TEEREX: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves

Tobias Cloosters, Michael Rodler, Lucas Davi
University of Duisburg-Essen, Germany
{tobias.cloosters, michael.rodler, lucas.davi}@uni-due.de

Abstract

Intel's Software Guard Extensions (SGX) introduced new instructions to switch the processor to *enclave mode* which protects it from introspection. While the enclave mode strongly protects the memory and the state of the processor, it cannot withstand memory corruption errors inside the enclave code. In this paper, we show that the attack surface of SGX enclaves provides new challenges for enclave developers as exploitable memory corruption vulnerabilities are easily introduced into enclave code. We develop TEEREX to automatically analyze enclave binary code for vulnerabilities introduced at the host-to-enclave boundary by means of symbolic execution. Our evaluation on public enclave binaries reveal that many of them suffer from memory corruption errors allowing an attacker to corrupt function pointers or perform arbitrary memory writes. As we will show, TEEREX features a specifically tailored framework for SGX enclaves that allows simple proof-of-concept exploit construction to assess the discovered vulnerabilities. Our findings reveal vulnerabilities in multiple enclaves, including enclaves developed by Intel, Baidu, and WolfSSL, as well as biometric fingerprint software deployed on popular laptop brands.

1 Introduction

Intel recently introduced a sophisticated trusted execution environment (TEE) called Software Guard Extensions (SGX) [33, 41, 55]. SGX allows application developers to create so-called *enclaves* to encapsulate sensitive application code and data inside a TEE that is completely isolated from other applications, operating systems, and hypervisors. The only trusted component in the SGX setting is the Intel CPU itself. Most prominently, SGX features confidentiality and integrity protection for any data that is written to its main memory. In addition, SGX implements well-known Trusted Computing concepts such as data binding and sealing as well as remote attestation, i.e., ensuring the remote SGX enclave is in a trustworthy state. Putting all these features together, this allows a user to establish a secure channel directly to the SGX enclave (which potentially runs in an untrusted cloud environment) and perform remote attestation to ensure the integrity of the remote SGX hardware and enclave. That said, SGX is a strong isolation mechanism for sensitive data (e.g., personal information or cryptographic keys) as well as security-critical code (e.g., for the sake of intellectual property protection). It also found its way into commercial applications, e.g., fingerprint sensor software (Section 5), DRM protection [22], and privacy-preserving applications like Signal [53]. As such a promising technology, SGX has been used and targeted extensively in previous research. Many projects propose to utilize SGX for enhanced security guarantees, e.g., processing private data in public clouds [6, 60].

From its infancy, it was clear that SGX cannot withstand all flavors of attacks [44]. In particular, SGX cannot protect against two classes of attacks: (1) side-channel attacks and (2) memory corruption attacks inside the enclave. The former attack technique exploits shared resources (e.g., cache) to steal secret information from within an enclave. This line of research has become a very active research field [71, 76]. Especially micro-architectural side-channels have been shown to be effective for attacking SGX enclaves due to the shared micro-architectural state of enclaves and untrusted code [68].

To our surprise, memory corruption attacks have been rarely investigated in the context of SGX. These attacks exploit programming errors (e.g., a buffer overflow) allowing an attacker to take over the enclave, hijacking the enclave's control-flow, and perform code-reuse attacks such as return-oriented programming (ROP) [62]. Further, the attacker can also exploit these errors to corrupt enclave data variables and pointers to launch data-oriented attacks such as information leaks or data-oriented programming (DOP) [37]. Prior research studied the applicability of offensive and defensive techniques against memory corruption exploits. For instance, Lee et al. [48] presented DARKROP, a code-reuse attack technique, which shows that the enclave code must not be known to an attacker to successfully launch ROP attacks against the enclave. Biondo et al. [7] showed that it is easily possible to launch powerful code-reuse attacks due to particularities of the Intel SGX SDK bypassing existing ASLR defenses such as SGX-Shield [61].

However, prior research on memory corruption attacks *always assumed the existence of memory errors, but did not investigate whether or to which extent such errors exist in real-world enclaves*. Due to the rather slow adoption of the SGX technology, this is not an easy question to answer. Ideally, SGX enclaves contain only a minimal amount of code, which can be manually audited or even formally verified to not contain any programming mistakes. However, in our experience, legacy code bases are often ported to SGX enclaves. These ports are often not revised to handle the specialties of SGX enclaves and inherit security vulnerabilities from the legacy code base or introduce new security vulnerabilities particular to SGX enclaves. This is similar for newly written SGX code by developers not familiar with the peculiarities of SGX.

One common aspect of all SGX enclaves is that they always link to an untrusted *host application*. The host application loads an SGX enclave into its address space as it would do in case of a shared library. Indeed, the Intel SGX SDK offers a C-function like interface allowing bidirectional communication from the host application to the enclave. This interface is highly critical as invalid input may lead to a privilege escalation attack. As shown by prior research in the context of other privilege separation technologies, this is especially true when software is partitioned into privilege levels [13, 36]. That said, whenever an enclave is called, it must take special care to validate any input, particularly when the input contains code or data pointers.

Contributions. In this paper, we demonstrate that the attack surface of SGX enclaves provides new challenges for enclave developers as exploitable memory corruption vulnerabilities are easily introduced into enclave code due to a combination of the unique threat model of SGX enclaves and the current prevalent programming model for SGX (i.e., the Intel SGX SDK). We introduce the first SGX vulnerability analysis framework, called TEEREX, to automatically analyze enclave binary code based on symbolic execution (see Section 4). We implement vulnerability detectors in TEEREX that take all the peculiarities of SGX enclaves into account allowing developers to identify vulnerabilities in enclave binaries a priori, i.e., before they are utilized in production.

We especially focus our investigation on the validation of pointers that are passed from the host application to the enclave. Our findings demonstrate that developers are not aware of the difficulties of securely implementing enclave code when dealing with the critical host-to-enclave boundary. We found that the automatically generated checks of the Intel SGX SDK are insufficient for non-trivial pointer-based data structures and a lack of proper manual validation of pointers or pointer-heavy data structures can easily lead to memory corruption vulnerabilities.

Using TEEREX, we identified several vulnerabilities in publicly available enclave binaries developed at major companies such as Intel, Baidu, and Synaptics (see Section 5). Our framework features detailed vulnerability reports significantly simplifying the construction of proof-of-concept exploits to assess the reported vulnerability. Even if no information on the enclave is available, we are able to construct exploits (see the fingerprint enclaves analyzed in Section 5.5 and 5.6). Our exploits hijack the enclave's control-flow, effectively bypassing all security guarantees of the SGX technology. By performing root-cause analysis we identified five vulnerability classes that repeatedly occur in our dataset: *Passing Data-Structures with Pointers (P1)*, *Returning pointers to enclave memory (P2)*, *Pointers to Overlapping Memory (P3)*, *NULL-Pointer Dereferences (P4)*, and *Time-of-Check Time-of-Use (P5)*.

Interestingly, among the enclaves we found vulnerable is one enclave written by Intel engineers and published as an open-source example enclave on Intel's GitHub page [38]. Another interesting finding is a vulnerability in a sample SGX enclave originally developed at Baidu with the Rust SGX SDK (now an Apache Incubator project). Rust features memory safety and as such has the potential to eradicate memory corruption attacks. However, the host-to-enclave boundary is inherently memory unsafe and as such, using memory-safe programming languages in SGX does not automatically result in secure enclave code.

2 Memory Corruption in SGX

The lack of built-in memory safety in the common system-level programming languages C/C++ has led to a multitude of memory corruption vulnerabilities in the last three decades [66]. These vulnerabilities allow an attacker to perform a limited or (often) arbitrary write to memory. Such malicious writes manipulate (1) control-flow information on stack and heap (e.g., return addresses and function pointers) or (2) so-called non-control data (e.g., decision-making variables). In both cases, the attacker influences the program's execution flow and eventually executes a malicious sequence of instructions. In the recent past, we witnessed an arms race between defenses and memory corruption attacks: data-execution prevention [56, 57] effectively prevents malicious code injection in data memory, but can be bypassed by means of return-oriented programming (ROP) attacks as these only reuse code already residing in code memory [62]. Software-diversity based defenses [43, 47] mitigate ROP attacks by randomizing the location of code in memory but are circumvented if an attacker manages to dynamically disclose the code location [64]. Similarly, control-flow integrity (CFI) [1] depends on the precision of the control-flow graph (CFG) as CFG over-approximation opens the door for subtle ROP attacks [11, 23, 30]. Lastly, even if one would be able to develop a perfect CFI scheme, non-control data attacks would still be a viable attack option as they only execute execution paths that adhere to the program's CFG [15, 37, 42, 58].

