SGXFUZZ: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing

Tobias Cloosters
tobias.cloosters@uni-due.de
University of Duisburg-Essen, Germany

Johannes Willbold
johannes.willbold@rub.de
Ruhr-Universität Bochum, Germany

Thorsten Holz
holz@cispa.de
CISPA Helmholtz Center for Information Security

Lucas Davi
lucas.davi@uni-due.de
University of Duisburg-Essen, Germany

Abstract

Intel's Software Guard Extensions (SGX) provide a non-introspectable trusted execution environment (TEE) to protect security-critical code from a potentially malicious OS. This protection can only be effective if the individual enclaves are secure, which is already challenging in regular software, and this becomes even more difficult for enclaves as the entire environment is potentially malicious. As such, many enclaves expose common vulnerabilities, e.g., memory corruption and SGX-specific vulnerabilities like null-pointer dereferences. While fuzzing is a popular technique to assess the security of software, dynamically analyzing enclaves is challenging as enclaves are meant to be non-introspectable. Further, they expect an allocated multi-pointer structure as input instead of a plain buffer.

In this paper, we present SGXFUZZ, a coverage-guided fuzzer that introduces a novel binary input structure synthesis method to expose enclave vulnerabilities even without source-code access. To obtain code coverage feedback from enclaves, we show how to extract enclave code from distribution formats. We also present an enclave runner that allows execution of the extracted enclave code as a user-space application at native speed, while emulating all relevant environment interactions of the enclave. We use this setup to fuzz enclaves using a state-of-the-art snapshot fuzzing engine that deploys our novel structure synthesis stage. This stage synthesizes multi-layer pointer structures and size fields incrementally on-the-fly based on fault signals. Furthermore, it matches the expected input format of the enclave without any prior knowledge. We evaluate our approach on 30 open- and closed-source enclaves and found a total of 79 new bugs and vulnerabilities.

1 Introduction

The concept of trusted execution environments (TEEs) and Intel's implementation called *Software Guard Extensions* (SGX) have recently seen broad research attention [1, 12, 13, 17, 41, 50, 61, 77] as well as increasing industry adoption [4, 32, 57]. Generally speaking, SGX provides hardware-based features to build enclaves that can execute security-critical code *without* the interference of a potentially malicious OS. These enclaves run inside a host application and should even remain secure after the host is compromised. These properties make SGX enclaves an attractive choice for potentially hostile environments, such as cloud environments or desktop platforms, where malicious users can potentially interfere with the system.

In the past years, several attacks against the SGX framework were demonstrated, mainly through side channel attacks [12, 13, 41, 61, 70]. While these attacks compromise the security of SGX enclaves in general, these issues have practical limitations and can usually be solved by Intel in the long run. However, individual enclaves may suffer from software vulnerabilities that compromise the enclave's confidentiality and integrity regardless of whether platform attacks succeed. Thus, while the security of the SGX platform is essential, software security of individual enclaves must be ensured in either case. Unfortunately, this aspect has largely been overlooked by research and an in-depth analysis is necessary, because in addition to common types of software vulnerabilities (e.g., buffer overflows, use-after-free vulnerabilities, type confusion), enclaves have to ensure that all trusted computations are performed inside the dedicated and trusted memory region as only memory in this region is secured through SGX. Thus, all input from a potential attacker must go through input sanitization; especially pointers must be handled with care (e.g., pointers to input buffers must point to memory outside the enclave). Afterwards, the data has to be copied through the trust boundary and sanitized within the enclave. This trust boundary between the enclave's host application and the associated enclave lead to many vulnerabilities in the past [17, 77].

TEEREX [17] was recently proposed to automate vulnerability discovery at the host-to-enclave boundary by means of symbolic execution. It is the first tool which is capable of analyzing enclave code and successfully detected novel vulnerabilities in multiple enclaves. On the downside, the approach suffers from path explosion and can only focus on specific parts of the enclave code and approximate (or even exclude) others. In contrast, a well-suited analysis approach to examine the security of a given system is fuzz testing (short: *fuzzing*). This method can be used for open- and closed-source applications to enable out-of-the-box analysis with no prior knowledge of system internals. Fuzzing is an established and robust method with a proven track record of discovered vulnerabilities and a very active research area [2, 10, 14, 28, 40, 48, 66, 68, 73, 81, 85]. This type of testing, and more specifically so-called greybox fuzzing, utilizes coverage feedback from the system under test and mutates the input to maximize code coverage. However, vanilla greybox fuzzing tools like AFL and related approaches [10, 82] cannot be used for enclaves, as they provide their input as a plain linear buffer. In contrast, enclaves expect their input to be a structure consisting of several arguments and pointers, which are possibly nested and contain again pointers to other pointer structures. Thus, fuzzing a single enclave call (ECALL) using a vanilla fuzzer would require detailed knowledge about the layout of the input structure and a manually written harness. Obtaining these structure layouts is feasible for open-source code by reading the type definitions in C code, albeit not trivial due to compiler-generated or nested/inherited fields. However, this knowledge requirement becomes an issue with closed-source binaries, where in-depth reverse engineering would be needed. Previous work on the synthesis of input formats for fuzzing focuses on the *purpose* of specific fields in *linear* input or to identify logic structures in the input—known as grammar-based fuzzers [9, 22, 28]. Other approaches depend on source code [46] or domain knowledge [51, 78]. Another integral part for fuzzers is the code-coverage feedback that is essential to efficiently cover large parts of the code. However, coverage cannot be measured directly in the trusted execution environment of enclaves, which is designed to be unobservable.

In this paper, we address these challenges and present the design and implementation of SGXFUZZ, a novel method to efficiently fuzz SGX enclaves by synthesizing nested input structures. In a first step, we show how to extract the memory from an enclave to obtain the actual enclave code. To this end, we introduce changes to the SGX drivers for Linux and Windows to automatically dump the enclave's code upon executing it. In the second step, we introduce a method to execute the previously obtained memory dump of the enclave *without* the need for SGX-capable hardware. We use the SIGILL signal to detect SGX' `ENCLU` instruction, which would normally perform SGX actions (i.e., a context switch to the trusted SGX context). To handle this, we set up our context that mimics the SGX context and should not be distinguishable for regular enclaves. This approach allows us to execute enclaves natively, without emulation overhead, outside the SGX environment.

To automatically fuzz the enclaves in our execution environment, we propose fuzzer extensions to test arbitrary enclaves without any prior knowledge of the input structures. The first extension utilizes fault signals from specifically crafted memory pages to incrementally learn the layout of the input structures and to uncover possible size fields of variable-sized arrays. The second extension probes different types of pointer in the previously determined input layout. More specifically, it tests whether pointers inside, outside, or on the enclave's memory boundary lead to distinctive code coverage, which lets us uncover software faults related to the trusted memory region of SGX.

We implement a prototype of the proposed approach and evaluate our fuzzing extension using 30 open- and closed-source enclaves. In total, we uncovered 79 new bugs and vulnerabilities. As we show in detail in Section 3 and Section 6.5, SGXFUZZ outperforms TEEREX as it discovers significantly more vulnerabilities and higher code coverage.

Contributions. In summary, we make the following contributions in this paper:

Enclave Dumping: Using the SGX driver, we completely extract executable enclave code on Windows and Linux in an automated way.

Enclave Runner: We show how to execute enclave code, including SGX-specific instructions, natively outside SGX to retrieve code coverage feedback.

Structure Synthesis: We present a method to synthesize multi-layer structures of pointers, arrays, and size fields using fuzzing.

Memory Location Havoc: To analyze whether enclaves are vulnerable to trust-boundary attacks, we develop a fuzzing stage that tries different memory locations for each of the pointers in the synthesized structure.

Fuzz Analysis: Utilizing the methods developed in this work, we analyze the security of 30 enclaves and found 79 new bugs and vulnerabilities. So far, a total of three CVEs were assigned and $13k in bug bounty were paid for the vulnerabilities we identified.

Limitations of Symbolic Execution: We analyze previous work on vulnerability detection using symbolic execution, discuss limitations, and extensively compare the results with SGXFUZZ.

2 Background

The Intel Software Guard Extensions (SGX) [45] are a prominent technique to create self-contained libraries that can be executed within a secure context, unobservable by anyone except the enclave's author. However, this neglects possible vulnerabilities, which makes security testing a necessity. This section provides background information on SGX, the enclave structures, and the calling convention of enclaves to explain the starting point of SGXFUZZ. Thereafter, we summarize the principles of greybox fuzzing and the modules that provide the foundation for SGXFUZZ.

2.1 Intel Software Guard Extensions (SGX)

SGX is an extension set that provides instructions on Intel CPUs to create encrypted and protected memory and code sections that can be called using a controlled interface. Since the initial state of an enclave is cryptographically verified and the integrity of the hardware can be attested remotely, this allows trusted execution on remote hardware.

Enclave Low-Level Interface. Initialization of an enclave is done at kernel level by copying the data provided by the enclave developer into memory pages from the *Enclave Page Cache (EPC)*. Next, the memory pages of the enclave are added to the virtual address space of the *host application*. Thereafter, the initial state is cryptographically verified. At this stage only the enclave itself can access its memory. The memory of the host application remains unrestricted and can also be accessed from the enclave. Further, enclaves define a public interface for function calls (ECALLs) from the host application which is specified in the *Thread Control Structures (TCS)*. These structures specify the only addresses at which execution in an enclave can start. Moreover, TCS are locked and can only be used by a single thread to enter the enclave, which limits the concurrency within enclaves.

Entering an enclave (i.e., switching to the secure context) is implemented in the *EENTER* leaf function of the `ENCLU` instruction. *EENTER* locks a given TCS, enables the secure processing mode, and transfers the execution context to the defined entry point. When the enclave finished the execution of the call, the execution context is transferred back using *EEXIT*. In detail, *EENTER* affects several registers, e.g., `RAX`, `RBX` and `RCX` are used as arguments, `RSP` and `RBP` are saved (and restored on *EEXIT*), and `FS` and `GS` are set according to the TCS. The remaining registers like `RDI` and `RSI` are unaffected by *EENTER* and used to pass user-defined arguments to the enclave. Since the registers are very limited in number and size, enclave developers usually use an SDK that provides copy stubs for more complex data types. These SDKs use the registers to pass a pointer to shared memory, which is then used to copy the actual arguments of the ECALLs. The next paragraph describes the calling convention of the Intel SGX SDK.

Enclave High-Level Interface (Intel SGX SDK). The Intel SGX SDK [42] provides stubs to call enclave functions with complex arguments using a single TCS. It uses `RDI` to indicate the enclave function (ECALL) and `RSI` as the pointer to a data structure that contains all arguments. The stubs validate the pointers and copy all data into secure memory. Next, the actual enclave function is invoked using these secure copies. The SDK also provides *outside calls (OCALLs)* that act in the reverse: The enclave is left, a specified function in the host application is executed, and the return values are copied into the enclave. In addition, the SDK adds another internal ECALL for initialization, that is usually executed right after the instantiation of an enclave to, e.g., setup global objects.

