiality purposes [68]. As these systems are used in release mode, we do not expect them to be compatible with FUZZSGX. Alternatively, these frameworks can be modified to link the enclave code with FUZZSGX RUNTIME before dynamically provisioning the code. However, Supporting such frameworks is outside the scope of this paper.

Bug reproducibility. For FUZZSGX, we were able to reproduce all of the bugs reported by the fuzzer, besides one particular project, SGX-SQLite. In the case of SQLite, the enclave maintains its internal state in the current database across executions. During fuzzing we were not able to capture the internal state of the project, resulting in bug reports where the input values were not able to reproduce the bug. We leave stateful fuzzing support for FUZZSGX as future work.

8. Related Work

SGX Vulnerabilities. Intel SGX introduces a novel way to minimize the TCB for an application. However, Intel SGX is vulnerable to multiple types of attacks. Jaehyuk et.al, [69] corrupted the memory of the ASLR-enabled SGX enclave successfully by using modified return-oriented programming attacks. Several works [70], [71] have shown that SGX is vulnerable to speculative execution attacks. Moreover, several side-channel attacks [55], [72] have

been successful in stealing data from within the enclaves, bypassing SGX memory protection.

SGX Vulnerability Detection and Prevention Tools. TeeRex [23] uses symbolic execution and pointer tracking to find bugs in enclaves. Khandaker et. al, [73] uses symbolic execution to find bugs in SGX programs. Similar to TeeRex, this paper also faces the path explosion problem. FUZZSGX makes symbolic execution optional, scaling it to larger projects. Like FUZZSGX, Emilia [24] fuzzes the syscall-app boundary in legacy applications to discover Iago vulnerabilities, however, Emilia works on legacy programs instead of SGX applications. Furthermore, it does not fuzz the user-host app boundary. FUZZSGX can mutate input at more boundaries in the system. De Backer [74] exploited side-channels to infer crashes inside enclaves, unlike FUZZSGX this work is not used to find new bugs but only to infer a crash inside an enclave.

Furthermore, there exist different works to protect SGX from vulnerabilities. Moat [75] protects the confidentiality of the SGX program against infrastructure protocol attacks. Several works [76], [77] have been proposed to defend SGX programs from side-channel attacks. SGXBounds [78] instruments SGX enclaves with bounds checking to mitigate memory corruption bugs. MPTEE [79] enables flexible page-level memory protection inside SGX enclaves.

Fuzzers. Even though FUZZSGX is the first coverage-guided fuzzer for Intel SGX applications, fuzzing has been a well-researched area in other domains, and various techniques, such as static analysis, interface extraction, symbolic execution, emulation, etc., have been utilized to augment the fuzzing process. For interface aware fuzzing, IMF [80] infers the API model using value and order dependence from kernel traces. Similarly, RESTLER [81] infers the producer-consumer dependencies for REST APIs by dynamically analyzing the responses for REST requests. While both of these tools try to infer the dependencies between different calls the methodology used is entirely different. FUZZSGX uses static analysis to extract dependencies, whereas these works use dynamic analysis to infer program dependencies. DIFUZE [82] conducts interface-aware fuzzing by learning the ioctl interface implemented by the device drivers. Similarly, like FUZZSGX, other fuzzers have used static analysis to aid fuzzing. There has been existing work on leveraging static analysis to augment the fuzzing process. Moonshine [83] uses static analysis to distill meaningful seeds for OS fuzzers by collecting the system call traces from userspace programs. Razzer [84] uses static analysis to find race conditions in the Linux kernel. Like FUZZSGX, several fuzzers [85], [86] have used symbolic execution to improve the efficacy of fuzzing.

9. Conclusion

While Intel SGX has been released for a while, a comprehensive fuzzing tool for SGX programs has been missing. In this paper, we presented FUZZSGX, a comprehensive fuzzing suite for Intel SGX enclave implementations. We show that Intel SGX programs still suffer from memory corruption bugs. FUZZSGX can find those corruptions using techniques such as symbolic execution, input mutation, and program mutation. By using FUZZSGX we found 93 bugs in 30 SGX projects. Based on

our results, FUZZSGX is a promising tool for automatically discovering bugs in real-world SGX programs.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported in part by NSF under grant NSF CNS-2145744. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the NSF. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