In general, SGX enclaves are as susceptible to memory corruption attacks as any other system software. In fact, almost all enclaves are developed in C/C++ mainly because the official Intel SGX SDK [40] provides a C/C++ development environment. Only recently, memory-safe languages such as Rust have been explored as a programming language for SGX enclaves [73]. However, as we will show, even these cannot guarantee that enclaves are free of memory corruption vulnerabilities.

One particular challenge arises when launching memory corruption attacks against SGX enclaves: since SGX enclaves are encrypted in memory and can be shipped as an encrypted binary [6, 60], an attacker cannot necessarily perform static analysis on the enclave's binary to search for interesting ROP gadgets (i.e., enclave code sequences maliciously combined to trigger malicious operations). Lee et al. [48] tackle this challenge by repeatedly executing an enclave, triggering the execution at different entry points, and analyzing memory access to dynamically identify ROP gadgets. Note that this attack does not apply to enclaves whose code addresses are randomized for each instantiation of the enclave. On the other hand, existing SGX randomization schemes such as SGX-Shield [61] are not able to apply randomization to all of the enclave's code area: Biondo et al. [7] demonstrated that the Intel SGX SDK provides enclave libraries that are not randomized and include several powerful ROP gadgets (i.e., gadgets that allow control of many processor registers). Specifically, these gadgets are invoked when resuming the context of an SGX enclave (OCALL-return). Hence, an attacker only needs to launch a memory corruption attack and provide counterfeit context information to hijack a vulnerable enclave.

Problem Setting. We observe that existing memory corruption attacks against SGX [7, 48] exploit the host-to-enclave boundary as this serves as entry point to trigger and halt enclave execution. Further, the existing attacks assumed that the attacker is capable of hijacking the control flow of the enclave's code by means of a given memory corruption vulnerability. However, the open question is whether such vulnerabilities are likely to occur when developing enclaves. To answer this question, we reverse-engineer public enclave code and develop automated analysis techniques to assess the security of enclaves regarding memory corruption vulnerabilities. Our findings demonstrate that an erroneous implementation of the API at the host-to-enclave boundary is often the root-cause for memory corruption vulnerabilities in SGX code.

3 SGX Preliminaries

In this section, we provide background information on the Software Guard Extensions (SGX) technology of modern Intel processors and more specifically the Intel SGX SDK. The Intel SGX SDK is currently the primary way to develop SGX enclave code and is officially endorsed by Intel.

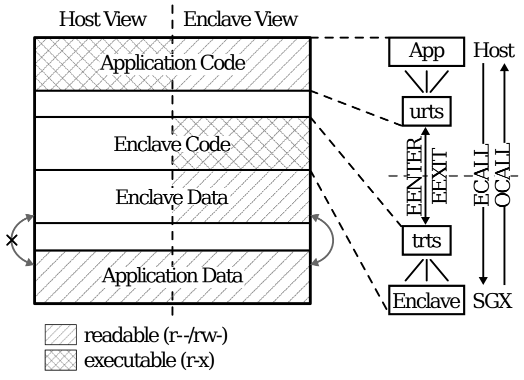


Figure 1: General overview on SGX-enabled applications.

3.1 Host-Enclave Interface

Figure 1 provides a general overview of the memory layout of SGX-enabled applications as well as the channel for host-to-enclave interaction. The SGX enclave is part of a user space application, called host process or application, which eventually loads and executes the enclave. Both host and enclave share the same virtual address space with the exception that the enclave resides in encrypted and integrity-protected memory. As shown in Figure 1, in the enclave view enclaves can access all of the host application's memory. Only the enclave memory is assumed as trusted, whereas all other memory parts are considered as untrusted.

The host process starts the enclave's execution by issuing the special *EENTER* instruction to enter the enclave. For this, enclaves define entry points in the so-called *thread control structures* (TCS), which are locked while in use by a thread. This makes the number of TCS also the maximum number of threads that can enter an enclave concurrently. A jump from the executing enclave to code in the host application results in a segmentation fault, effectively making host code non-executable for the enclave. As such, the enclave must explicitly leave enclave mode by using the *EEXIT* instruction before the thread can execute any non-enclave code.

The Intel SGX SDK provides the concept of *ECALLs* (enclave calls) on-top of *EENTER* to control the transition from application code to enclave code. For a simple enclave, not requiring multi-threading, the SDK uses only one TCS which is called with the index of the desired ECALL. First, the host application calls the ECALL wrapper in the untrusted runtime (urts). Next, the urts prepares the transition to the SGX enclave according to the so-called EDL file. Second, it executes the *EENTER* instruction to transfer control to the enclave code. More specifically, control is transferred to an enclave entry point in the SDK's trusted runtime (trts). The trts takes

```
enclave {
  trusted { // ECALLs
    public void ecall_size1( // explicit size
        [in, size=100] void* ptr);
    public void ecall_size2( // variable size in len
        [in, size=len] void* ptr, size_t len);
    public void ecall_user( // dangerous user_check
        [user_check] void* ptr);
  };
};
```

Figure 2: Example for the EDL syntax.

care of the context switch and sets up the enclave execution environment: (1) it switches the stack to a stack in enclave memory, (2) allocates secure memory and copies the arguments into the enclave, (3) calls the actual ECALL function, and finally (4) clears the registers before returning to the host application's code. Similarly, the SDK also supports calling functions of the untrusted host application, which is referred to as *OCALLs* (outside calls). For OCALLs, the trts saves the enclave's execution state to enclave memory and restores it when the call returns.

3.2 The EDL Interface Specifications

The Intel SGX SDK uses the EDL (*Enclave Definition Language*), a custom specification language to define the ECALL and OCALL interface of an enclave. The EDL language resembles a C-header file with additional syntax to specify SGX-specific information. It allows the developer to specify the prototypes of functions available as ECALL and the valid data format of input arguments. Based on the EDL file, the SDK generates wrapper code to transparently connect the function stubs in the host application with the ECALLs in the enclave. The parameters of ECALLs are transferred using auto-generated data structures. When the application invokes an ECALL, the SDK-generated code stores all parameters in the prepared structure in the untrusted host application memory. These will then be fetched by the SDK code in the enclave and unpacked for the actual ECALL code.

The SDK must be able to determine the size of the arguments to allocate a fitting buffer in the secure memory. Thus, every pointer type has to be annotated with a size such that the SDK can determine the size of the underlying buffer. Currently, the Intel SGX SDK supports copying C data types such as basic integer types, composed basic data types (*struct*) without nested pointers, 0-terminated/C-style strings, and pointers to arrays of fixed length.

Figure 2 shows an example of different features of the EDL language. In this example, a `void*` pointer is annotated with `[in, size=100]`. The SDK will generate code that allocates 100 bytes in enclave memory and copies 100 bytes from untrusted memory into the enclave. Alternatively, the developer can also specify dynamic lengths, which then refer to other parameters by name. However, when writing the interface definition in EDL, there are some peculiarities that have to be taken into account. First, it is possible to disable the SDK features. A pointer that is annotated as `[user_check]` is passed to the enclave without any auto-generated check. It is up to the enclave developer to validate the underlying buffer. Second, compound data types are only shallow copied. They are treated as buffers with a fixed size and are simply copied into secure memory. Data structures are not recursively copied, i.e. it is not checked if any of the fields in the structure is a pointer type. So, even if a developer uses the Intel SGX SDK to protect the ECALL API, there are many cases that additionally require custom validation code, which is error-prone.

4 TEEREX Symbolic Enclave Analyzer

We develop a novel symbolic execution framework, called TEEREX[1] to *automatically* identify vulnerabilities of SGX enclaves. Our framework does not only identify vulnerabilities, but also generates a detailed vulnerability report which significantly simplifies the process of constructing proof-of-concept exploits against the vulnerable SGX enclave. It supports all platforms supported by the Intel SGX SDK: Windows (PE) and Linux (ELF) binaries and both 32 and 64-bit enclaves. Note that we apply symbolic execution on the *binary level* to be able to analyze closed-source, proprietary enclaves. Our prototype of TEEREX supports the standard enclave format of the Intel SGX SDK and leaves support of custom enclave formats and loaders (e.g., the Graphene framework [67]) as future work. Further, we focus our analysis on unencrypted enclave code. In case the enclave code is encrypted neither TEEREX nor any other static analysis tool can analyze the enclave without knowing the secret key. TEEREX must be able to read and properly load the enclave's code.