The SDK translates ECALL signatures into structures as follows:

- A function without arguments (`void ecall()`) just uses a `nullptr` instead of a struct pointer.
- Return values and primitive types are packed into the struct: `int ecall(int arg)`
  $\rightarrow$ `struct ms_ecall_t { int ret; int arg; }`
- Buffers are combined with a size field:
  `void ecall([size=buf_s] char* buf, size_t buf_s)`
  $\rightarrow$ `struct ms_ecall_t { char* buf; size_t buf_s; }`
- For C-strings, an internal size field is added that is checked against `strlen` in the enclave:
  `void ecall([string] char* buf)`
  $\rightarrow$ `struct ms_ecall_t { char* buf; size_t len; }`

Additionally, user types may be passed unchecked using a raw pointer (`void ecall([user_check] void* ptr)`), but this easily leads to vulnerabilities.

Memory Corruption Vulnerabilities. In this paper, we focus on memory corruption attacks [74] against software running inside an SGX enclave. In general, these attacks are possible due to programming errors that allow an attacker to perform malicious reads and writes to application memory, e.g., overwriting data on the stack or the heap. These attacks have devastating consequences for security-critical SGX software as they often allow extraction of cryptographic keys stored in enclave memory [53]. Furthermore, SGX enclaves developed with the Intel SGX SDK offer a large attack surface for launching return-oriented programming attacks [69]—the state-of-the-art memory corruption attack technique—as they provide a large number of powerful gadgets (i.e., allowing access to many CPU registers) that are located at fixed memory locations [8]. Recent studies confirm that invoking these gadgets is highly probable, as public SGX frameworks and SGX enclaves including commercial fingerprint driver software suffer from a variety of memory corruption vulnerabilities [17, 77]. One important lesson from existing vulnerability studies in SGX software is the high probability of null-pointer-dereference errors. While these play a negligible role in modern PC software, they are highly relevant in the context of SGX as the underlying SGX adversary model assumes an attacker that controls the kernel, thereby allowing memory allocation at arbitrary addresses (including the `NULL` page). We will come back to these errors when analyzing real-world enclaves in Section 6.

2.2 Greybox Fuzzing

Fuzzing is an automated software testing method where random input is used to test a target program for potential software faults. Depending on the target's execution result, the fuzzer mutates the input to explore previously unseen code paths. This way, the fuzzer aims to maximize the total code coverage achieved during the entire fuzzing process.

Fuzzers are separated into different flavors depending on their ability to retrieve information about the target's execution run. Greybox fuzzers have no access to the target's source code, but utilize code coverage feedback to monitor whether the target executed a new code path. The feedback is generated through instrumentation [25, 54, 65], emulation [29, 58, 59], or hardware extension like Intel PT [16, 52, 56, 62, 66, 67, 84]; the feedback is usually recorded in a bitmap.

Greybox fuzzers use the coverage feedback to deploy a fuzzing loop, where they generate an input, pass it to the target, and record the code coverage [11, 14, 64, 82]. After each execution, the fuzzer analyzes the coverage bitmap to see whether a new code path was discovered. Inputs that generate novel coverage are saved and used as a base to start new fuzzing iterations. A fuzzing iteration consists of several stages, where each stage usually deploys a custom input mutator and executes the target several times to achieve new coverage based on the stage's mutator [3, 9]. This combines the sophistication of different mutators to tackle specific code paths in a target with the randomness of basic fuzzing strategies.

Snapshot Fuzzing. Fuzzers test targets repeatedly with generated inputs and aim at achieving high efficiency. This requires an efficient technique to reset targets to the state when they accept the input. On Linux, this is usually implemented using a fork server that forks the target process just before receiving the fuzzing payload [14, 36, 81, 82]. Hence, the target process does not have to be created from scratch for each execution.

Snapshot-based fuzzers [66, 72, 79] use a different strategy: these fuzzers create a complete snapshot of the target process that contains the target's memory and execution context. During the execution of the target, the fuzzer tracks which pages of the snapshot are changed and then reloads only these changed pages upon starting a new execution. This strategy leads to higher execution rates compared to the fork-server approach and can efficiently reset the state of the entire program (or enclave) after each execution.

3 Fuzzing vs. Symbolic Execution: Comparison to TEEREX

Aside from fuzzing there are other techniques for vulnerability detection like the TEEREX framework [17] which uses symbolic execution and is specialized to find vulnerabilities in SGX enclaves. TEEREX can extract the ECALLs from enclaves and symbolically calculate vulnerable states. However, TEEREX suffers from several issues that lead to false positives and false negatives. We evaluate these issues based on case studies and describe how they reveal the limitations of the symbolic execution approach. We start with a brief description of conceptional differences between fuzzing and symbolic execution, which shows the inherent limitations.

3.1 Conceptual Differences

First, there are the fundamental advantages of fuzzing and symbolic execution, respectively. *Fuzzing* provides complete and reproducible test cases with a low false positive rate. However, every test case starts at the entry point and tests a full execution path. Further, fuzzers have to test identical code paths multiple times until they find a new path because the exact relation between input bytes and branches taken is unknown.

Symbolic Execution, on the other hand, is a comprehensive approach to explore all possible program states in a single analysis. However, a complete analysis of non-negligible code sizes is infeasible due to the amount of possible states. Hence, practical symbolic execution approaches focus on specific parts and approximate or exclude others. Moreover, there are conceptional no-ops in programs that introduce high complexity for a symbolic solver. For example, SGX enclaves create secure copies of input using `malloc` and `memcpy`. While this hardly affects possible memory corruptions, these memory management functions consist of many read/write instructions that impose a burden for the symbolic memory. As a countermeasure, symbolic execution engines try to replace these functions with mocks that are optimized for symbolic engines, but this is not trivial for stripped closed-source binaries, whereas the overhead of these functions is negligible for a fuzzer. As a result, it is difficult for TEEREX to analyze ECALLs with complex input structures in stripped closed-source binaries.

3.2 Limitations of TEEREX

TEEREX aims to find a symbolic connection between the arguments of an ECALL and executed read, write, and jump/call instructions. Thereafter, it analyzes the context and determines if the conditions on the state allow unintended destinations. When argument data is used as an address without further check, this is an indicator for a vulnerability.

This analysis depends on soundness and (relative) completeness of the symbolic analysis. This section discusses cases when approximations and optimizations lead to false reports.

3.2.1 False Negatives

First, we analyze reports of TEEREX to discuss sources of false negatives caused by the symbolic execution approach.

Case Study: Heap Memory Corruption. During our evaluation, we discovered three bugs that corrupt the heap memory by writing beyond the allocated memory. In each of the cases, bytes are written linearly, so that it is not possible to write to arbitrary memory locations. In general, these bugs are hard to detect by a greybox fuzzer because memory validators like ASAN [34] cannot be added after compilation.

However, Intel's trusted library for SGX utilizes a custom implementation of `malloc` and `free` because system functions are not available in enclaves. This implementation stores a header consisting of a magic value and the size of an allocated memory chunk prior to the actual memory. The magic value is a 64-bit random value, generated on the first call of `malloc` and is equal for all chunks. The corresponding `free` function checks this magic value, reads the chunk size, and then reads the magic number following that chunk. If either of them is invalid, the function terminates with a segmentation fault. Thus, when a bug corrupts heap memory, this magic number is destroyed and the program crashes on the subsequent call of `free` or `malloc`. In addition, `free` crashes on any pointer not allocated by `malloc`, as the magic number is missing. This mechanism provides the fuzzer with an excellent bug oracle for heap memory corruption and freeing invalid (e.g., uninitialized) pointers. On the other hand, this scenario is challenging for symbolic execution:

Incomplete model. When symbols are available, TEEREX and in particular the underlying framework ANGR, uses symbolic replacements for standard functions to reduce the path complexity. Specifically, the replacement for `malloc` always returns a new chunk of heap memory and `free` is a no-op which does nothing and always succeeds. In this heap model, the oracle is not available and this kind of bug cannot be found.

State/Constraint Explosion. When symbols are not available, TEEREX cannot replace the memory management functions as described above. Consequently, the checks of the SDK's `malloc` are still present, but functions like `memcpy` add more constraints to the symbolic state than the solver can handle. This is particularly complex when the size argument is symbolic. As a result, the storage for each state as well as the evaluation time for branch decisions increases significantly. This forces TEEREX to abort exploration of these paths early and miss potential bugs.

3.2.2 False Positives

Second, we sample the results reported by TEEREX to find common sources for false positives. However, the number of reports make an exhaustive evaluation infeasible.

Case Study: Pointer Range Checks. The results for the enclaves sgxwallet and BiORAM-SGX include reports for instructions in the ECALL wrapper that copy data into `[out]` buffers. These out buffers (destination addresses) are attacker-controlled, however, the SDK wrapper ensures that out buffers are outside the enclave.

Incomplete Address-Space Model. TEEREX emulates enclaves without the clear separation of the address space of the host application. In addition, ECALL arguments are fully symbolic so that concrete values for nested addresses may be within as well as outside the enclave. When TEEREX encounters a write to an attacker-controlled address, it may conclude that this is a vulnerability, although the address was range-checked. This is especially the case for out buffers, which are a common concept of enclaves, where an enclave writes to a user-defined non-secure address by design. This leads to false positives because these addresses usually cannot be used to manipulate trusted memory. As a countermeasure, TEEREX tracks the range checks for pointers of the SDK, but cannot automatically infer whether these constraints make the pointer in question a false positive. Additionally, this pointer tracking technique requires symbols and thus does not aid for the analysis of closed-source enclaves.

4 SGXFUZZ Architecture

The high-level view of our fuzzing architecture is visualized in Figure 1. This section introduces the main components. We first show how to execute enclaves outside SGX while still maintaining memory separation and context switches during the enclave execution as part of our runner. In the second step, we show the design of our fuzzing setup to synthesize input structure layouts incrementally using signals. Thereafter, we describe our novel mutation stages that are specialized to find vulnerabilities in SGX enclaves. The implementation details of each component will be described in Section 5.

Enclave Dumper (cf. 5.1). We first extract the enclave's memory from the enclave's distribution format (shared library file, `.so`). Our enclave dumper saves the memory (code and data), and additional metadata, i.e., the enclave's memory layout, page permissions, and the entry points (TCS). We compile the enclave data into our enclave runner, which is a regular executable that acts as a harness around the enclave.