References

- [1] Ayeks, “ayeks/sgx-hardware: This is a list of hardware which is supports intel sgx - software guard extensions.” <https://github.com/ayeks/SGX-hardware>, 05 2022, (Accessed on 10/03/2021).
- [2] K. Kim, C. H. Kim, J. J. Rhee, X. Yu, H. Chen, D. J. Tian, and B. Lee, “Vessels: Efficient and scalable deep learning prediction on trusted processors,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 462–476. [Online]. Available: <https://doi.org/10.1145/3419111.3421282>
- [3] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, “Vc3: Trustworthy data analytics in the cloud using sgx,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 38–54.
- [4] F. Tramèr and D. Boneh, “Slalom: Fast, verifiable and private execution of neural networks in trusted hardware,” in *International Conference on Learning Representations*, 2019. [Online]. Available: <https://openreview.net/forum?id=rJVorjCkKQ>
- [5] C. Priebe, K. Vaswani, and M. Costa, “Enclavedb: A secure database using sgx,” in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 264–278.
- [6] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, “Obliv: An efficient oblivious search index,” in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 279–296.
- [7] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, “Pesos: Policy enhanced secure object store,” in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, ser. EuroSys ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3190508.3190518>
- [8] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, “Shieldstore: Shielded in-memory key-value storage with sgx,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys ’19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3302424.3303951>
- [9] S. Eskandarian and M. Zaharia, “Oblidb: Oblivious query processing for secure databases,” *Proc. VLDB Endow.*, vol. 13, no. 2, p. 169–183, Oct. 2019. [Online]. Available: <https://doi.org/10.14778/3364324.3364331>
- [10] S. Shinde, S. Wang, P. Yuan, A. Hobor, A. Roychoudhury, and P. Saxena, “Besfs: A POSIX filesystem for enclaves with a mechanized safety proof,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 523–540. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/shinde>
- [11] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, “Obliviate: A data oblivious filesystem for intel sgx,” in *NDSS*. : NDSS, 2018.
- [12] J. I. Choi, D. J. Tian, G. Hernandez, C. Patton, B. Mood, T. Shrimpton, K. R. B. Butler, and P. Traynor, “A Hybrid Approach to Secure Function Evaluation using SGX,” in *Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS’19)*, 2019.
- [13] S. Arnaudov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. L. Stillwell, D. Goltzsche, D. Eysers, R. Kapitza, P. Pietzuch, and C. Fetzer, “SCONE: Secure linux containers with intel SGX,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703. [Online]. Available: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov>
- [14] R. Paccagnella, P. Datta, W. Hassan, A. Bates, C. Fletcher, A. Miller, and D. Tian, “Custos: Practical tamper-evident auditing of operating systems using trusted execution,” in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 01 2020.
- [15] D. J. Tian, J. I. Choi, G. Hernandez, P. Traynor, and K. R. B. Butler, “A practical intel sgx setting for linux containers in the cloud,” in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 255–266. [Online]. Available: <https://doi.org/10.1145/3292006.3300030>
- [16] D. Sturzenegger, A. Sardon, S. Deml, and T. Hardjono, “Confidential computing for privacy-preserving contact tracing,” *arXiv preprint arXiv:2006.14235*, 2020.
- [17] F. Y. Rashid, “The rise of confidential computing: Big tech companies are adopting a new security model to protect data while it’s in use-[news],” *IEEE Spectrum*, vol. 57, no. 6, pp. 8–9, 2020.
- [18] Intel, “Linux sgx sdk,” <https://github.com/intel/linux-sgx>, 2022.
- [19] M. Schwarz, S. Weiser, and D. Gruß, “Practical enclave malware with intel sgx,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science. Springer International, 6 2019, pp. 177–196.
- [20] 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.
- [21] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, “Sgxax: How sgx fails in practice,” 2020.
- [22] N. Weichbrodt, A. Kurmus, P. R. Pietzuch, and R. Kapitza, “Asynchoshock: Exploiting synchronisation bugs in intel sgx enclaves,” in *ESORICS*, 2016.
- [23] 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.
- [24] R. Cui, L. Zhao, and D. Lie, “Emilia: Catching iago in legacy code,” in *Network and Distributed Systems Security (NDSS) Symposium 2021 21-25 February 2021*, 2021.
- [25] 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.
- [26] T. Cloosters, J. Willbold, T. Holz, and L. Davi, “{SGX}Fuzz: Efficiently synthesizing nested structures for {SGX} enclave fuzzing,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3147–3164.
- [27] “Attestation services for intel® software guard extensions,” <https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html>, (Accessed on 03/11/2023).
- [28] A. Kleen and B. Strong, “Intel processor trace on linux,” *Tracing Summit*, vol. 2015, 2015.
- [29] “Binaryonly fuzzing | aflplusplus,” https://aflplusplus.com/docs/binaryonly_fuzzing/, (Accessed on 03/11/2023).
- [30] 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.