In what follows, we describe the overall architecture of TEEREX (Section 4.1), elaborate on several challenges and how we tackled them (Section 4.2), and finally describe our vulnerability detection engines in detail (Section 4.3).

4.1 Architecture

Symbolic execution was first proposed in the 70s as a generalization of testing [8, 46] and has become one of the standard tools for high coverage testing and vulnerability analysis [5, 10, 12, 63]. However, the modeling of side effects caused by the operating system (OS) is highly challenging, e.g., symbolic execution must typically simulate and support all OS system calls and manage a simulated file system [5]. Fortunately, there are several SGX peculiarities that simplify symbolic execution for SGX enclaves: enclave code is self-contained (i.e., no external dependencies like libraries) and

[1]TEEREX stands for *Trusted Enclave Ecall Runtime EXploiter*

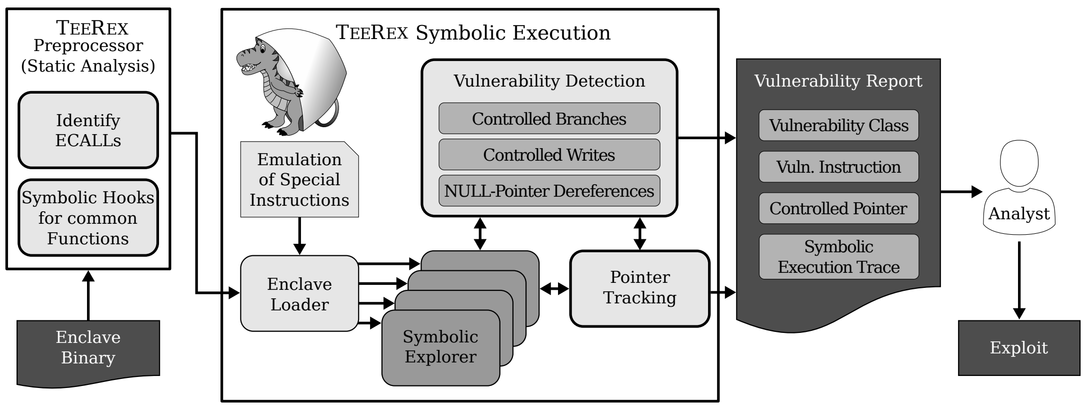


Figure 3: Architecture of TeeRex

isolated from the rest of the system. SGX enclaves are prohibited to perform any system calls and any interaction with the OS is handled by means of an *OCALL* to the untrusted host application.

Figure 3 shows the architecture of TeeRex' symbolic analysis pipeline. The main goal of TeeRex is to find vulnerable states during the symbolic exploration. Further, it aims to collect meta-data to eventually generate a detailed vulnerability report. This is achieved by executing each ECALL symbolically and checking every state for different vulnerability classes. To produce accurate vulnerability reports, we add pointer tracking to the symbolic execution engine. This allows us to track pointer dereferences and propagate labels that allow us to distinguish between data loaded from enclave and host memory. As a result, TeeRex can spot vulnerable instructions that read data from outside of the protected enclave memory. This is a necessary design decision as enclaves can be loaded by arbitrary (malicious) host applications.

We leverage the well-known ANGR framework [63] as our symbolic explorer. This allows us to extract memory constraints from enclave code, which is subsequently needed for vulnerability analysis. ANGR itself does not support executing SGX enclaves because: (1) ANGR cannot jump from the host application to the enclave (2) there is no setup for an initial environment to directly execute ECALLs, (3) enclaves utilize CPU instructions not supported by ANGR, (4) TeeRex leverages enclave specifics to scale over multiple processes and machines, while ANGR is limited to one thread, and (5) the common trusted functions for memory allocation are not directly supported by ANGR. Furthermore, ANGR does not perform any vulnerability analysis by itself: its purpose is to provide a robust and comprehensive framework to perform static analysis and symbolic execution. As we will describe in Section 4.2, TeeRex tackles all the above mentioned challenges. As shown in Figure 3, TeeRex is split into several major components.

Preprocessor: The first step in the pipeline depicted in Figure 3 is to pre-process the enclave binary to (1) identify instructions and functions that cannot be executed symbolically, and (2) to locate the ECALL table and extract the addresses of the ECALL functions. This preliminary static analysis step allows us to instrument specific binary instructions to increase the performance and coverage of the analysis.

Enclave Loader: The enclave loader sets up the initial environment to execute one ECALL. It replaces the identified *common functions* and *special instructions* with emulating Python code. Further, it creates the argument structure for the ECALL with unconstrained symbolic values.

Symbolic Explorer: The symbolic execution performed by the ANGR framework can be distributed across multiple machines, as the ECALLs are analyzed individually. The results are merged later in the vulnerability reports for the analyst.

Vulnerability Detection: TeeRex analyzes the symbolic states during ANGR's symbolic exploration for vulnerabilities in the enclaves. It specifically analyzes instructions that access memory and jumps. This is described in detail in Section 4.3.

Pointer Tracking: The majority of vulnerabilities in SGX enclaves are due to insecure pointer usage and lack of pointer validation. TeeRex implements pointer tracking by analyzing all pointer dereferences and propagating labels between symbolic values. More specifically, TeeRex uses a taint-style analysis annotating every value loaded from memory with the address, where the value was loaded from. This allows TeeRex to determine the source of a value, e.g., whether a function pointer used for an indirect call was loaded from enclave, host memory, or loaded via a parameter passed to the ECALL function.

Furthermore, TeeRex places hooks on Intel SGX SDK functions that are used to validate whether an address is within secure memory. Whenever the enclave uses one of these functions, TeeRex forks the symbolic execution into two states: one where the address is within enclave memory and one where the address is outside enclave memory. This information is used by TEEREX to assess whether a bug is exploitable and report identified vulnerabilities more accurately.

Vulnerability Report: Finally, TEEREX produces a vulnerability report, which contains (1) the type of the vulnerability, (2) the location in the binary, (3) the controlled pointer and its position in the attacker-controlled input and (4) an execution trace to reach the vulnerable instruction. The vulnerability report provides sufficient detailed information to an analyst for constructing a proof-of-concept exploit, even for closed-source enclaves (see Section 5).

4.2 Challenges

Next, we will describe several challenges when applying symbolic execution to enclave binaries and how our design tackles them.

C1: Accuracy and Scalability. Enclaves built with the Intel SGX SDK define only a few (often one) entry point in the thread-control structure (TCS). This entry point is the trusted runtime (trts) that is responsible for setting up the enclave execution environment, calling exception handlers, and multiplexing ECALLs. For this, the arguments of an ECALL are packed by the untrusted runtime (urts) to be unpacked upon entering the enclave by the trts. This introduces an additional layer of pointer indirection for all ECALL parameters. The specifics of the enclave management in the trts are heavily dependent on the intrinsics of the SGX instructions and the enclave's internal metadata, which are not present in the emulated environment. This introduces high complexity and a major challenge for a symbolic execution analysis because (1) the enclave initialization routines result in many memory accesses through symbolic addresses, which is a notoriously hard problem for symbolic execution engines in general [5, 12], and (2) due to the low-level nature of the trts code the symbolic execution lacks semantic information about the execution context when it finally reaches the ECALL functions. Hence, it is not feasible to map symbolic memory ranges to ECALL parameters once the symbolic execution analyzes the actual ECALL function.

However, symbolically executing the whole trts code is conceptually uninteresting for identifying vulnerabilities in ECALLs as the trts is independent of ECALLs. As such, we designed TEEREX in such a way that it is able to skip symbolic execution of the trts and instead targets ECALL functions directly. To do so, TEEREX first extracts the ECALL table from the enclave binary. Next, symbolic execution is started at the beginning of every ECALL separately. This allows TEEREX to produce very accurate vulnerability reports as it is now possible to directly control the arguments passed to the ECALL function. At the same time, it reduces the overhead of executing code that is not meaningful for identifying exploitable bugs in enclaves. Furthermore, starting the analysis for each ECALL function separately and skipping the SDK runtime components allows parallelization of the symbolic execution process. Note that ANGR is originally restricted to one thread due to the limits of the Python implementation.