Native Enclave Runner (cf. 5.2). The enclave runner (or harness) is capable of executing any ECALL outside SGX using the dumped enclave, a serialized structure description, and the payload bytes. First, the harness initializes the simulated SGX environment: It installs handlers for the SGX instructions and prepares memory and registers for the *EENTER* call. Then, the runner reads the input structure definition and the payload—usually from the fuzzer during the fuzzing loop—and rebuilds its representation in memory. That is, it allocates all necessary buffers, places the addresses of nested buffers in the parent buffers, and fills the remaining fields with the payload. Thereafter, the given ECALL is invoked using this structure as the argument. We simulate the context switch to the SGX environment, jump to the ECALL entry, and the execution of the enclave continues as programmed by its developer. When the enclave code encounters one of the SGX-specific functions, the CPU interrupts because it is not in SGX-mode, and the harness is notified via a signal (SIGILL). We emulate this instruction and resume execution right after that instruction. If the execution encounters a crash, e.g., a segmentation fault, the harness collects the fault information and submits those to the fuzzer for the structure synthesis.

This runner enables us to execute any enclave natively in normal user-space without any SGX restrictions or emulation overhead. In addition, we receive detailed feedback about crash reasons, thereby enabling our structure synthesis.

SGX Structure Fuzzer/Synthesis (cf. 5.3). The novel fuzzing stage for structures uses the signals emitted during the enclave's execution to generate and adapt the layouts of the structures. We detect missing fields and incrementally

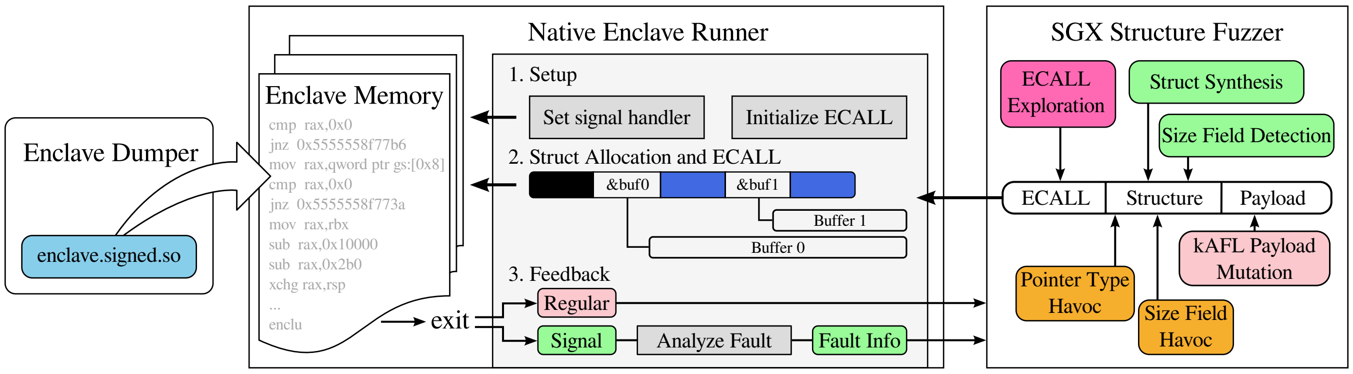


Figure 1: SGXFUZZ Architecture.

recover the expected layout of the input. In detail, whenever the fuzzer finds a new input that causes a signal (SIGSEGV), the fuzzer checks whether a struct mutation can resolve the crash, i.e., it examines whether the enclave interprets part of the payload as pointer. If this is the case, it alters the structure layout for subsequent fuzzing runs, so that the enclave will find a pointer/sub-buffer at that location. To further improve the detection of bugs specific to SGX and the structure-based fuzzing approach, the fuzzer also tests whether the execution path diverts when the pointers of child buffers are within special regions like the enclave's memory.

5 SGXFUZZ Implementation

We now provide details of our prototype implementation of SGXFUZZ. We first describe the enclave dumper, followed by a description of the enclave runner. Finally, we discuss our fuzzer extensions, implemented on top of the kAFL fuzzer [67] and the Nyx snapshot-fuzzing engine [66].

5.1 Extracting Enclaves from Distribution Formats

In the fuzzing process, we want to match the execution of the original SGX hardware as closely as possible. Therefore, we need a dump of the enclave's memory to load it into our fuzzing harness. However, most enclaves are not distributed as raw memory dumps but packed into a container format like ELF (`.so`) or PE (`.dll`). We develop different methods for extracting the memory of enclaves precisely as the hardware during initialization. In this process, we also collect all entry points (TCS) and memory permissions within the enclaves to recreate the exact application setup in our fuzzing harness.

SGX uses at runtime a continuous block of secure memory (EPC) [20] that contains all sections needed to run the secure application, most notably, the code section and sections for the stack and the heap data. The Intel SGX SDK uses the ELF/PE format to store enclaves and loading is modeled after common application binaries. However, compared to the default OS loader, there are some important differences for loading enclaves: (1) Enclaves do not contain gaps, therefore, the loader cannot use the default address ranges for, e.g., the stack, (2) host applications (compiled using the Intel SGX SDK) apply some patches during load time, e.g., to address relocation sections, (3) the exact loading behavior may differ depending on the host application, SDKs, or SDK versions, because they are not technically bound to a standard.

First, we modify the driver module of SGX for Linux [44] to dump every loaded enclave. This is a reliable way to extract the exact memory that the hardware uses for initialization and is independent of the distribution format of the enclave, the host application, and any framework used. Since this method requires SGX-capable hardware, we develop a second approach to automate fuzzing of SGX enclaves in, e.g., Continuous Integration (CI) pipelines on non-capable server hardware: The SGX SDK comes with a signing tool [42] that calculates a hash over the enclave memory to create the signing data. This hash requires the enclave to be unpacked, thus we can dump the memory during this calculation. This is a fast and hardware-independent method to dump the memory, but is only applicable to the SDK's enclave format and there is a chance that the result contains erroneous bytes due to version differences. However, it produces correct results when the SDK versions match, and we have only seen issues when the enclave was compiled using an SDK version several years older than the dumper's version. In that case, we could back-port the dumper to the older SDK version to correctly dump these enclaves.

Dumping enclaves for Windows is hindered because the driver and the signing tool are closed-source and cannot be patched as easily. However, we implement automatic extraction of Windows enclaves using a debugger and specific breakpoints in the signing tool. Alternatively, we can enable the debug mode of enclaves and extract the memory after loading, which requires some manual work. Both methods are rather slow and cumbersome, but they produce valid memory dumps despite the tools being closed-source.

5.2 Fuzzing Harness: Running Hardware Enclave Binaries in Normal User-Space

Fuzzing SGX enclaves requires an *introspectable* environment that can *quickly* execute ECALLs and return to the *initial state* for the subsequent execution. These requirements make fuzzing using SGX hardware impracticable for several reasons. First, SGX enclaves are meant to be *not* introspectable. There are approaches to collect coverage data using side channels [23], however, they are slow and yield far less information than using debugging features. Further, we could execute the enclaves in (hardware) debug mode and use the special enclave debugging instructions to extract coverage data, but the capabilities of these instructions to produce coverage data are limited and comparably slow. Second, the protections of SGX hinder the state reset that is required for efficient fuzzing. SGX enclaves cannot be forked, so to run a second test case using a real enclave, the secure memory has to be allocated and initialized from scratch, including the enclave's initial measurement. Alternatively, it may be possible to reset the state of an enclave using the debugging instructions. However, the debugging instructions cannot reset whole pages, so a slow loop would be needed to reset the whole enclave. In practice, both approaches are too slow for fuzzing. Third, the secure memory (EPC) is limited to less than 128 MB [26] on common CPUs, which is the available memory for all enclaves and imposes a hard limit on the number of enclaves that can be in memory simultaneously. Therefore, a high performance CPU will quickly surpass this limit when trying to create an enclave for every core for efficient parallelized fuzz testing.

We develop an interrupt-based runtime that can run a hardware-targeted enclave as a regular user-space application. This enables a fuzzer to record code coverage feedback through state-of-the-art greybox fuzzing techniques like Intel PT [52]. While Intel PT cannot record genuine enclaves, it is possible using our approach as we never actually switch to the secure context, but instead execute all enclave code in normal user-space.

In detail, our runner executes SGX enclaves as follows: At first, when the runner is started, it initializes a continuous block of memory with the enclave's memory in its own address space and applies the memory permissions as recorded by the dumper. Then it executes the SDK's special initialization ECALL as the normal enclave creation would do. Thereafter, the runner reads the input from the fuzzer and allocates memory for the input structure according to the serialized layout specification. Thereby, every buffer is placed at the end of a memory page to be followed directly by an inaccessible guard page as depicted in Figure 2. This enables the fault detection required by the structure synthesis (cf. Section 5.3.1). When (nested) child pointers are present, those are allocated analogously and the addresses are placed into the parent buffers. Similar, the values for size fields are calculated and inserted into the respective buffers. The remaining fields of the structure are filled with the fuzzing payload. Then the runner

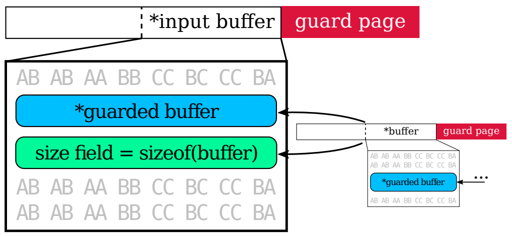
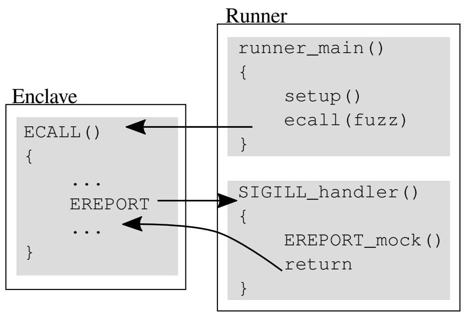


Figure 2: Allocation scheme of the fuzzing harness. Every buffer is followed by a guard pages that enables to detect the buffer sizes using fault signals.



```
ECALL()
{
    ...
    EREPORT
    ...
}
```

```
runner_main()
{
    setup()
    ecall(fuzz)
}

SIGILL_handler()
{
    EREPORT_mock()
    return
}
```

Figure 3: SIGILL-handler to emulate special SGX instructions.

can execute a given ECALL by jumping to the entry point defined in the TCS using the structure as argument.

Whenever the execution reaches an SGX instruction, a SIGILL signal is emitted and handled by the runner to emulate the SGX instruction. The runtime reads the encountered instruction, adjusts the register state accordingly (e.g., generate fake keys or change memory permission), and resumes the execution (Figure 3). When an *EEXIT* instruction is reached, which also triggers the signal, the runner terminates.

When the runner receives a memory error (segmentation fault), it uses Zydis [86] to disassemble the faulting instruction and calculate the memory address that caused the error, which is reported to the fuzzer for struct-synthesis (Section 5.3.1).