- [31] 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.
- [32] V. Costan and S. Devadas, "Intel sgx explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.
- [33] AFL, "Ubuntu manpage: afl-gcc - gcc wrapper for american fuzzy lop (afl)," <http://manpages.ubuntu.com/manpages/bionic/man1/afl-gcc.1.html>, 05 2022, (Accessed on 04/04/2021).
- [34] Kirit1193, "kirit1193/intel-sgx-fuzzer: Fuzz sealing and unsealing operations in sgx using afl," <https://github.com/kirit1193/Intel-SGX-Fuzzer>, 05 2022, (Accessed on 10/03/2021).
- [35] 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.
- [36] Signal, "Signal contact discovery," <https://signal.org/blog/private-contact-discovery/>, 2022.
- [37] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 645–658.
- [38] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 955–970.
- [39] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 121–133, 2009.
- [40] Google, "Tsan support for static libraries," <https://github.com/google/sanitizers/issues/1383>, 2022.
- [41] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "{FuzzGen}: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287.
- [42] 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.
- [43] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 861–875.
- [44] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "Grimoire: Synthesizing structure while fuzzing," in *USENIX Security Symposium*, vol. 19, 2019.
- [45] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *NDSS*, vol. 19, 2019, pp. 1–15.
- [46] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.
- [47] O. Purdila, L. A. Grijincu, and N. Tapus, "Lkl: The linux kernel library," in *9th RoEduNet IEEE International Conference*. IEEE, 2010, pp. 328–333.
- [48] "afl/readme.qemu at master · mirrorer/afl · github," https://github.com/mirrorer/afl/blob/master/qemu_mode/README.qemu, (Accessed on 03/14/2023).
- [49] C. Priebe, D. Muthukumar, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "Sgx-lkl: Securing the host os interface for trusted execution," *arXiv preprint arXiv:1908.11143*, 2019.
- [50] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.
- [51] D. Tsafir, T. Hertz, D. A. Wagner, and D. Da Silva, "Portably solving file toctou races with hardness amplification," in *FAST*, vol. 8, 2008, pp. 1–18.
- [52] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.
- [53] Github, "Web assembly enclave," <https://github.com/SabaEskandarian/webassemblyEnclave>, 2022.
- [54] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 640–656.
- [55] J. V. Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.
- [56] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth et al., "Knowledge discovery and data mining: Towards a unifying framework," in *KDD*, vol. 96, 1996, pp. 82–88.
- [57] K. L. Oehler and R. M. Gray, "Combining image compression and classification using vector quantization," *IEEE transactions on pattern analysis and machine intelligence*, vol. 17, no. 5, pp. 461–473, 1995.
- [58] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern classification and scene analysis*. Wiley New York, 1973, vol. 3.
- [59] S. K-means, "Sgx k-means," <https://github.com/dsc-sgx/sgx-kmeans>, 2022.
- [60] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–37, 2019.
- [61] Kubo, "kubo/funchook: Hook function calls by inserting jump instructions at runtime," <https://github.com/kubo/funchook>, 2022, (Accessed on 05/01/2022).
- [62] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, C. Salls, N. Stephens, R. Wang, and G. Vigna, "Mechanical phish: Resilient autonomous hacking," *IEEE Security Privacy*, vol. 16, no. 2, pp. 12–22, 2018.
- [63] "Open enclave · github," <https://github.com/openenclave>, (Accessed on 10/26/2022).
- [64] "Keystone enclave · github," <https://github.com/keystone-enclave>, (Accessed on 10/26/2022).
- [65] "Enarx | enarx," <https://enarx.dev/>, (Accessed on 10/26/2022).
- [66] "Github - apache/incubator-teaclave: Apache teaclave (incubating) is an open source universal secure computing platform, making computation on privacy-sensitive data safe and simple." <https://github.com/apache/incubator-teaclave>, (Accessed on 10/26/2022).
- [67] P. Jain, S. J. Desai, M.-W. Shih, T. Kim, S. M. Kim, J.-H. Lee, C. Choi, Y. Shin, B. B. Kang, and D. Han, "Opensgx: An open platform for sgx research," in *NDSS*, vol. 16, 2016, pp. 21–24.
- [68] E. Bauman, H. Wang, M. Zhang, and Z. Lin, "Sgxelide: enabling enclave code secrecy via self-modification," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 75–86.
- [69] 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)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 523–539. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk>
- [70] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/bulck>
- [71] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 142–157.