C2: Standard Memory Functions. Another source of path complexity arises from the standard memory functions. Methods like `memcpy` or `malloc` are reimplemented in ANGR as so-called *SimProcedures* at a higher level. Instead of symbolically executing the binary code of a function like `memcpy`, ANGR instead invokes the corresponding SimProcedure to update the symbolic state. This is possible because most applications load these functions dynamically from a library in the system, which can be easily intercepted. However, the self-contained enclave code comes with its own trusted version of these functions. As such, TEEREX searches the enclave code for trusted versions of these functions and places hooks to invoke the corresponding SimProcedure instead.

C3: Unsupported CPU Instructions. Since SGX has been recently integrated into new Intel CPUs, there are several advanced instructions included in enclave code that are unsupported by the symbolic explorer either because they are too new or too complex to be easily implemented symbolically. This includes the primary SGX instruction `enclu` to enter/exit an enclave, but also the non-SGX-specific instructions `rdrand` and `xsave/xrstor`, which are used in OCALLs to save and restore all registers from memory when the execution passes the host-enclave boundary. To tackle this challenge, we avoid executing the SGX-specific entry instructions but directly invoke the ECALL functions during the symbolic execution. We deal with other unsupported, but frequently executed instructions, by hooking into them. The hooks reimplement and emulate the instructions in Python to update the symbolic state accordingly.

C4: Global State of Enclaves and Chains of ECALLs. Enclaves can be entered multiple times at different ECALLs with different attacker-controlled input data, with each of the calls altering the internal global state of the enclave. Hence, the control-flow of an ECALL does not only depend on its arguments, but also on all prior invoked ECALLs. Taking this into account, an accurate symbolic exploration of an ECALL requires exhaustive knowledge about the effects of all ECALLs. To address this issue, TEEREX analyzes each ECALL individually and treats all (secure) global state (i.e., global variables in the `data` and `bss` sections) of an enclave as initialized with unconstrained symbolic values. This allows our tool to also explore paths of an ECALL that are not reachable with an enclave's initial global state. However, the global state is typically not fully attacker-controlled but rather initialized to zero or changed to some value by a different ECALL. Thus, the assumption that the global state is completely unconstrained can potentially lead to a situation, where our TEEREX wrongly reports an attacker-controlled jump or write although the state might be limited to only safe values. Nevertheless, the analysis results are still useful because they can lift limited exploitation primitives (e.g., null-pointer dereference or write to an arbitrary address with a fixed value) to full control-flow hijacking attacks (see Section 5.5 for an example).

4.3 Vulnerability Detection Components

We implemented three major vulnerability detection components in TEEREX: (1) attacker-controlled branches (control-flow hijacking), (2) controlled writes, and (3) NULL-pointer dereferences. To analyze an enclave, TEEREX first reads the ECALL table from an enclave and symbolically executes the ECALL functions sequentially. We pass fully symbolic arguments to each ECALL function and symbolically explore its code. Our symbolic execution tool currently supports detecting two major classes of exploit primitives: control-flow hijacking and controlled writes. In addition, we detect if the enclave dereferences a NULL-pointer.

Control-Flow Hijacking. To identify control-flow hijacking vulnerabilities, TEEREX searches for program paths, where the enclave utilizes attacker-controlled data as a jump target. To be more precise, TEEREX detects and reports unconstrained jumps that are encountered during symbolic execution.

Anything that is attacker-controlled (i.e., input and the whole address-space outside of enclave memory) is marked as an unconstrained fully symbolic value during symbolic execution. This means that when the ECALL uses one of its symbolic arguments as a jump target, it will jump to an unconstrained symbolic value. Furthermore, loading the jump target from uninitialized memory also leads to loading an unconstrained symbolic value. On the other hand, if the enclave validates the jump target pointer to be within a certain set of allowed values, then the symbolic execution engine will gather constraints on the symbol representing the jump target during the analysis of the validation code. The jump target is now tightly constrained to be within a certain set of allowed—assumed to be safe—values, which will not trigger an alarm. However, any use of an unconstrained pointer as a jump target results in TEEREX reporting a controlled jump, as here no prior validation was found and the attacker has full control over the jump target.

Controlled Write. TEEREX searches for writes to arbitrary (unconstrained) memory addresses during symbolic exploration. Therefore, we track every pointer dereference and propagate labels similar to taint analysis [17, 72]. This makes it possible to infer the relation of a corrupting pointer to the input arguments. This includes the levels of indirection and corresponding offsets. When a pointer is utilized for a memory write, TEEREX checks whether the address is related to attacker-controlled memory. If the address was loaded based on input arguments, the attacker can directly control the address used in the memory write instruction. Furthermore, TEEREX uses the solver of the symbolic execution engine to test whether the address of a write can possibly point to an arbitrary memory location inside of the enclave memory. If so, we can infer that we discovered an arbitrary write gadget.

Any write to an arbitrary address must be considered as a vulnerability regardless of whether the value written is attacker-controllable. For instance, a controlled write to an arbitrary address with a fixed single byte value (e.g., 0x0a) is often sufficient to corrupt a pointer in enclave memory. With complete control of the address space in the SGX setting, the attacker can map memory pages at almost any address. As a result, it is sufficient if the attacker can partially corrupt a pointer in enclave memory and make it point to insecure memory, which still is a valid memory location (see exploit in Section 5.5 for an example). As such, TEEREX reports any memory write to an attacker controlled address, regardless of the value written.

NULL-Pointer Dereference. On the x86 architecture, the page at address 0 (NULL) in the virtual address space of a user space program is a legitimate address. However, in C/C++, pointers are typically initialized to the null-pointer and many functions from standard libraries return the null-pointer to indicate an error. As such, dereferencing a null-pointer is a common problem in C/C++ code but typically not considered critical as the null page is not mapped, i.e. the process only crashes when trying to dereference a null-pointer. On the contrary, in the SGX setting a null-pointer dereference is critical since the null page is typically not within trusted enclave memory. As such, we need to consider it as controlled by the attacker. TEEREX analyzes every memory access and checks whether the address is pointing to the zero page mapped at address 0 (typically $<$ 0x1000). If this is the case, TEEREX reports that the code is dereferencing a null-pointer.

5 Enclave Analysis Results

To evaluate the effectiveness of TEEREX on real-world enclaves, we gathered a dataset consisting of open-source and proprietary public enclaves. Table 1 provides an overview of all the enclaves we analyzed with TEEREX. Our dataset contains enclaves developed by well-known companies such as Intel and Baidu. We also included SGX-protected fingerprint software that is utilized in Dell and Lenovo laptops. Note that it was highly challenging finding projects utilizing the SGX technology. We assume this is due to the fact that SGX is a rather new technology, hardware-support on client machines is still not widely available, and as such, SGX is primarily used in cloud settings where the enclave is simply not publicly available.

We use the following methodology for analyzing the enclaves in our dataset: first, we analyze the enclaves with TEEREX. Second, using the vulnerability report of TEEREX, we verify the vulnerabilities, perform root-cause analysis to

Project Name	Analyzed Version	Exploit	Fixed Version(s)	Source Code	Target	Number of ECALLs
Intel GMP Example [38]	9533574f95b97	✓	0491317b4112b	✓	Linux amd64	6
Rust SGX SDK's tlsclient [25, 73]	1.0.9	✓	f975a19982740	✓	Linux amd64	8
TaLoS [2, 27]	bb0b61925347b	✓	not planned	✓	Linux amd64	207
WolfSSL Example Enclave [74]	d330c53baff52	✓	1862c108d7e3b	✓	Linux amd64	22
Synaptics SynaTEE Driver	5.2.3535.26	✓	[20, 34, 49, 65]	×	Windows amd64	2 (76)*
Goodix Fingerprint Driver	2.1.32.200	✓	[21, 24]	×	Windows amd64	56
SignalApp Contact Discovery [53]	1.13	×	-	✓	Linux amd64	7

Table 1: Dataset of public enclaves and their susceptibility to exploitation.
* One ECALL immediately branches to 75 different actions.

identify the vulnerability, and finally construct a proof-of-concept (PoC) exploit. In our PoC exploits, we aim to hijack the instruction pointer while the processor is in enclave mode. Given this capability, an attacker can utilize existing code-reuse attack techniques to achieve arbitrary code execution [7, 62]. By constructing such a PoC exploit, we gain confidence that the issues discovered by TEEREX are indeed serious vulnerabilities.