5.3 SGX Structure Fuzzer

We base our implementation of SGXFuzz on the kAFL fuzzer [67], which is a feedback-driven greybox fuzzer that utilizes Intel Processor Trace (PT) [52] for coverage feedback. In addition to the classic AFL mutation stages [82], this implementation of kAFL features a radamsa [39] mutation stage and the Redqueen [3] input mutator, which is considered as the state-of-the-art solution to tackle common branch constraints.

The fuzzer collects coverage feedback from Intel PT in an AFL-like bitmap using modified versions of QEMU/KVM to execute the targets. In this setup, the targets are not emulated, but run natively on our fuzzing hardware, which significantly reduces the fuzzing overhead. The snapshot engine of Nyx [66] initializes the targets in KVM and creates a snapshot of the complete state of the fuzzing target right before processing the fuzzing input, which enables an efficient state reset. The snapshot engine replaces the fork-server scheme that is commonly used in AFL or AFL++ [29]. We choose Nyx because it provides state-of-the-art fuzzing performance and significantly outperforms [66] QEMU-based fuzzing.

5.3.1 Structure Synthesis

We develop a custom fuzzer stage that incrementally and dynamically synthesizes the layouts of input structures. It is based on error signals and uses guard pages as depicted in Figure 2. Structures are allocated in a way that every input is followed by an inaccessible guard page, which ensures that the fuzzer is notified, when the enclave tries to access more data than provided—beyond the end of a buffer. The fuzzer starts with an empty initial structure, that gradually evolves during the fuzzing towards the expected complete input layout. Since layouts may depend on the values within (e.g., type fields in union types), the fuzzer tracks layouts tied to the input where it was found, so that diverse layouts may evolve for different inputs/paths in a target.

We will now describe the synthesis process in detail. The structure synthesis is a stage of the fuzzing process like other input mutation stages that every input traverses. When an input and layout from a previous stage reached the structure synthesis, it checks whether this input causes the target to exit abnormally, with a SIGSEGV signal. In that case, the target attempted to access an invalid memory reference, either due to an incomplete structure or due to a bug.

Our structure synthesis then collects all necessary addresses from the target and checks whether the faulting location is within one of the guard pages. This is a strong signal that the provided input buffer is too small. Therefore, the synthesis stage increases the size of this buffer gradually as long as a fault is reported within this guard page. Figure 4 shows these faults and the influence on the structure. Usually, the final length of the buffer cannot be deducted directly from the first report because most enclaves use a linear copy function to copy the buffer into secure memory, so that the error location is always at the beginning of the guard page. Newly allocated bytes of increased buffers are filled with random bytes.

As a second case, the fault may not be in a guard page, but contained as a value in one of the input buffers. This indicates that the target expected the value to be a pointer to another buffer—and the synthesis extends the layout with a new buffer to make this value a pointer (see Figure 4). The synthesis also tolerates small offsets between the faulting address and candidate values in the structure to identify cases where not the first byte of the missing buffer is referenced but another field. We chose `0x100` as a reasonable maximum offset that will usually identify the correct value but is small enough to not lead to significant false positives due to randomly similar unrelated values. However, it might lead to incomplete structures for targets that read (only) a field, e.g., at offset `0x110` (cf. Section 6.3). Further, for newly added pointers, the size of the corresponding buffer is still unknown. Therefore, it is initialized with a size of zero and the fuzzer executes the target again. The target will now find a pointer where it previously found a non-pointer value. Accessing this pointer will still trigger a fault, but this time it will be within the guard page of the newly added pointer and it can be increased as described above.

5.3.2 Size Field Detection

Input structures often contain a field for the size of variable-length buffers because dynamic length calculation on untrusted data using functions like `strlen` is prone to introduce information leakage [77]. Hence, the size should either be constant or provided with the buffer until the data is copied to secure memory. If the intention was to pass a NULL-terminated string, the length should only be validated afterwards, using the secure copy.

In the context of fuzzing, size fields that are included in the buffer, are initially also filled with (random) fuzzing payload. In most cases the resulting value is far from the actual buffer size and input becomes cropped or is rejected early and the fuzzer can hardly pass the copy functions at the beginning of many ECALLs. To counteract this, we search for size fields whenever the buffer layout is changed in the struct synthesis stage. When this is successful and size fields are found, they are marked in the struct layout and set to the size of the corresponding buffer rather than filled with fuzzing input.

In particular, the size field detection is triggered, when the target reports a fault in a guard page which cannot be fixed by a small increment of the buffer. This indicates that the size of the buffer is not limited by a (reasonably small) constant number, but by other means like a large integer originating from random bytes in the fuzzing input. To find the position of the size field in the structure, the detection sequentially inserts the actual size of the buffer at all offsets of the fuzzing payload. If at one offset the fault is resolved (or a different unrelated fault), this indicates that this value limited the `memcpy` causing the error. Next, the offset is verified by executing the same test again, but with the value of $size+1$, which is expected to produce the same fault position as observed initially if the offset is indeed the correct size-field of the buffer. An offset that passes both checks is inserted into the struct layout and the subsequent fuzz testing will use the (static) size of the buffer at this offset instead of filling it with fuzzing input. While the relation of a dynamically-sized buffer is ensured during the fuzz runs, it may still be too small for the semantics of the enclave. However, enclaves using size fields will just compare the size value and buffers are not increased due to the guard-page method. We tackle this in a later stage that tests whether different buffer size yield additional coverage. The stage that we call *Size Field Havoc* sequentially adjusts the size of buffers that are linked to a size field to test if specific sizes yield unique coverage. We chose to test sizes up to 256 bytes as this is a reasonable size for buffers

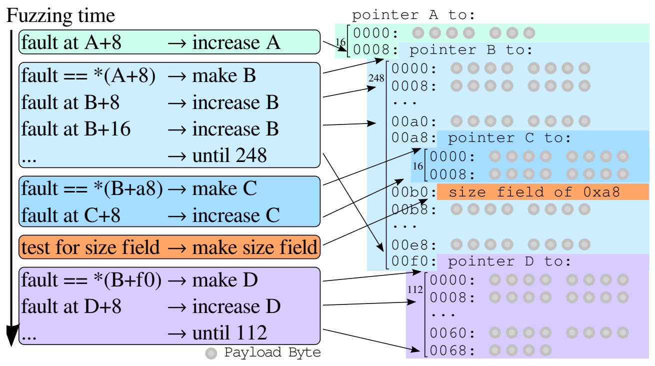


Figure 4: Struct Synthesis Events during fuzzing (left) and the resulting changes to the nested pointer structure (right).

in the struct layouts and it is small enough that subsequent input mutation stages have a fair chance to mutate impactful bytes.

5.4 Fuzzing Edge Cases and SGX-Specific Extensions

In addition to the structure synthesis, we implement features to tackle edge cases that hinder fuzzing of some enclaves and add SGX-specific bug oracles.

ECALL Exploration. The ECALL interface specifies two arguments, the index of the ECALL in the ECALL table and the (pointer to the) argument structure. Since our fuzzing stages only process the latter, and the index is not part of the structure, the index is not mutated in these stages. Therefore, we initially explore the ECALLs offered by the enclave and test each ECALL ID from 0 to 255 whether it contributes new coverage, i.e., is an existing ECALL number. This yields an initial payload for every ECALL from which the structure recovery can proceed. While it is supported that an enclave defines more than 256 ECALLs, we have yet to see one that actually does. In that case, we can easily increase the upper boundary.

Custom Initialization ECALL. During our evaluation, we encountered three enclaves that define a custom initialization call to set up a context, which is required to be called first. Typically, every other ECALL checks if the initialization has been called and immediately returns otherwise. Thus, meaningful fuzzing is not possible without including the initialization call. Our fuzzer supports to load and execute the layout and payload of a specific ECALL during the setup of the runner, before the kAFL/Nyx environment takes the snapshot and the fuzzing loop starts. This way, the initialization ECALL has zero overhead on the actual fuzzing runs, as it is only executed once during the setup. This feature makes the Signal enclaves fuzzable which require an initial constant call to *init*. This feature can be seen as a lightweight variant of *call chains* discussed in Section 7, which we consider future work.

Pointer Location Havoc. We implemented an SGX-specific bug oracle that we call *pointer location havoc* to search for bugs related to the confusion of memory regions within SGX enclaves. Enclaves usually specifically check and only accept pointers from the outside, i.e., non-SGX memory, using the dedicated functions `sgx_is_outside_enclave` and `sgx_is_within_enclave`. However, some enclaves also accept pointers within the range of the enclave—either because of an error or to explicitly reuse a previously allocated reference. Since this is a bad practice that often leads to vulnerabilities that compromise the enclave's integrity [17], we develop this oracle to find cases where the target accepts such dangerous pointers. We iteratively alter the synthesized structure layout and substitute the regular outside pointers with pointers from different regions. Specifically, we test *In-Enclave Pointers*, pointers within the range of the enclave's secure memory, and *Boundary Pointers*, pointers to the boundary between user-space and enclave, so that the last byte of the buffer is the first byte of the enclave. Boundary pointers can uncover errors that incorrectly attribute overlapping buffers to the secure memory region. Note that this process only changes the addresses of the pointers, whereas the content of the buffers stay the same. This does not violate the threat model of an enclave that accepts in-enclave pointers because an attacker can use a pointer to the secure copy of his own input to get a content-controlled in-enclave buffer.

We additionally use *inaccessible* pointers in the respective ranges to explicitly test whether an enclave actually uses these kinds of pointers or just preliminary rejects them based on a range check. For example, if a regular/outside pointer within a structure is inaccessible, the enclave will segfault upon using it. If the same fault is found for an inaccessible *inside pointer*, the enclave used this pointer without check and is probably vulnerable.

5.5 Post Processing

The novel structure recovery stage uses segmentation faults to detect incomplete structure layouts. However, this overlaps with the bug detection of the fuzzer that uses segmentation faults as bug indicator. This introduces edge cases where the fuzzing engine cannot clearly decide on-the-fly whether a crash is attributed to the structure synthesis or to the crash-based bug detection. These cases are stored in the fuzzing output.