- [72] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "Platypus: Software-based power side-channel attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.
- [73] M. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "Coin attacks: On insecurity of enclave untrusted interfaces in sgx," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, 03 2020, pp. 971–985.
- [74] T. De Backer, "Fuzzing intel sgx enclaves," Master's thesis, KU Leuven, 2018.
- [75] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.
- [76] Z. yu Zhang, X. li Zhang, Q. Li, K. Sun, Y. Zhang, S. Liu, Y. Liu, and X. Li, "See through walls: Detecting malware in sgx enclaves with sgx-bouncer," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2021.
- [77] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," *arXiv preprint arXiv:2006.13353*, 2020.
- [78] 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.
- [79] W. Zhao, K. Lu, Y. Qi, and S. Qi, "Mptee: bringing flexible and efficient memory protection to intel sgx," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15.
- [80] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2345–2358.
- [81] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758.
- [82] 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.
- [83] S. Pailoor, A. Aday, and S. Jana, "Moonshine: optimizing os fuzzer seed selection with trace distillation," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 729–743.
- [84] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 754–768.
- [85] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel," in *NDSS*, 2020.
- [86] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: a practical concolic execution engine tailored for hybrid fuzzing," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 745–761.
- [87] S. S. Enclave, "Sgx sqlite enclave," https://github.com/yerzhan7/SGX_SQLite, 2022.
- [88] S. MPI, "Sgx mpi," <https://github.com/dsc-sgx/sgx-mpi>, 2022.
- [89] S. Analytics, "Secure analytics," <https://github.com/swarupchandra/secure-analytics-sgx>, 2022.

Appendix

1. Evaluation Dataset Selection Criteria

For evaluating FUZZSGX, we searched Github with "SGX" keyword and filtered out the top 50 projects based on stars. Out of these projects, we fuzzed the projects that FUZZSGX was able to fuzz, as FUZZSGX only works

#	Project Name	Project Version
1	EleOS	15a6f23
2	TalOS	bb0b619
3	sgx-ra-sample	96f5b5c
4	secpass	b89454f
5	hello-enclave	a3358ac
6	Graphene	bdddc84
7	SGX-LKL	69073fb
8	mbedtls-SGX	eab8e36
9	SGX-DFC	d3f7e47
10	BiORAM-SGX	8a16861
11	SGX-AES-256	b0d87a8
12	SGX-server-client	ae36ec
13	SGX_SQLite	c470f0a
14	eigen-sgx	34dc0d0
15	mnist-sgx	07ab764
16	secure-analytics-sgx	43f7e62
17	sgx-db	dafef82
18	sgx-gmp-demo	85cd040
19	sgx-gwas	24e59f3
20	sgx-kmeans	8ab6035
21	sgx-log	7b5530e
22	sgx-mpi	504ce81
23	sgx-nbench	799f0fc
24	sgx-pwenclave	b81eace
25	sgx_protect_file	5f2e64e
26	webassemblyEnclave	2281ac9
27	wolfssl-examples	6a165d8
28	SGX-OpenSSL	2a1b2b4
29	signalapp	a6a0c15
30	sgx-wallet	700f2a6

TABLE 5: List of projects fuzzed.

with C code and is currently only adaptable with Intel SGX SDK projects. In addition to this, we try to evaluate projects, explored by existing vulnerability finding tools such as Emilia and TeeREX. Furthermore, where applicable we also tried fuzzing projects in dumb mode, i.e., without code coverage feedback. Table 5 lists all the projects we have fuzzed.

2. Bug Report

Table 6 gives the full list of spatial bugs found by FUZZSGX.