For our PoCs we assume the standard SGX adversary model [18, 55] in which the attacker has full control over the user space and operating system/hypervisor. More specifically, our current PoCs assume a standard OS (Ubuntu 18.04 and Windows 10), which are configured or patched to allow the attacker to map the page at address 0. The enclaves are all compiled with the standard Intel SGX SDK. Note that our PoCs do not need to bypass ASLR since the untrusted OS selects the address space layout of the enclave. Our PoC exploits attempt to get full control over the instruction pointer, which is typically sufficient to perform arbitrary code execution [7].

Using TEEREX, we identified vulnerabilities in all of our analyzed SGX enclaves except the SignalApp contact discovery service [53]. In our analysis, we observed that the ECALL interface of this enclave is comparatively small and simple. For each of the vulnerable enclaves, we successfully developed PoC exploits of which all enable full instruction pointer control. We performed responsible disclosure for all vulnerable enclaves listed in Table 1. All vendors have acknowledged our findings and all vendors, except for one, developed fixes for the vulnerabilities we reported.

We also performed root-cause analysis on our findings and identified several problematic code patterns that lead to vulnerabilities. Table 2 shows an overview of the results of our analysis. We identified and successfully abused all different types of exploit primitives that TEEREX detected. Based on our root-cause analysis we identified bug classes specific to SGX that easily lead to vulnerabilities in enclave code. In what follows, we discuss in detail the vulnerable enclaves and bug classes we identified.

```
void e_mpz_add(mpz_t *c_unsafe,
               mpz_t *a_unsafe,
               mpz_t *b_unsafe) {
  mpz_t a, b, c;
  /* [computation code omitted] */
  // mpz_set copies the underlying buffer
  // of the biginteger "c" to the buffer pointer
  // contained in the "c_unsafe" variable
  mpz_set(*c_unsafe, c);
}
```

Figure 4: Excerpt of the vulnerable code in the *Intel GMP Example* enclave.

5.1 Intel GMP Example

Intel provides the *GNU Multiple Precision Arithmetic Library* for SGX and a corresponding demo application. The enclave code takes two GMP big integers as parameters, performs an arithmetic computation, and returns the result. TEEREX identified an arbitrary write vulnerability in the enclave code, which we used in our PoC exploit to gain arbitrary code execution. The data structure behind the GMP big integer internally utilizes a pointer to refer to an underlying buffer that contains the variably-sized data of the big integer. TEEREX identified that this pointer is not sanitized allowing a memory write to an arbitrary location. This vulnerability shows how likely it is for SGX developers utilizing a third-party library, to miss validating a pointer inside of opaque data structures.

The problem behind the vulnerability is that the numbers passed to the enclave are GMP big integer objects representing arbitrary large integers. The GMP big integer data structures utilize dynamically allocated storage internally; they contain a pointer to the underlying buffer that stores the actual integer value. However, the enclave fails to properly validate the pointer inside of the GMP data structure. Figure 4 shows part of the vulnerable code: the enclave receives three big integer parameters. The first one, called `c_unsafe`, is used as an output parameter. The enclave uses functionality of the GMP library that is not SGX-aware: the `mpz_set` function. As such, the library function simply copies the output to the attacker-controlled underlying buffer of the `c_unsafe` big integer. This neglects the fact that the underlying buffer of this

		Intel GMP Example	Rust SGX SDK's tlsclient	TaLoS	WolfSSL Example Enclave	Synaptics SynaTEE Driver	Goodix Fingerprint Driver
Bug Classes	P1: Passing Data-Structures with Pointers	●	●	●	-	●	●
	P2: Returning pointers to enclave memory	●	●	●	●	-	-
	P3: Pointers to Overlapping Memory	-	●	-	-	-	-
	P4: NULL-Pointer Dereferences	-	-	●	-	●	●
	P5: Time-of-Check Time-of-Use	-	-	●	-	-	-
Exploit Primitive	Control-Flow Hijack	-	●	●	●	●	●
	Controlled Write	●	-	-	-	●	●
	NULL-pointer Dereference	-	-	●	-	●	●

Table 2: Overview of results of our analysis of public enclave code. Some patterns are not applicable for every enclave, because the relevant code constructs are not used or the source is unavailable.

big integer can actually point to arbitrary memory, including enclave memory.

This vulnerability allows an attacker to perform an arbitrary memory write, with controlled content and controlled size. TEEREX identifies the arbitrary write vulnerability in multiple ECALLs. They all share the same structure as the one depicted in Figure 4. In our proof-of-concept exploit, we abuse the `e_mpz_add` ECALL: we set the value of the underlying buffer of the big integer parameter `a_unsafe` to our payload, the big integer `b_unsafe` to a big integer initialized as 0, and the underlying buffer of `c_unsafe` to our target address for the arbitrary write. We choose an address on the enclave stack that points to a return address used by the enclave. This effectively allows us to write a ROP-payload directly onto the enclave stack.

Intel acknowledged the problem, updated their documentation, and fixed the issue by using serialization: instead of passing pointers to GMP structures, the demo code now serializes GMP big integer objects to strings and passes those strings over the host-to-enclave boundary. The enclave then deserializes the data structure, computes the result, and finally returns the serialized result back to the host application. Since no longer GMP big integer pointers are passed between the host and enclave, this fixes the vulnerability and removes the problematic pattern *Passing Data-Structures with Pointers (P1)* from the enclave code, which is defined in the following:

P1: Passing Data-Structures with Pointers. This type of vulnerability occurs due to complex data types in C/C++ that are using pointers as their primary mechanism to form complex data structures like lists, trees, or maps. When programming with the Intel SGX SDK, the interface provided by an enclave allows utilization of complex data types using pointers. However, currently the Intel SGX SDK does not automatically perform a recursive copy/validation of pointer-heavy data structures. As a consequence, it becomes dangerous to pass data structures containing pointers to an enclave. Any data structure containing pointers must be treated the same way as pointers annotated with the `[user_check]` attribute.

5.2 WolfSSL Example Enclave

WolfSSL [75] is a small TLS/SSL library without external dependencies designed for embedded devices and applications that require to be small and self-contained. It also features SGX support. The wolfSSL project offers an enclave that showcases how to use the wolfSSL TLS library within SGX. The enclave allows the host application to terminate a TLS connection within the SGX enclave thereby protecting all cryptographic secrets used by TLS. However, the enclave exposes a large subset of the WolfSSL API via the ECALL interface. We analyzed the enclave with TEEREX and discovered a control-flow hijacking primitive in the enclave. Our root-cause analysis revealed the following pattern, which is common to all the TLS enclaves we analyzed.

P2: Returning pointers to enclave memory. We observed that many enclaves provide functionality to allocate a resource in secure memory, e.g., a TLS session or a file object, and then return a reference to this resource to the host application. The next time the host application attempts to use this resource, the corresponding function of the enclave is called with that reference as a parameter. In C/C++ code, this is typically achieved by returning and passing a pointer to the object containing the resource's data. The enclave typically validates that the given pointer indeed points to secure memory.

In the case of wolfSSL, the legacy API of the TLS library was almost directly accessible through the ECALL API of the WolfSSL Example Enclave, only secured by the in-secure-memory check, which still entailed passing the pointers of the TLS context, TLS session, and I/O buffer objects between host and enclave. These data structures are part of a legacy

```
/* ECALL Definition in EDL */
// a pointer to enclave memory returned
public WOLFSSL* enc_wolfSSL_new([user_check] WOLFSSL_CTX* ctx);
// pointer is passed to enclave
public int enc_wolfSSL_connect([user_check]WOLFSSL* ssl);
// ...
```

```
/* C Source Code */
typedef int (*CallbackIOSend)(WOLFSSL *ssl, char *buf,
                              int sz, void *ctx);
/* WolfSSL session type */
struct WOLFSSL {
    WOLFSSL_CTX*    ctx;
    /* ... */
    // attacker-controlled function pointer!
    CallbackIOSend  CBIOSend;
}
// ...
int enc_wolfSSL_connect(WOLFSSL* ssl) {
 ① if(sgx_is_within_enclave(ssl, wolfSSL_GetObjectSize()) != 1)
        abort();
    /* ... */ }
```

Figure 5: Relevant parts of the EDL definition and C source code of the *tlsclient* enclave. Note the insufficient validation ①.