We developed an automated post-fuzzing procedure in form of a script that uses the final fuzzing output to correctly identify miss-classified crashes that should have been originally attributed to the structure synthesis.

i. NULL-pointer Dereference: When the reported fault address is very small, there is a high probability that this is caused by a NULL-pointer dereference bug. Thus, we do not use addresses below `0x100` for struct recovery even if there is a suitable candidate value in the payload. The fuzzer will quickly find another test case with a larger value to trigger the struct recovery. During the post-processing, we automatically filter small fault values where a matching small payload value is present to identify real `nullptr` dereferences.

ii. Pointers in Payload: The fuzzer and especially the fuzzer's RedQueen extension can add valid pointers by chance at the correct offset in the payload. However, since they are not part of the generated layout, the contents behind these pointers is arbitrary and not filled with fuzzing payload. These are not reproducible and not meaningful to indicate bugs. The post-analysis can easily identify those because the fuzzer will also show a structure with this offset marked as a pointer.

iii. Large Pointer Offsets: Enclaves may access pointers using an offset larger than `0x100`, which currently is not used by our fuzzer for sub-buffer identification. However, incomplete structures due to this reason are rare, and our results show that we can synthesize structs with high accuracy (Section 6.3). On the other hand, increasing this value can cause false positives in the structure synthesis.

iv. Size Fields Prior to Detection: When not-yet-detected size fields are present in the structure, their values are determined from the fuzzing payload. When additionally the targets accesses this buffer non-linearly—e.g., it first accesses the last byte (`buffer+size`)—the structure synthesis will receive a fault at a (large) random offset and it cannot reasonably match the fault to a buffer. This fault is then rejected by the struct synthesis and generates an entry in the crash log. However, due to the fuzzer's speed and number of executions, it also finds payloads that correctly trigger the synthesis and size field detection. The post-analysis identifies these cases from the not yet final structure that is logged with the crash.

Filtering these types of false positives reduces the total amount of reported crashes from up to a few hundred to a suitable amount for manual analysis (cf. Table 3). We manually verified that all remaining crashes were caused by an actual software fault in the target.

6 Evaluation

In this section, we evaluate the SGXFUZZ prototype to validate our design choices. We first validate our choice of fuzzing runtime showing that the native snapshot-based fuzzing engine is superior to alternative emulation-based engines. We then run SGXFUZZ on a large set of real-world enclaves and evaluate the accuracy of the structure synthesis. We further compare the scalability and the achieved coverage of SGXFUZZ to TEEREX and show that our approach covers more basic blocks, does not suffer from state explosion, and reports no false positives.

Experimental Setup. We used two test benches for our evaluation: One server with two Intel Xeon Gold 6230 and 196 GB RAM, and a second server with two Intel Xeon Gold 6230R and also 196 GB RAM. We fuzzed each target for 24 h using 40 cores (resulting in 960 *core-hours* spent per target). We used the same initial seed for all fuzzing runs.

Fuzzing Targets. Table 1 shows the targets that we used for our fuzzing evaluation. We selected our enclaves from a broad spectrum ranging from production enclaves to research prototypes. Unfortunately, we could not include a password manager with SGX capability. The two password managers *Dashlane* and *1Password* (before v7.0) advertise SGX on their website, and *1Password* even distributes binaries labeled with SGX. However, we were not able to find any trace of actual enclave usage in both cases.

Responsible Disclosure. We disclosed our findings in a coordinated way to the authors and vendors of the tested enclaves. We sent them detailed reports and helped them to fix the identified software faults. Synaptics assigned CVE-2021-3675, and sgxwallet CVE-2021-36218 & CVE-2021-36219.

6.1 Results

In total, we found 79 vulnerabilities, of which three have been assigned CVEs and a bug bounty of $13k was issued. Table 1 shows the number of manually verified unique vulnerabilities for each fuzzing target and Table 2 shows the exact class of each vulnerability accordingly. Additionally, Table 3 includes coverage data and reports for SGXFUZZ and TEEREX, which will be discussed in the following sections.

Notably, we did not encounter any false positives in our evaluation of SGXFUZZ. We manually verified that each report is caused by an actual bug, only some bugs occurred duplicated, resulting in more reports than verified bugs.

Security Implications. Table 2 shows that a lot of vulnerabilities are classified as either null-pointer dereferences or uninitialized pointer usage. However, in the context of SGX they are not less severe than, e.g., a memory corruption. In the case of the Goodix Fingerprint Driver enclaves, we were able to craft an arbitrary read exploit with a single null pointer dereference. Unfortunately, the vendors of the Gingytech Fingerprint Driver and the Goodix Fingerprint Driver enclaves have no intention to fix the vulnerabilities even after providing a fully functional proof-of-concept exploit.

6.2 Runtime Considerations

We develop the first binary-compatible runtime for hardware-compiled SGX enclaves that does not require SGX hardware support. OpenSGX [47], a QEMU-based environment for SGX code, does not provide a suitable runtime for our fuzzing targets because it cannot execute enclaves compiled for SGX hardware, but requires a dedicated compiler and SDK setup. Thus, OpenSGX cannot run binary-only enclaves due to lacking binary compatibility. In addition, OpenSGX has been deprecated since 2016 and cannot run real-world enclave code.[1] Nevertheless, to give a rough baseline, we measure the execution speed of OpenSGX' hello-world example, which

[1] https://github.com/sslab-gatech/opensgx/issues/50

Enclave	Version	#ECALLs	#Execs	#Covered Basic Blocks	#Bugs	Struct Accuracy *(source code)*
BiORAM-SGX	`d86dab22dba12`	15	$6.8*10^9$	1802	0	Coverage Match
ELAN Fingerprint Driver	3.4.12210.10801	25	$7.0*10^9$	675	0	`closed-source`
ELAN Fingerprint Biometric SSL	3.4.12210.10801	29	$5.7*10^9$	2206	14	`closed-source`
Gingytech Fingerprint Driver	2.0.1712.0318	3	$4.6*10^9$	4080	16	`closed-source`
Goodix Fingerprint Driver Coating Enclave	2.1.145.103	30	$5.2*10^9$	1698	6	`closed-source`
Goodix Fingerprint Driver Glass Enclave	2.1.145.103	30	$4.7*10^9$	1736	8	`closed-source`
Goodix Fingerprint Driver WBDI Enclave	2.1.145.103	31	$6.2*10^9$	1404	14	`closed-source`
Intel AE Launch Enclave (LE)	2.13	2	$6.0*10^9$	429	0	Perfect Match
Intel AE Provisioning Cert. Enc. (PCE)	2.13	2	$9.8*10^9$	490	0	Perfect Match
Intel AE Provisioning Enclave (PVE)	2.13	2	$4.1*10^9$	452	0	Perfect Match
Intel AE Quoting Enclave (QE)	2.13	2	$5.9*10^9$	407	0	Coverage Match
Intel SDK Initialize ECALL	2.13	1	$3.1*10^9$	257	0	Coverage Match
KubeTEE TFF	`1c5ab9f5ca645`	3	$1.8*10^9$	3320	0	Perfect Match
Ledger BOLOS	`573464ed78354`	13	$7.4*10^9$	1076	0	Coverage Match
lockbox	`0c70a7d02c1b7`	17	$6.7*10^9$	3689	2	Coverage Match
MobileCoin	1.0.1	1	$5.3*10^9$	11888	0	Coverage Match
Occlum Runtime Libos	0.22.0	7	$7.1*10^9$	977	0	Coverage Match
OMEC Project's C3P0 – Dealer	`771c0c383c4d9`	6	$7.2*10^9$	226	1	Perfect Match
OMEC Project's C3P0 – KMS	`771c0c383c4d9`	4	$6.8*10^9$	1673	1	Perfect Match
Plinius	`31a51ff3d2d90`	7	$4.8*10^9$	646	2	Coverage Match
SGX Darknet	`0fe09ccb9aa62`	4	$6.1*10^9$	527	3	Coverage Match
sgxwallet	1.58.3	38	$2.6*10^9$	3865	2	missed output buffer
Signal Contact Discovery	1.13	7	$7.0*10^9$	560	0	Coverage Match
Signal Secure Value Recovery	1.0.20	7	$5.9*10^9$	11989	0	Coverage Match
STANlite	`16467c8034e84`	3	$2.6*10^9$	6621	4	Perfect Match
Synaptics Fingerprint Driver Enclave	5.2.3539.26	2	$6.7*10^9$	1416	1	`closed-source`
Tensorflow Lite	`5dc3b3d97844d`	1	$6.7*10^9$	492	0	Perfect Match
Town Crier	`33471ff56cb75`	33	$3.2*10^9$	4748	5	Coverage Match
TresorSGX	`d2e529ea977fe`	4	$6.7*10^9$	622	0	Perfect Match
WolfSSL	099ec3b	22	$7.5*10^9$	714	0	Perfect Match
$n = 30$		Σ	$171.2*10^9$		79	

Table 1: Target enclaves and fuzzing runs by SGXFUZZ.

No.	Enclave	Type
1–3	Goodix Fingerprint Driver Enclaves (WBDI, Coating, Glass)	Out-Of-Bounds Read
4, 5		Heap-Buffer-Overflow
6, 7		Free Uninitialized Pointer
8, 9		Memset Uninitialized Pointer
10–28		Null Pointer Deref.
29	sgxwallet	Free Uninitialized Pointer
30		Out-Of-Bounds Write
31–46	Gingytech Fingerprint Driver	Unchecked Input Addresses
		Unchecked Callback Address
47–50	STANlite	Null Pointer Deref.
		Unchecked Input Addresses
51–62	ELAN Fingerprint Biometric SSL	Null Pointer Deref.
63, 64		Use Uninitialized Pointer
65	Town Crier	Null Pointer Deref.
66–69		Stack Overflow
70	Synaptics Fingerprint Driver Enclave	Out-Of-Bounds Write
71	OMEC Project's C3P0 – Dealer	Null Pointer Deref.
72	OMEC Project's C3P0 – KMS	Null Pointer Deref.
73, 74	lockbox	Null Pointer Deref.
75–77	SGX Darknet	String Overflow
78, 79	Plinius	Unchecked Input Addresses

Table 2: Bugs and Vulnerabilities found by SGXFUZZ.

achieves about 33 k/s executions. An equivalent hello-world enclave in our runner achieves 200 k/s executions.

6.3 Structure Synthesis Accuracy

We now evaluate the accuracy of the struct layouts generated through our structure synthesis. We assess the layout's accuracy by comparing the synthesized layout with the expected layout obtained from the source files of an enclave. Since this method requires source code access, we conduct this evaluation only on a subset of our fuzzing targets. However, we manually verified that the structures for the binary-only targets are reasonable. We refer to the structure from the source code as *reference struct*.

While comparing the reference struct and our synthesized layout, we compare each pointer's offset and the amount of allocated memory behind the pointers. If that memory again contains a pointer, we continue this comparison recursively. In addition, we also compare the location of size fields and which buffer they belong to. If this relationship is not apparent through the name in the source code, we read through the source code to deduct the relationship from the code's semantic.

The results of our evaluation are shown in Table 1. Ten out of the 23 examined open-source enclaves match the expected structure from the source code perfectly for every ECALL. This means that *all* pointers are synthesized at the correct offset with the exact amount of memory as the reference, and that *all* size fields were detected and reference the correct pointer.