No.	Type	Project Name	Version	Function / File:line	Status
1	Null pointer access	webasm [53]	2281ac9	main / App.cpp:447	Confirmed
2	Null pointer access	webasm [53]	2281ac9	load_module / wa.cpp:1731	Confirmed
3	Null pointer access	webasm [53]	2281ac9	acalloc / util.cpp:89	Confirmed
4	Null pointer access	webasm [53]	2281ac9	load_module / wa.cpp:1953	Confirmed
5	Null pointer access	webasm [53]	2281ac9	arealloc / util.cpp:99	Confirmed
6	Null pointer access	webasm [53]	2281ac9	invoke / wa.cpp:2018	Confirmed
7	Null pointer access	webasm [53]	2281ac9	thunk_out / wa.cpp:555	Confirmed
8	Null pointer access	webasm [53]	2281ac9	load_module / wa.cpp:1669	Confirmed
9	Null pointer access	webasm [53]	2281ac9	read_LEB / util.cpp:31	Confirmed
10	Null pointer access	webasm [53]	2281ac9	setup_thunk_in / wa.cpp:605	Confirmed
11	Out-of-bounds access*	SGX SQLite [87]	c470f0a	Unknown*	Notified
12	Null pointer access	sgx-ikl [49]	c8cb0b8	lkl_virtio_console_add / virtio_console.c:40	Confirmed
13	Null pointer access	sgx-ikl [49]	c8cb0b8	initialize_enclave_event_channel / enclave_event_channel.c:180	Confirmed
14	Null pointer access	sgx-ikl [49]	c8cb0b8	enclave_nanos / enclave_timer.c:31	Confirmed
15	Null pointer access	sgx-ikl [49]	c8cb0b8	copy_shared_memory / enclave_oe.c:295	Confirmed
16	Null pointer access	sgx-ikl [49]	c8cb0b8	copy_shared_memory / enclave_oe.c:296	Confirmed
17	Null pointer access	sgx-ikl [49]	c8cb0b8	free_shared_memory / enclave_oe.c:332	Confirmed
18	Null pointer access	graphene [37]	v1.0.1	sgx_copy_to_enclav / enclave_framework.c:89	Confirmed
19	Null pointer access	graphene [37]	v1.1	sgx_copy_to_enclav / enclave_framework.c:89	Confirmed
20	Abort	sgx-mpi [88]	504ce81	main / App.cpp:187	Notified
21	Out-of-bounds access	linux-sgx [18]	2.11	ECALL_new_container_classes_demo / Libcxx.cpp:594	Notified
22	Hang	sgx-kmeans [59]	8ab6035	k_means / Enclave.cpp:32	Confirmed
23	Out-of-bounds access	linux-sgx [18]	2.11	init_global_object / global_init.c:186	Confirmed
24	Out-of-bounds access	linux-sgx [18]	2.11	trts_ECALL / trts_ecall.cpp:259	Notified
25	Heap overflow	signalapp [36]	1.13	main / app.c:412	NA
26	Heap overflow	secure-analytics-sgx [89]	43576a2	extract_features / analytics_util.cpp:278	Notified
27	Heap overflow	secure-analytics-sgx [89]	43576a2	main / analytics_util.cpp:278	Notified
28	Abort	secure-analytics-sgx [89]	43576a2	startDTObivTraining / dt_obliv.cpp:178	Notified
29	Abort	secure-analytics-sgx [89]	43576a2	startDTObivTesting / dt_obliv.cpp:252	Notified
30	Heap overflow	secure-analytics-sgx [89]	43576a2	DTLearn_obliv / dt_obliv.cpp:76	Notified
31	Abort	secure-analytics-sgx [89]	43576a2	startDTRandTraining / dt_rand.cpp:422	Notified
32	Abort	secure-analytics-sgx [89]	43576a2	startDTRandTesting / dt_rand.cpp:506	Notified
33	Abort	secure-analytics-sgx [89]	43576a2	startDTTesting / dt_sgx.cpp:138	Notified
34	Abort	secure-analytics-sgx [89]	43576a2	startDTTraining / dt_sgx.cpp:68	Notified
35	Abort	secure-analytics-sgx [89]	43576a2	startKMOblivTraining / km_obliv.cpp:349	Notified
36	Abort	secure-analytics-sgx [89]	43576a2	startKMOblivTraining / km_obliv.cpp:351	Notified
37	Heap overflow	secure-analytics-sgx [89]	43576a2	startKMOblivTraining / km_obliv.cpp:351	Notified
38	Abort	secure-analytics-sgx [89]	43576a2	startKMOblivTraining / km_obliv.cpp:441	Notified
39	Heap overflow	secure-analytics-sgx [89]	43576a2	startKMOblivTraining / km_obliv.cpp:441	Notified