API which were not designed with a split trust model in mind and it is very hard for the enclave to thoroughly validate the pointers forwarded to the legacy interface. Figure 5 shows the definition of the ECALL interface: a pointer to a `WOFLSSL` structure is passed with the `[user_check]` attribute. Note, that the `WOLFSSL` data structure contains a function pointer used for issuing callbacks in the TLS library (`CBIOSend`). TEEREX identified a control-flow hijacking primitive by passing a fake `WOLFSSL` data structure with an attacker-controlled `CBIOSend` function pointer.

However, the WolfSSL Example Enclave still implements a pointer validation routine: it validates that the pointer does point to enclave memory (Figure 5: ①). However, this pointer validation is not sufficient to protect the enclave. It is common that an attacker can actually control parts of the enclave memory, simply by providing input arguments. For example, an attacker can abuse a different ECALL with a buffer parameter to force the enclave to copy arbitrary data into enclave memory. In our PoC exploit, we abused the function `enc_wolfSSL_CTX_use_PrivateKey_buffer` to copy a fake `WOLFSSL` structure into unrelated enclave memory (a simple buffer). Thereafter, we call the function `enc_wolfSSL_connect`, which uses the attacker-controlled `CBIOSend` function pointer in the fake data structure, which now resides in secure memory.

This could either be fixed by using session identifiers as it was done by the Rust SGX SDK's tlsclient enclave (cf. Section 5.3) or—to not change the external API—by saving all created session pointers in secure memory and only accepting these known pointers.

```
/* ECALL Definition in EDL */
public void* tls_client_new()
public int tls_client_write(
    [user_check]   void* session,
    [in, size=cnt] char* buf,
                   int cnt);
```

```
// Rust Source Code
pub extern "C" fn tls_client_write(
                session: *const c_void,
                bu: * const c_char,
                cnt: c_int)  -> c_int {
 ① if session.is_null() {
        return -1;
    }

 ② if rsgx_raw_is_outside_enclave(
              session as * const u8,
              mem::size_of::<TlsClient>()) {
        return -1;
    }
    rsgx_lfence();

    let session = unsafe { &mut *(session as *mut TlsClient) };
```

Figure 6: Vulnerable Rust code: Check ② can be bypassed.

5.3 Rust SGX SDK's tlsclient/server

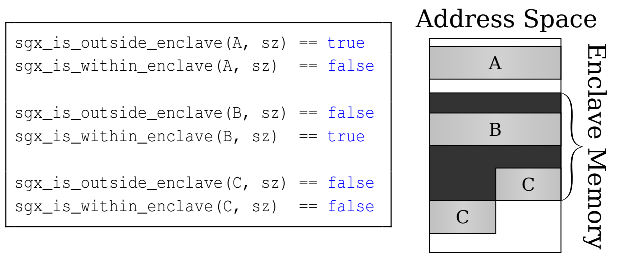
The Rust SGX SDK [25] aims at introducing memory safety for SGX. As such, enclaves developed with this framework should very unlikely suffer from memory corruption bugs. To validate this, we analyze code shipped with the Rust SGX SDK that shows how to run a TLS server and client inside of an SGX enclave. The code consists of two similarly structured applications and enclaves that interconnect using TLS to send an HTTP request. This shows how secure communication can be achieved while secret keys remain in protected memory. Since both applications are similar in terms of their enclave interfaces, we only discuss the *tlsclient* enclave. The enclave API consists of functions to create a new TLS session and then utilize the session to send and receive data securely. TEEREX discovered a control-flow hijacking primitive in the enclave function `tls_client_write` that abuses the session pointer parameter of the ECALL. The root cause for the vulnerability of this enclave is the same pattern that already made the WolfSSL Example Enclave (Section 5.2) vulnerable (*Returning pointers to enclave memory (P2)*). The TLS session object is allocated within *enclave memory* with the `tls_client_new` function and then passed to further API calls like `tls_client_write`. The pointer has to be marked as `user_check`. Otherwise, the SGX SDK would reject the raw pointer. However, there is a variable nested in the `TLSSession` object that contains a pointer to a virtual method table (vtable) for dynamic dispatch. By controlling the pointer to the object, the attacker controls the pointer to the virtual method table and gains full control over the target of an indirect call.

The enclave code, as shown in Figure 6, implements two pointer validation checks on the session pointer: (1) the pointer is checked to be not null ① and (2) not to be outside of the enclave ②. However, the check at ② is not sufficient to protect the enclave since there are two possible bypasses. First, the attacker can abuse a different ECALL to copy attacker-controlled data from the host application into the enclave memory (cf. Section 5.2). Second, the check at ② neglects that there are three memory states: outside, within the enclave, and partially inside the enclave. Hence, `outside_enclave` and `within_enclave` are not strictly inverse, both return `false` for any memory that is neither strictly outside nor strictly within the enclave. The intention of the enclave developer for check ② was to assess whether the session pointer does indeed point to memory inside of the enclave, i.e. return an error if it is not strictly within (`if ! rsgx_raw_is_within_enclave(...) return -1;`). This error belongs to the following pattern.

P3: Pointers to Overlapping Memory. For validating that an object is in secure memory, the Intel SGX SDK provides two functions: `sgx_is_within_enclave` and `sgx_is_outside_enclave`. These functions check whether a memory area is *strictly* outside or inside enclave memory. However, they return unexpected results when handling edge-cases, where a memory buffer is overlapping both areas. Figure 7 shows three different scenarios with buffers located either outside, inside, or outside as well as inside enclave memory. The validation functions from the Intel SGX SDK return `false` for buffers that are overlapping both memory areas.

In the case of the Rust SGX SDK's tlsclient, we can abuse the buggy check in ② to bypass the pointer validation routine in our PoC exploit. We allocate a page in the virtual address space right before the first page of enclave memory. Thereafter, we place a fake `TLSSession` object such that the last byte of the object is still part of enclave memory (i.e., the overlapping case). This construction bypasses the validation at ② since the memory is not strictly outside enclave memory. However, the important part—the address of the vtable—is still stored in untrusted host memory. Hence, we can fully control the target of an indirect jump and launch a code-reuse attack. Our findings demonstrate that using a memory-safe language like Rust does not automatically ensure memory-safe enclaves. That is, the entire software stack must be guaranteed to be memory-safe.

The developers of the Rust SGX SDK acknowledged the problem and promptly updated their code. Akin to our suggestions to the developers, the enclave code now utilizes session identifiers instead of pointers to identify TLS sessions; similar to using file descriptors on Unix-like systems. Upon session creation in `tls_client_new`, the pointer to the TLS session object is now inserted into a hashmap, which is then used to map the identifier in subsequent ECALLs. Hence, no pointers are passed on the host-to-enclave boundary. This drastically reduces the attack surface of the enclave and eradicates both the vulnerability pattern *Returning pointers to enclave memory (P2)* and *Pointers to Overlapping Memory (P3)*.



```
sgx_is_outside_enclave(A, sz) == true
sgx_is_within_enclave(A, sz)  == false

sgx_is_outside_enclave(B, sz) == false
sgx_is_within_enclave(B, sz)  == true

sgx_is_outside_enclave(C, sz) == false
sgx_is_within_enclave(C, sz)  == false
```

Figure 7: Possible buffer locations in SGX.

5.4 TaLoS

The open-source enclave TaLoS supports terminating TLS inside of SGX enclaves within production webservers such as the Apache webserver [27]. To achieve this, TaLoS introduces SGX specific patches to the *libressl TLS implementation*. The enclave exposes almost the entire TLS API of libressl over the ECALL interface, which utilizes many pointers that are marked as `[user_check]`. As such, this enclave contains the vulnerability patterns *Passing Data-Structures with Pointers (P1)* and *Returning pointers to enclave memory (P2)*. However, the enclave does not simply return a raw pointer as it is the case for the enclaves WolfSSL Example Enclave (Section 5.2) and Rust SGX SDK's tlsclient/server (Section 5.3). Instead, it uses a shadowing mechanism that synchronizes selected fields (e.g. of the primary `SSL` data structure) between the trusted and untrusted world. This allows the host application to access some fields of the data structure, while keeping the actual copy in enclave memory [2]. This design choice was taken to allow unmodified web servers to interact with the SGX wrapped TLS API. In principle, the shadowing mechanism is a legitimate pointer validation mechanism and allows the enclave to verify pointers passed by the untrusted host application. However, the exposed API is quite comprehensive and TEEREX discovered an ECALL that uses a function pointer in its data structure, where shadowing was missing. This underlines the need for automated analysis tools, such as TEEREX, to automatically identify missing pointer validation code. Furthermore, we identified many potential sources for vulnerabilities in the code that handled the shadowing mechanism. The shadowing mechanism failed to take into account that the *NULL* pointer is a valid pointer in the SGX context.