There are also 12 cases of *Coverage Match*, which means that only fields from the input structure were synthesized where access to was covered during fuzzing. Thus, the synthesis did not fully restore the input layout in these cases, but this is not a limitation of our approach in any way because the pointers were not used in the covered code paths, and it does not matter to the program if they exist or not. As soon as they are used, our synthesis will most likely provide them. The missing coverage is caused by common fuzzing roadblocks, i.e., check sums. Since our approach is conceptually not tied to a specific fuzzer but theoretically works with any greybox fuzzer, it would be possible to develop a payload mutator that handles these roadblocks. However, we already use state-of-the-art greybox fuzzing techniques like Redqueen [3] that aim to tackle common fuzzing roadblocks.

Ultimately, our approach could recover all pointers and size field on the assessed subset of samples except for one enclave that needed minimal manual assistance (Section 6.3).

Maximum Fault Pointer Offset. During the implementation in Section 5.3.1, we used the value `0x100` as the maximal offset of the fault address to the corresponding value in the input that was used as a pointer to determine the payload position where a pointer is expected. As we chose this constant arbitrarily, we now evaluate its impact.

The previous evaluation already shows that the struct synthesis creates perfectly matching layouts for covered code, including pointers with buffers larger than `0x100` bytes. The synthesis of such large structs can only work if a field with an offset lower than `0x100` is accessed first *on any covered code path*. Then, after that pointer is synthesized, the sizes can grow beyond `0x100` bytes. Since it is a typical pattern for enclaves to copy buffers into the secure enclave memory before accessing them, it is ensured that the fields of such large buffers are accessed in ascending order, e.g., through `memcpy`.

There are cases in our logs of the sgxwallet enclave where fault values have a difference between `0x100` and `0x1000` and thus not triggered the struct synthesis. However, our synthesis could ultimately detect these pointers through other code paths showing that our synthesis even works in these cases. A difference even larger than `0x1000`—as unlikely as it would be—would access memory beyond the guard page of our allocation scheme and cannot be matched to a pointer in the layout in a meaningful way.

Union Fields and Type Enums. Another point of concern were union fields, where distinct data structures are used based on a type field. We were only able to identify this pattern in one closed source enclave (Synaptics Fingerprint Driver Enclave), which hinders a detailed analysis. However, SGXFUZZ synthesized slightly different structures based on the (manually confirmed) type field, which shows that SGXFUZZ is able to handle this case. Since the layouts are tied to the fuzzer's input mutation scheduler, the generated layouts can branch together with the payloads for different code paths. Thus, union fields pose in no way a limitation of our approach.

Case Study: Output buffer in sgxwallet. The sgxwallet enclave challenged our structure synthesis using large, constantly sized output buffers. While the synthesis worked perfectly on most fields of the buffer, it missed creating the output buffer to the structure with a size of 1024 bytes. The manual analysis quickly revealed that the struct synthesis rejected some faults with the difference of `0x400`, which is larger than its tolerance. Those were caused when the target accessed this buffer at the end of the ECALL to write its output back to the normal world. Since the sizes are hard-coded and no other access is made to this buffer, the fuzzer could not generate the final struct layout in this case. For this case manual confirmation is needed, i.e., manually adding this buffer to the seeds of the fuzzer.

6.4 Feature Evaluation

SGXFUZZ includes several mechanisms to improve the code coverage and its ability to find enclave-specific bugs (cf. Section 5). In the following, we first demonstrate the effectiveness of the Struct Recovery, Size Field Detection, and Size Field Havoc by evaluating the coverage gained through the individual features using an ablation study on four enclaves. Thereafter, we demonstrate the effectiveness of the pointer location havoc by showing that it successfully detects vulnerabilities due to memory location in several enclaves.

6.4.1 Ablation Study

We conduct our ablation study with four configurations, as each feature depends on the previous one. First, we perform a fuzzing run without our Struct Recovery. We only use our fuzzing setup and enclave dumping to make the enclaves fuzzable. Then, we provide the enclave a single linear input buffer for each enclave's ECALL. There are no pointers or identified size fields in the input. Next, we sequentially enable our features, the *Struct Recovery*, *Size Field Detection*, and *Size Field Havoc* in the subsequent runs with a total of four different fuzzing campaigns per evaluated enclave. We measure the achieved code coverage (unique covered basic blocks) and compare them between the four configurations. Since we are comparing different fuzzing campaigns that are nondeterministic, we use five redundant fuzzing runs to reduce noise.

Figure 5 shows three distinct result patterns across four enclaves. The base case of Signal Secure Value Recovery, where none of our fuzzing features are active, has a median of 495 uniquely covered basic blocks. In contrast, activating our Struct Recovery increases the covered basic blocks by 12 times to 5,942 on average. However, the Struct Recovery measurement has a high variance ranging from 4,047 to 7,712 because size fields are not yet detected and random fuzzing input is used in size fields. Using the Size Field Detection, these size fields get detected but will have a fixed value from

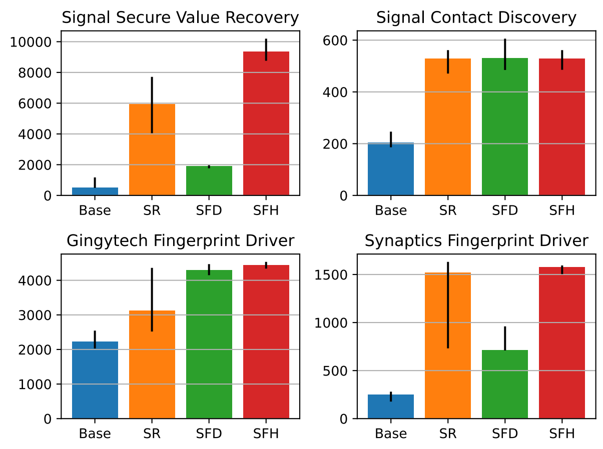


Figure 5: Ablation Study showing the cumulative effect of the different fuzzing features. (1) Flat Buffer, (2) Struct Recovery, (3) Size Field Detection, and (4) Size Field Havoc.

the time of detection. This decreases the covered basic blocks, as branches that depend on differently sized buffers are not triggered. Lastly, the Size Field Havoc solves that problem by mutating detected size values. This yields the highest of all covered basics blocks (10,196) and reduces the variance in the fuzzing process from 3,365 with Struct Recovery only to 1,437 with Size Field Havoc.

Synaptics Fingerprint Driver Enclave shows essentially the same pattern. Struct Recovery increases the covered basic blocks by 6.1 times, but with a high variance of 900 basic blocks between the lowest and highest run. Size Field Detection alone reduces the covered basic blocks, but in combination with Size Field Havoc, it yields higher results on average. However, in the case of the Synaptics Fingerprint Driver Enclave, the highest measurement of 1,631 basic blocks was done with Struct Recovery only. This is to be expected with a sufficient amount of repetitions, as has the chance to guess the size fields lucky. Nonetheless, the ten times higher variance of Struct Recovery compared to Size Field Havoc's variance of 91 basic blocks proves that the combination of all features provides significantly more stable and reproducible fuzzing runs.

Signal Contact Discovery shows a different pattern. The Struct Recovery still yields a major advantage over the base case with a 2.6-fold increase of covered basic blocks. However, the following configurations do not provide coverage gain because only one ECALL uses a size field and—while the Size Field Detection identifies it correctly—its usage depends on global data, which SGXFUZZ currently cannot cover (cf. Section 7).

Finally, the Gingytech Fingerprint Driver enclave shows the third pattern, which consists of three steps, where the Struct Recovery yields a 1.4-fold increase in covered basic blocks. Then, the Size Field Detection increases the coverage again by 1.4 to 4,297 basic blocks. However, the Size Field Havoc only increases the coverage to 4,440 basic blocks, a 2.0-fold increase compared to the base case. After reverse engineering parts of the enclave's code, we concluded that the enclave contains several complex conditions. These complex conditions considered several fields and from the payload, including size fields, which a fuzzer has to guess correctly to flip the branch condition. Since our Size Field Detection effectively removes one of the fields from equations by fixing it to a buffer's size, the complexity that the fuzzer has to solve shrinks. Thus, it becomes easier for the fuzzer to flip these branch conditions.

In conclusion, this ablation study shows that the features of SGXFUZZ are able to tackle the blockers of the enclave interface and yield in conjunction a consistently high coverage.

6.4.2 Pointer Location Havoc

We evaluate the pointer location havoc based on the number of discovered vulnerabilities. In contrast to the aforementioned structure recovery and size field features, it does not aim to increase code coverage. Instead, it is a bug oracle to test if enclaves accept attacker-controlled addresses within secure memory (cf. Section 5.4).

The general insight is that most recent enclaves correctly use the Intel SGX SDK's wrapper to avoid these vulnerabilities. Though, using this oracle, we found a total of 9 bugs related to memory locations in 4 enclaves. These vulnerabilities are caused by using `[user_check]` in the EDL files and then insufficiently checking the pointers in the user code. In most cases, this type of vulnerability could be easily fixed by using the SGX SDK's `[in]` and `[out]` flags that create secure copies. However, these SDK features are only usable for flat buffers and would not have been sufficient in the case of the Gingytech Fingerprint Driver enclave, which had 3 of these bugs. This enclave passes C++ objects that contain *vtables* and pointers to other objects.

6.5 Scalability of SGXFUZZ vs. TEEREX

We compare SGXFUZZ in two key areas to TEEREX to evaluate the performance improvements offered by our approach. First, we analyze the enclaves from Table 1 in TEEREX and record the achieved code coverage and the number of results that it generated. Second, we compare the number of reported findings for both tools.

Coverage. The results in Table 3 show that, without exception, SGXFUZZ achieves higher code coverage in every enclave. The increase in code coverage can go as high as 29-times or even 61-times the code coverage compares to TEEREX. Regardless, the median factor of the coverage gain is 2.2.

This shows that SGXFUZZ achieves higher code coverage in all cases. Assuming that our broad spectrum of enclaves includes a wide spectrum of code complexities, this shows that SGXFUZZ scales to a higher code coverage on varying code path complexity.

Number of Reports. Further, Table 3 shows the number of reported bug candidates. TEEREX discovered candidates for 17 enclaves, however, 13 of these enclaves have more than 200 reports, which is a large number for a human analyst to analyze manually. Moreover, 8 enclaves have over 1,000 reports, which can hardly be fully analyzed by a human. SGXFUZZ, in comparison, found bugs in 14 enclaves, of which only one is above 200 reports. This enclave, the Gingytech Fingerprint Driver is exceptionally insecure because it does not sanitize the input and uses values from the fuzzing payload directly as `call` and `jmp` addresses. In comparison, TEEREX reports a total of 16,624 findings for the same enclave, about 70 times as much. On average of all tested enclaves, TEEREX produces more candidates than SGXFUZZ by a factor of 302, which makes the number of reports from TEEREX hardly analyzable by a human. In contrast, SGXFUZZ' number of reports are always within range of human capabilities.