40	Abort	secure-analytics-sgx [89]	43576a2	startKMRandTraining / km_rand.cpp:202	Notified
41	Abort	secure-analytics-sgx [89]	43576a2	startKMRandTraining / km_rand.cpp:305	Notified
42	Abort	secure-analytics-sgx [89]	43576a2	test / km_sgx.cpp:150	Notified
43	Heap overflow	secure-analytics-sgx [89]	43576a2	test / km_sgx.cpp:150	Notified
44	Abort	secure-analytics-sgx [89]	43576a2	main / km_sgx.cpp:191	Notified
45	Heap overflow	secure-analytics-sgx [89]	43576a2	startKMTraining / km_sgx.cpp:191	Notified
46	Abort	secure-analytics-sgx [89]	43576a2	startKMTraining / km_sgx.cpp:198	Notified
47	Abort	secure-analytics-sgx [89]	43576a2	startKMTraining / km_sgx.cpp:228	Notified
48	Abort	secure-analytics-sgx [89]	43576a2	startKMTraining / km_sgx.cpp:236	Notified
49	Abort	secure-analytics-sgx [89]	43576a2	startKMTraining / km_sgx.cpp:248	Notified
50	Abort	secure-analytics-sgx [89]	43576a2	startKMTTesting / km_sgx.cpp:281	Notified
50	Abort	secure-analytics-sgx [89]	43576a2	startKMTTesting / km_sgx.cpp:281	Notified
50	Heap overflow	secure-analytics-sgx [89]	43576a2	closestCentroid / km_sgx.cpp:29	Notified
51	Heap overflow	secure-analytics-sgx [89]	43576a2	EM / km_sgx.cpp:58	Notified
53	Abort	secure-analytics-sgx [89]	43576a2	startNBOblivTesting / nb_obliv.cpp:133	Notified
54	Abort	secure-analytics-sgx [89]	43576a2	startNBOblivTraining / nb_obliv.cpp:64	Notified
55	Abort	secure-analytics-sgx [89]	43576a2	startNBRandTraining / nb_rand.cpp:36	Notified
56	Abort	secure-analytics-sgx [89]	43576a2	startNBRandTesting / nb_rand.cpp:107	Notified
57	Abort	secure-analytics-sgx [89]	43576a2	startNBRandTraining / nb_rand.cpp:36	Notified
58	Divide by zero	secure-analytics-sgx [89]	43576a2	startNBRandTesting / nb_rand.cpp:98	Notified
59	Abort	secure-analytics-sgx [89]	43576a2	startNBTesting / nb_sgx.cpp:127	Notified
60	Abort	secure-analytics-sgx [89]	43576a2	startNBTraining / nb_sgx.cpp:70	Notified
61	Abort	secure-analytics-sgx [89]	43576a2	initialize_NB / nbayes.cpp:17	Notified
62	Abort	secure-analytics-sgx [89]	43576a2	initialize_NB / nbayes.cpp:26	Notified
63	Heap overflow	webasm [53]	2281ac9	invoke / wa.cpp:2020	Confirmed
64	Abort	webasm [53]	2281ac9	read_string / util.cpp:59	Confirmed
65	Null pointer access	webasm [53]	2281ac9	read_string / util.cpp:59	Confirmed
66	Heap overflow	webasm [53]	2281ac9	sgx_ecall_load_invoke_allInOne / Enclave_t.c:379	Confirmed
67	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:1004	Confirmed
68	Heap overflow	webasm [53]	2281ac9	acalloc / util.cpp:87	Confirmed
69	Heap overflow	webasm [53]	2281ac9	load_module / wa.cpp:1810	Confirmed
70	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:667	Confirmed
71	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:996	Confirmed
73	Heap overflow	webasm [53]	2281ac9	setup_call / wa.cpp:617	Confirmed
74	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:1067	Confirmed
75	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:1006	Confirmed
76	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:903	Confirmed
77	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:998	Confirmed
78	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:1011	Confirmed
79	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:992	Confirmed
80	Heap overflow	webasm [53]	2281ac9	interpret / wa.cpp:1016	Confirmed

TABLE 6: List of bugs found by FUZZSGX