P4: NULL-Pointer Dereferences. The special *NULL* (or *nullptr*) value is used in C/C++ code to signal that a pointer is not referencing any object. However, it is represented by the numeric value 0, but on x86 systems (using virtual memory) the address 0 is a valid address. Typically, there is no valid memory mapped to address 0. Hence, any accidental NULL pointer dereference results in a crash of the process (SEGFAULT). However, a malicious host program or OS can map valid data at the page at address 0. Thus, a NULL pointer dereference turns into a valid load and a bogus value from the page at address 0 is read instead of crashing the enclave.

```
BIO* ecall_SSL_get_rbio(SSL *out_s) {
①// out_s is not checked, can be in enclave memory
  /** Shadowing Mechanism **/
  hashmap* m = get_ssl_hardening();
  // returns NULL for invalid out_s
②SSL* in_s = hashmapGet(m, out_s);
  // copy arbitrary enclave memory to the NULL page
③SSL_copy_fields_to_in_struct(in_s, out_s);
④/* [...] libressl logic */
  // copy from the NULL page to arbitrary enclave memory
⑤SSL_copy_fields_to_out_struct(in_s, out_s); // [...]
```

Figure 8: Relevant parts of the EDL definition and C source code of the TaLoS enclave.

This is similar to the kernel scenario, where the address 0 is typically a valid address in the user space. As a mitigation, many OS kernels disallow mapping any memory at address 0. However, for NULL pointer dereferences inside of SGX enclaves, there is currently no mitigation available, since the OS is considered untrusted in the SGX threat model. As such, an enclave must assume that the page at address 0 is mapped into the address space.

Figure 8 shows the relevant code that contains a NULL-pointer dereference. This snippet contains two mistakes: first, the pointer parameter `out_s` is supposed to point to the outside version of the TLS structure. However, the enclave does not validate that the `out_s` actually points to outside enclave memory (①). As such, an attacker can simply pass some memory location inside of the enclave memory. The function call at ② retrieves the shadowed `SSL` object that is within enclave memory. However, when passing a bogus pointer this function will return a NULL-pointer to signal an error, which is not checked by the enclave. The function call at ③ is the synchronization function that copies selected fields from the outside `SSL` structure to the inside structure. In case of an attack, the `out_s` pointer does point to an arbitrary location inside of the enclave, e.g., a secret key and `in_s` points to the NULL-page. Thus, the enclave copies arbitrary data from enclave memory to the NULL-page resulting in an *arbitrary read exploit*.

Furthermore, the same bugs shown in Figure 8 can also be turned into an arbitrary write exploit primitive: for the function call marked with ⑤, the enclave synchronizes back the fields of the inside structure to the outside copy. In our NULL-pointer dereference attack, the variable `in_s` points to the NULL-page, while the variable `out_s` points to some arbitrary enclave memory location. However, we have to overcome a race condition challenge to also control the value that is written. Recall that the enclave first reads the value from enclave memory and thereafter writes the value to the NULL-page (③). Hence, it would write back the same value to enclave memory that was copied to the NULL-page. To tackle the race condition, we execute a different thread in the host application and change the contents of the NULL-page while the code between the two synchronization functions (④) is executed. This effectively gives an attacker the arbitrary write capability. Note that prior research has shown that it is trivial to win race conditions in the SGX threat model. Since the attacker is in full control of the OS and the scheduling of the enclave's thread, the attacker can even single-step through the enclave code [70].

P5: Time-of-Check Time-of-Use. Enclaves run in an environment where it is easy to introduce Time-of-Check Time-of-Use (TOCTOU) bugs. While the enclave developer can limit how many threads can concurrently enter an SGX enclave, the enclave developer has no control over the untrusted and possibly malicious OS. When accessing host application memory, the enclave must assume that a separate host application thread can always change any content in the untrusted memory area. As a consequence, an enclave cannot validate any data structures outside of the enclave memory. In the TaLoS example, we utilized a race condition similar to TOCTOU bugs to exploit the enclave.

5.5 Synaptics SynaTEE Driver

Synaptics recently started to utilize SGX enclaves to securely process fingerprint data on Windows in Lenovo and HP laptops. The closed-source fingerprint driver contains a user space component with an SGX enclave. TEEREX discovered a control-flow hijacking primitive that can be exploited due to a NULL-pointer dereference (cf. TaLoS in Section 5.4). The enclave utilizes a pointer in the global state, which is initialized as a NULL pointer. Normally, this pointer would be initialized to point to a data structure inside enclave memory, but the attacker could potentially load the enclave and trigger the NULL pointer dereference without initializing this pointer.

We chose not to exploit the NULL-pointer dereference since the latest Windows versions strictly prohibit mapping a page at address 0. That being said, the SGX threat model assumes that the attacker has full control over the OS, i.e., an attacker with OS privileges can disable this mitigation in the Windows kernel. We demonstrated the feasibility of this in our PoC exploit for the Goodix enclave (see Section 5.6). To avoid patching the Windows kernel and to make our PoC exploit more portable, we utilize a second finding of TEEREX: a limited write exploit primitive due to an improperly sanitized pointer heavy data structure that is passed to the enclave. This exploit primitive allows us to write a fixed byte-value to an arbitrary address. We used this in our PoC exploit to first corrupt the pointer in the global state of the enclave to make it point to a fixed address in untrusted host application memory. Next, we mapped our exploit payload to this fixed address thereby avoiding allocation of a page at address 0.

We chained two exploit primitives in our PoC Exploit, both discovered by TEEREX. The vulnerabilities we identified are due to the code patterns *Passing Data-Structures with Pointers (P1)* (cf. Intel GMP Example in Section 5.1) and *NULL-Pointer Dereferences (P4)* (cf. TaLoS in Section 5.4).

5.6 Goodix Fingerprint Driver

The fingerprint reader driver is shipped on recent Dell laptops and uses SGX enclaves to process biometric data. The black-box analysis of TEEREX discovered multiple limited controllable write primitives to arbitrary addresses. For our exploit, we combined two of them to achieve a full control-flow hijack.

The first primitive, denoted as C_{16}, discovered by TEEREX copies a 16 bit value loaded from a NULL-pointer (see Section 5.4) to the address supplied in the ECALL argument by an attacker. We patched the Windows kernel using a kernel debugger and disabled the check that prevents Windows user space applications to map the address 0, allowing us to exploit the NULL-pointer dereference in the enclave. While the attacker controls the value and the address in the first primitive C_{16}, due to the limited size of the controlled value, this primitive can only partially overwrite the instruction pointer. Although this partial overwrite is often sufficient [26], we combine it with a second primitive also discovered by TEEREX to achieve a full (64-bit) arbitrary write. The second primitive, denoted as F_{64}, is a limited write primitive that copies a 64 bit value loaded from a fixed address A that is within secure memory to an attacker-controlled pointer in the ECALL argument. We execute primitive C_{16} four times to copy a full 64 bit value in 16 bit chunks to the address A, which is used in primitive F_{64}. This gives us control over the 64 bit value that is written by F_{64}. Subsequently, we can then use primitive F_{64} to overwrite, e.g., a return address in secure memory.

The analysis of this enclave demonstrates that the vulnerability report produced by TEEREX (cf. Section 4) provides sufficient information to easily create a PoC exploit for enclaves where source code is not available. We only needed to combine two primitives and for both TEEREX reported the source and target addresses of the writes and the necessary ECALL arguments.

5.7 Vulnerability Disclosure

We provided the developers of all the vulnerable enclaves a detailed report explaining the problematic code patterns, a working PoC exploit, and suggested fixes. All of them confirmed our findings. We supported the enclave developers by validating the patched versions with TEEREX. Table 1 shows the version number of the fixed enclave code, as far as they were available to us. As a response to our report, Intel changed the code of the Intel GMP Example enclave to use a serialization-based approach for parameters crossing the host-to-enclave boundary. Since serialization avoids passing raw object pointers at the host-to-enclave boundary, the vulnerabilities were successfully fixed. The developers of both, the WolfSSL Example Enclave and the Rust SGX SDK's tlsclient, followed our suggestions and stopped using pointers as resource references. Both enclaves now utilize integer identifiers to look up the respective TLS session objects in a table inside of enclave memory. The original developer of the TaLoS acknowledged our findings, but notified us that he lacks the resources to develop fixes. As such, this project must now be considered a deprecated and abandoned research project. Synaptics issued CVE-2019-18619 [20] for the vulnerabilities we reported. Given the high sensitivity of biometric data, they promptly developed a patch. After coordinated disclosure with OEM vendors patches were published in July, 2020 (HP [34], Lenovo [49]). The security team of Goodix developed a patch that we successfully verified with TEEREX and Goodix issued CVE-2020-11667. As of July, 2020 patched drivers for Dell laptops are available [24].