Additionally, since SGXFUZZ does not produce false positives, all reports are caused by real bugs (cf. Section 6.1). Contrasting, TEEREX produces a large number of reports that contain false positives due to limitations of symbolic executions (cf. Section 3.2). This again increases the effort to find actual bugs among reports. As evidenced by the drastically reduced number of findings and increased number of validated bugs, SGXFUZZ improves the scalability of human analysis.

6.6 Coverage

The coverage achieved by a fuzzer and the maximum achievable coverage are crucial metrics. However, there are multiple challenges that make it infeasible to determine the amount of reachable code in an enclave, which we will elaborate on in this section. Effectively, we lack ground truth for a fair quantitative analysis.

There are different approaches to approximate the amount of reachable code. First, we could assume that all code within an enclave is reachable. The problem with this approach is that all libraries are linked statically because dynamically linking (system) libraries is not possible in SGX. This includes large but varying amounts of dead code into the enclave binary. For example, in sgxwallet, `memcpy` has a total of 1,117 basic blocks, of which we covered 184. This version of `memcpy` uses unrolled loops for specific alignments and selects different instruction set extensions based on processor features to enhance performance. Since our fuzzer naturally cannot cover the branches for two different instruction sets on one platform, this creates dead code that is not easily identifiable. In contrast, to sgxwallet, MobileCoin uses another version of `memcpy` with 21 basic blocks and Signal Secure Value Recovery's `memcpy` only has a single basic block that uses `rep movsb`. A similar problem effects other libraries. For example, in sgxwallet, functions from the C++ standard library accumulate a total of 4,457 basic blocks, of which we covered 69. A static analysis shows that at most 212 blocks may be reached through any ECALL indicating significant amounts of dead code.

Second, we could exclude all libraries and favor the remaining application code. On the one hand, identifying library code is not an easy task by itself because library code is often inlined and not clearly distinguishable in the binaries. On the other hand, the primary logic of an enclave may be contained within a library and therefore should not be excluded, like an enclave that wraps an SSL or crypto library. Thus, in addition to the trusted standard library and STL, there are other libraries such as a Rust trusted standard library and mbedTLS that contribute to the total number of unreachable code blocks with an unknown amount. We have no means of determining how much code of these libraries is actually used or essential in the enclaves' logic.

In addition, we cannot use solutions like gcov [75] to determine code coverage in open-source scenarios. For similar reasons, the measurements of gcov are unreliable because is would exclude pre-compiled libraries and include dead source code (e.g., code for specific CPU features or runtime configurations). In practical terms, including gcov in the compiler setup for SGX requires a considerable amount of engineering because it relies on the system's standard library for file system operations. However, enclaves have to exclude system libraries and Intel's trusted libc for SGX does not provide file system operations. As such, gcov is not compatible with the trusted libc.

Considering the aforementioned reasons, we can only count the total number of basic blocks in the enclave binary. However, this number does not represent in any scenario the reachable code. Hence, the coverage ratios should be taken with care. Note that we do not implement new payload mutators for SGXFUZZ, but contribute to the fundamental fuzzing capabilities of SGX enclaves and ensure that existing greybox fuzzing techniques can be applied. Hence, the mutation-specific aspects and a detailed coverage analysis that evaluates the underlying fuzzers, kAFL, RedQueen, and radamsa, are presented in the respective papers [3, 39, 67]. The evaluation of the fuzzing capability of SGX enclaves in Section 6.3 shows that the input structure synthesis works as expected.

We include our achieved code coverage, as well as the total amount of basic blocks in the enclave binaries in Table 3.

7 Discussion and Future Work

Analyzing OCALLs. In this work, we focus on the execution of ECALLs which are the primary means and only entry point of an enclave and thus, must enforce the security requirements of the trusted execution environment. However, ECALLs may exit early for an OCALL to be resumed later. Since our prototype implementation SGXFUZZ does not resume after OCALLs, this can lead to missed coverage. However, we found that this limitation is tolerable, as we found only two enclaves that we could not fuzz effectively due to a lack of OCALL support and many enclaves use no OCALLs at all.

Enclave	Total #BBs	TEEREX #BBs		TEEREX Reports	SGXFUZZ #BBs		SGXFUZZ Reports	SGXFUZZ Verified Bugs	SGXFUZZ #BB / TEEREX #BB
BiORAM-SGX	48865	788	1.6 %	2831	1802	3.7 %	0	0	2.29
ELAN Fingerprint Driver	33969	666	2.0 %	0	675	2.0 %	0	0	1.01
ELAN Fingerprint Biometric SSL	37739	1408	3.7 %	22037	2206	5.8 %	67	14	1.57
Gingytech Fingerprint Driver	47913	2275	4.7 %	16624	4080	8.5 %	236	16	1.79
Goodix Fingerprint Driver Coating Enclave	20800	942	4.5 %	884	1698	8.2 %	11	6	1.80
Goodix Fingerprint Driver Glass Enclave	32672	942	2.9 %	1328	1736	5.3 %	10	8	1.84
Goodix Fingerprint Driver WBDI Enclave	21169	655	3.1 %	1	1404	6.6 %	56	14	2.14
Intel AE Launch Enclave (LE)	13270	149	1.1 %	0	429	3.2 %	0	0	2.88
Intel AE Provisioning Cert. Enc. (PCE)	18438	208	1.1 %	0	490	2.7 %	0	0	2.36
Intel AE Provisioning Enclave (PVE)	32368	190	0.6 %	0	452	1.4 %	0	0	2.38
Intel AE Quoting Enclave (QE)	32012	180	0.6 %	0	407	1.3 %	0	0	2.26
Intel SDK Initialize ECALL	–	not supported			257	–	0	0	–
KubeTEE TFF	92595	429	0.5 %	903	3320	3.6 %	0	0	7.74
Ledger BOLOS	35100	431	1.2 %	412	1076	3.1 %	0	0	2.50
lockbox	62176	2387	3.8 %	7406	3689	5.9 %	42	2	1.55
MobileCoin	42203	669	1.6 %	0	11888	28.2 %	0	0	17.77
Occlum Runtime Libos	61575	433	0.7 %	247	977	1.6 %	0	0	2.26
OMEC Project's C3P0 – Dealer	27478	220	0.8 %	1084	226	0.8 %	1	1	1.03
OMEC Project's C3P0 – KMS	26169	483	1.8 %	93	1673	6.4 %	1	1	3.46
Plinius	17188	489	2.8 %	412	646	3.8 %	2	2	1.32
SGX Darknet	9944	300	3.0 %	491	527	5.3 %	4	3	1.76
sgxwallet	25528	3274	12.8 %	4303	3865	15.1 %	5	2	1.18
Signal Contact Discovery	2299	236	10.3 %	0	560	24.4 %	0	0	2.37
Signal Secure Value Recovery	36675	225	0.6 %	0	11989	32.7 %	0	0	53.28
STANlite	30688	2279	7.4 %	0	6621	21.6 %	5	4	2.91
Synaptics Fingerprint Driver Enclave	51878	114	0.2 %	0	1416	2.7 %	4	1	12.42
Tensorflow Lite	14631	244	1.7 %	0	492	3.4 %	0	0	2.02
Town Crier	41798	3755	9.0 %	6568	4748	11.4 %	14	1	1.26
TresorSGX	4859	283	5.8 %	1	622	12.8 %	0	0	2.20
WolfSSL	19167	345	1.8 %	2	714	3.7 %	0	0	2.07

Table 3: Result comparison of SGXFUZZ and TEEREX. The median factor of the coverage increase is 2.20 (mean: 4.88).

In addition to providing no advantages to most enclaves, universal OCALL support for SGXFUZZ is not trivial and effectively requires call chain support. While our runner is capable of executing OCALLs, the return values of OCALLs are under malicious control and the semantics are unknown to the runner. Therefore, OCALL support is equivalent to support a call chain consisting of the main ECALL followed by multiple calls to the SDK's special OCALL-return ECALL. This requires the fuzzer to generate multiple *dependent* input payloads for the ECALL and the following OCALLs. Further, each OCALL requires its own custom input structure layout. Since we have few examples of missing OCALLs being problematic, we leave this topic as future work.

Dynamic ECALL Chains. Since enclaves are stateful, calling ECALLs subsequently in a particular order can enhance coverage and reach more complex program states than each ECALL individually. The most common pattern is an *Initialization ECALLs* that must be called prior to any other ECALL to set up the internal state. We solve this using a static ECALL during the fuzzing setup (Section 5.4). However, fully dynamic call chain support poses similar challenges as OCALL support and requires the fuzzer to generate multiple *dependent* input payloads and synthesize multiple layouts. On top of that, dynamic call chains require sophisticated selection of the calls to chain. For OCALLs the call chain is determined by the originating ECALL and dependency is implicitly given. ECALL-chains have no natural order, so a naive fuzzing approach for chains is likely to waste time on chains of independent ECALLs that can never produce additional coverage.

Fuzzing Other Enclave ABIs. During this work, we limit our approach to SGX and the corresponding Intel SGX SDK. However, there are other enclave ABIs such as Google Asylo [35], Graphene [76], or the Fortanix Rust SDK [30]. SGXFUZZ is conceptually not tied to any enclave ABI and fuzzing another ABI should be as simple as tweaking the runner to set up a different enclave environment to conform with another calling convention. The structure synthesis only requires default memory accesses that emit SEGV signals. Further, none of the SGX specific fuzzer extension discussed in Section 5.4 is conceptually tied to SGX but rather to the concept of enclaves in general. The pointer location havoc and the corresponding bug oracle apply to any trusted execution environment (TEE), where the enclave and the user-space have dedicated regions in a shared address space.

8 Related Work

SGX and TEE vulnerabilities. Recent work on the security of SGX exposed a series of side-channel attacks [12, 13, 41, 61, 70] that may leak secret data due to hardware flaws. While these attacks can be prevented globally using special compiler options or using updated CPU generations, vulnerabilities of individual enclaves need to be addressed by the vendor. As such, an analysis of individual enclaves is still required because these vulnerabilities cannot be prevented by updating the SGX environment.

Fuzzing. Fuzzing [10, 11, 55] is a popular technique to assess the security of software and hardware components [27, 59, 63] and to find impactful vulnerabilities. There has been significant interest to enable fuzzing for even more platforms and targets to integrate automated security testing. Grammar-based input fuzzers [6, 33, 49, 80, 83] infer invariants in well-structured data types to produce impactful test cases for complex targets. These fuzzers operate on linear input and cannot handle nested pointers. Protocol fuzzers [5, 18, 24, 31] generate a *sequence* of messages (or function calls) to emulate the interaction between two parties. They analyze data dependencies of the message in sequence to generate useful test messages that iteratively alter the internal state of the target and uncover vulnerabilities. These fuzzers tackle the challenges to detect the types of message data, or to infer the internal state of a remote target to guide the fuzzing process to uncovered states. However, data is sent to a well-defined endpoint (e.g., network or syscall) so that pointer detection is not required. Further, these fuzzers often rely on the generation of binary data structures based on source code: Difuze [19] uses kernel driver source files to generate data structure to fuzz these drivers. syzkaller [37] uses a grammar made from Linux header files to generate sequences of syscalls to uncover bugs in the Linux kernel. SyzGen [15] improves on syzkaller and automatically generates syscall specifications for macOS drivers. Morphuzz [60] leverages the MMIO address mapping in the hypervisor to generate sequences of I/O commands for virtual device drivers. FuzzGen [46] analyzes the source code of libraries and host applications to infer data structures. In contrast, our approach can infer nested structure layouts on-the-fly for closed-source binaries during the fuzzing run.

The HFL fuzzer [51] symbolically tracks the interactions between kernel and user-space memory to recover data structures. While there are similarities to SGX' memory model, the trusted world in SGX can access any host-memory directly, contrary to the kernel which uses special access functions, which HFL relies on. As such, adapting HFL's approach to SGX requires analyzing *all* pointer dereferences, not only those in access functions. This requires more intensive tracking and leads to the state explosion issues as discussed in Section 3. Most importantly, none of the existing approaches were suitable for fuzzing SGX as they require source code or rely on assumptions that do not hold for the SGX environment.

Emulation and SGX. Emulated environments for SGX have not yet been in the focus of research. PartEMU [38] is an emulator for ARM TrustZone that enables fuzzing of TrustZone enclave. On the other hand, there are various approaches to emulate normal environments inside SGX enclaves [1, 7, 21, 71, 76]. Further, Intel has published KVM SGX [43] which is used to pass-through real SGX hardware enclaves to a guest system and cannot be used to emulate SGX and should not be confused with our approach of SGX emulation.

9 Conclusion and Summary

In this paper we present SGXFUZZ, a novel approach to synthesize binary input structures with nested pointers that enables coverage-guided fuzzing of SGX enclaves without prior knowledge of the ECALL's semantics. To retrieve coverage feedback from otherwise not introspectable enclaves, we present enclave extraction methods and an enclave runner for user-space execution of enclaves at native speed.

We evaluate our approach and show that our structure synthesis offers a robust method to generate input layouts on-the-fly during a fuzzing process. In addition, we show that SGXFUZZ outperforms TEEREX in terms of covered code, detected bugs, and human analysis effort.

Acknowledgment

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972 and SFB 1119 – 236615297 within project S2. The work was partially supported by the MKW-NRW research training group *SecHuman*. We thank our shepherd Chengyu Song and the anonymous reviewers for their valuable feedback.

References

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Nov. 2016.

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *IEEE Symposium on Security and Privacy*. 2020.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, R. Gawlik, and T. Holz. "Redqueen: Fuzzing with Input-to-State Correspondence". In: *NDSS*. 2019.

[4] Various Authors. *OpenEnclave – SDK for developing enclaves*. URL: https://github.com/openenclave/openenclave (visited on June 8, 2021).

[5] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. "SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr". In: *International conference on information security*. Springer. 2006.

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. "Synthesizing program input grammars". In: *ACM SIGPLAN Notices* 52.6 (2017).

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

[8] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *27th USENIX Security Symposium, USENIX Security*. 2018.

[9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. "GRIMOIRE: Synthesizing Structure while Fuzzing". In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed greybox fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Association for Computing Machinery, 2016.

[12] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul 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, 2018.

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

[14] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018.

[15] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. "SyzGen: Automated Generation of Syscall Specification of Closed-Source MacOS Drivers". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. 2021.

[16] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. "PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2019.

[17] Tobias Cloosters, Michael Rodler, and Lucas Davi. "TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[18] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. "Prospex: Protocol specification extraction". In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE. 2009.

[19] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. "Difuze: Interface aware fuzzing for kernel drivers". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[20] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: (2016). URL: https://eprint.iacr.org/2016/086.

[21] Jinhua Cui, Shweta Shinde, Satyaki Sen, Prateek Saxena, and Pinghai Yuan. "Dynamic Binary Translation for SGX Enclaves". In: *ACM Trans. Priv. Secur.* (Apr. 2022).

[22] Weidong Cui, Marcus Peinado, Karl Chen, Helen Wang, and Luis Irún-Briz. "Tupni: Automatic reverse engineering of input formats". In: Jan. 2008.

[23] Thomas De Backer. "Fuzzing Intel SGX Enclaves". MA thesis. KU Leuven, 2019. URL: https://u.debacker.me/Thomas_De_BackerFuzzing_Intel_SGX_enclaves.pdf.

[24] Joeri De Ruiter and Erik Poll. "Protocol State Fuzzing of {TLS} Implementations". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015.

[25] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020.

[26] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. "Everything You Should Know About Intel SGX Performance on Virtualized Systems". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2019).

[27] Bo Feng, Alejandro Mera, and Long Lu. "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[28] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. *WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats*. Nov. 2019.

[29] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++ : Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[30] Fortanix. *Fortanix*. URL: https://fortanix.com/ (visited on June 8, 2021).

[31] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Pulsar: Stateful black-box fuzzing of proprietary network protocols". In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2015.

[32] Edgeless Systems GMBH. *Edgeless Systems*. URL: https://www.edgeless.systems/ (visited on June 8, 2021).

[33] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. "Grammar-based whitebox fuzzing". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2008.

[34] Google. *AddressSanitizer*. URL: https://github.com/google/sanitizers/wiki/AddressSanitizer (visited on June 8, 2021).

[35] Google. *Asylo*. URL: https://github.com/google/asylo (visited on June 8, 2021).

[36] Google. *Honggfuzz*. URL: https://github.com/google/honggfuzz (visited on June 8, 2021).

[37] HyungSeok Han and Sang Kil Cha. "Imf: Inferred model-based fuzzer". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[38] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. "PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 789–806.

[39] Aki Helin. *radamsa - a genereal purpose fuzzer*. 2021. URL: https://gitlab.com/akihe/radamsa (visited on June 8, 2021).

[40] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. "Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction". In: *ieee-oakland*. 2020.

[41] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.1 (Nov. 2019).

[42] Intel. *Intel® Software Guard Extensions SDK for Linux**. URL: https://01.org/intel-software-guard-extensions (visited on Aug. 20, 2021).

[43] Intel. *KVM SGX*. URL: https://github.com/intel/kvm-sgx (visited on June 8, 2021).

[44] Intel. *Linux SGX Driver*. 2016. URL: https://github.com/intel/linux-sgx-driver/.

[45] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*. Intel. Dec. 2017.

[46] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. "Fuzzgen: Automatic fuzzer generation". In: *29th USENIX Security Symposium (USENIX Security '20)*. 2020.

[47] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. "OpenSGX: An Open Platform for SGX Research". In: *Proceedings of the Network and Distributed System Security Symposium*. Feb. 2016. URL: https://github.com/sslab-gatech/opensgx.

[48] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. "WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning". In: *Symposium on Network and Distributed System Security (NDSS)*. 2021.

[49] SoftSec KAIST. *CodeAlchemist*. URL: https://github.com/SoftSec-KAIST/CodeAlchemist (visited on June 8, 2021).

[50] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao 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.

[51] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. "HFL: Hybrid Fuzzing on the Linux Kernel". In: *NDSS*. 2020.

[52] Andi Kleen and Beeman Strong. "Intel processor trace on linux". In: *Tracing Summit* (2015).

[53] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *26th USENIX Security Symposium (USENIX Security 17)*. Aug. 2017.

[54] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. "Steelix: Program-State Based Binary Fuzzing". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Association for Computing Machinery, 2017.

[55] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. "Fuzzing: State of the Art". In: *IEEE Transactions on Reliability* 67.3 (2018).

[56] Jiashuo Liang, Guancheng Li, Chao Zhang, Ming Yuan, Xingman Chen, and Xinhui Han. "RIPT – An Efficient Multi-Core Record-Replay System". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. 2020.

[57] R3 HoldCo LLC. *Conclave*. URL: https://conclave.net/ (visited on June 8, 2021).

[58] Dominik Maier, Benedikt Radtke, and Bastian Harren. "Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing". In: *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019.

[59] Dominik Maier, Lukas Seidel, and Shinjo Park. "BaseSAFE: Baseband Sanitized Fuzzing through Emulation". In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '20. New York, NY, USA: Association for Computing Machinery, 2020.

[60] "Morphuzz: Bending (Input) Space to Fuzz Virtual Devices". In: *31st USENIX Security Symposium (USENIX Security 22)*. Aug. 2022.

[61] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020.

[62] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. "Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[63] Hui Peng and Mathias Payer. "USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[64] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: Fuzzing by Program Transformation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018.

[65] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing". In: Feb. 2017.

[66] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[67] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. "KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. 2017.

[68] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. "Nyx-Net: Network Fuzzing with Incremental Snapshots". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. 2022.

[69] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS*. 2007.

[70] Ming-Wei Shih, S. Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *NDSS*. 2017.

[71] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. "Panoply: Low-TCB Linux Applications With SGX Enclaves." In: *NDSS*. 2017.

[72] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. "Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020.

[73] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting fuzzing through selective symbolic execution". In: *Symposium on Network and Distributed System Security (NDSS)*. 2016.

[74] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy, S&P*. 2013.

[75] GCC Team. *gcov—a Test Coverage Programs*. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (visited on Jan. 28, 2022).

[76] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference, USENIX ATC*. 2017.

[77] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2019.

[78] Dmitry Vyukov. *syzkaller*. URL: https://github.com/google/syzkaller (visited on June 8, 2021).

[79] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. "Designing New Operating Primitives to Improve Fuzzing Performance". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[80] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. "ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

[81] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *27th USENIX Security Symposium (USENIX Security 18)*. Aug. 2018.

[82] Michal Zalewski. *American Fuzzing Lop (AFL)*. URL: http://lcamtuf.coredump.cx/afl/ (visited on Nov. 13, 2021).

[83] Google Project Zero. *fuzzilli*. URL: https://github.com/googleprojectzero/fuzzilli (visited on June 8, 2021).

[84] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. "PTfuzz: Guided Fuzzing With Processor Trace Feedback". In: *IEEE Access* (2018).

[85] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang. "STOCHFUZZ: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting". In: *IEEE Symposium on Security and Privacy*. 2021.

[86] Zyantific. *Zydis*. URL: https://github.com/zyantific/zydis (visited on June 8, 2021).