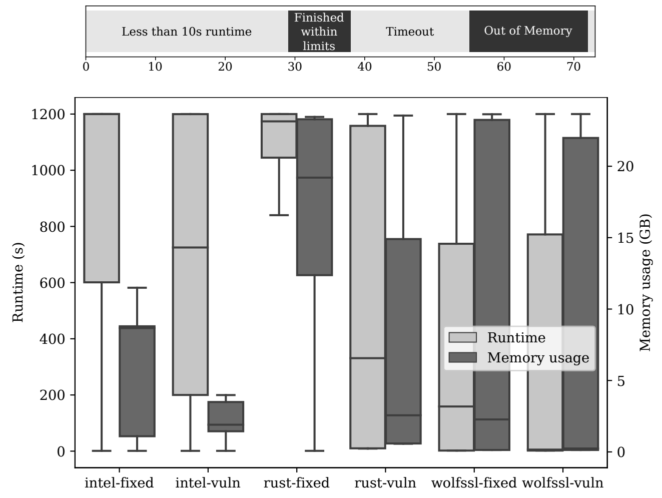


Figure 9: Runtime and memory usage of the benchmarked enclaves.

6 Performance and Accuracy

In this section, we analyze the efficiency and effectiveness of TEEREX. We focus our analysis on the three enclaves Intel GMP Example, Rust SGX SDK's tlsclient, and WolfSSL Example Enclave since for these (1) the source code is available, and (2) a patched version already exists. These insights allow us to compare TEEREX' behavior on the vulnerable and fixed enclaves and reason about the occurrences of false alarms.

6.1 Performance and Memory Usage

Our strategy is as follows: We analyze each ECALL using TEEREX for a maximum of 20 min using one CPU core up to a memory limit of 24 GB. The analysis was conducted on an AMD EPYC Processor with 3.7 GHz and 100 GB RAM allowing us to analyze up to 4 ECALLs in parallel. TEEREX utilizes angr version 8.20.1.7 running on CPython 3.6.9 and Ubuntu 18.04.4. All the exploitable primitives that we utilized in our PoC exploits are discovered within our time window of 20 min.

For the three enclaves (Intel GMP Example, Rust SGX SDK's tlsclient, and WolfSSL Example Enclave), we analyzed the 73 ECALLs in detail. The results are depicted in Figure 9: the average memory usage over all ECALLs of those enclaves is 8.8 GB ($\sigma = 9.8$ GB). The significant deviation for memory usage is mainly due to the highly variable size and complexity of the ECALLs. Out of the analyzed ECALLs 40 % finished within 10 s, 52 % finished within the given limits, and 48 % exceeded the limits (23 % by time, also 23 % by memory, and 1 % by time and memory).

Our analysis in Section 5 demonstrates that using this analysis strategy is sufficient to successfully uncover problematic code patterns. While symbolic execution is a powerful analysis technique, it requires high computing resources (CPU time and memory) to explore the state space of a program. Hence, it is only natural that the analysis of some of the ECALLs hits the resource limits we defined for the benchmarking experiments. However, during a security analysis of an enclave, the analyst can schedule more time and memory as needed for specific ECALLs. Furthermore, we did not yet implement all of the advanced techniques to improve the efficiency of symbolic execution that was proposed in prior work [4, 5, 12]. This was simply not necessary to discover vulnerabilities in our set of analyzed enclaves.

6.2 Accuracy and False Alarms

Since the analysis of TeeRex does focus on soundness rather than on completeness, the number of false alarms is rather small. Note that a complete false positive analysis is impossible as we lack any ground-truth, i.e., all of our findings are zero-day vulnerabilities and we are unable to provide any comparison of TeeRex to related approaches since there does not yet exist any other automated vulnerability discovery approach for SGX enclaves. Hence, we opted for the following strategy: we confirmed TeeRex' alarms by constructing PoC exploits and disclosing our findings to the affected vendors. After the vendors fixed the vulnerabilities, we manually verified that the enclaves' source code (for Intel GMP Example, Rust SGX SDK's tlsclient, and WolfSSL Example Enclave) does not contain further vulnerabilities. These patched enclaves give us a limited form of ground-truth as any finding in the updated enclaves is a false alarm.

In our analysis of the three vulnerable enclaves TeeRex produced 149 findings. By constructing a proof-of-concept exploit based on the findings of TeeRex, we confirm that those findings were indeed true alarms. We selected gadgets in shallow program paths containing the least conditions on the initial state (i.e., constraints on the enclave's pre-ECALL state) and then constructed a PoC exploit based on the selected gadgets.

Thereafter, we analyzed the patched versions of the enclaves. TeeRex confirmed that our original and exploited findings are not longer present in the patched enclaves. However, the analysis of TeeRex still produced 56 findings. Our root-cause analysis of those findings reveals a possible indicator of a false alarm in TeeRex' reports: global memory is treated as unconstrained symbolic value by TeeRex (see challenge C4 in Section 4.2). For example, the patched Intel GMP Example utilizes an initializing ECALL which sets up a function pointer in global memory. TeeRex discovered that other ECALLs do not check that function pointer before use. Due to the ECALL-centric analysis of TeeRex, the function pointer is considered unconstrained and a controlled jump is reported. However, in reality, the function pointer can only take fixed values. Thus, this finding on its own is not exploitable. On the other hand, in case TeeRex would have also discovered a controlled write primitive, we still would have been able to construct a proof-of-concept exploit. The false positives that we encountered in the other patched enclaves (Rust SGX SDK's tlsclient and WolfSSL Example Enclave) are caused by the same issue. In our future work, we plan to annotate and filter such false alarms as *low severity* based on TeeRex' pointer-tracking component.

7 Discussion

Analyzing OCALLs. TeeRex puts its focus on ECALLs as those are the prevalent way to pass data to enclaves. Further, since OCALLs are only reachable through ECALLs, their support is a precondition for OCALLs. Nevertheless, we plan to implement OCALL-support in our future work.

Handling the OCALL interface is particularly challenging due to the lack of semantic information. From a binary analysis point-of-view, an OCALL is not easily distinguished from a regular return from an ECALL, i.e., both utilize the *EEXIT* instruction to exit the enclave. As such, TeeRex will stop executing a program path in the enclave once an OCALL (or *EEXIT*) is reached and thus will not analyze any ECALL code beyond the first OCALL. To overcome this limitation, we utilize symbol information to detect OCALL invocations in TeeRex. If TeeRex discovers that an OCALL is executed, e.g., due to symbols and functions of the Intel SGX SDK, then TeeRex will skip the execution of the OCALL and set the return value of the OCALL to an unconstrained symbolic value. This allows TeeRex to continue the analysis after the OCALL with a rough over-approximation of the OCALL's effects since the actual semantics of the OCALL are not emulated. We leave the development of a heuristic to detect OCALLs on a binary-level without symbols as future work.

Manual Effort with TeeRex. TeeRex automatically detects vulnerabilities in enclaves. More specifically, TeeRex reports the exploit primitives resulting from the vulnerabilities.

For instance, TEEREX will show the location of a controlled write combined with the constraints (i.e., possible values) on the address, value, and path that leads to the write instruction. An analyst must then inspect the report and decide whether the findings or any combination of findings is exploitable, or if the alarm is a false positive. While the information reported by TEEREX is sufficient to construct PoC exploits, we plan to incorporate exploit generation schemes as proposed in prior work [3, 12, 32, 35] into TEEREX to automatically synthesize a malicious host application that reproduces the crash.

Fuzzing Enclaves. Coverage-guided fuzzing is another prominent technique to identify vulnerabilities in binary code [77]. In contrast to symbolic execution, fuzzing scales well to large software projects. As such, fuzzing would potentially allow analysis of large and complex enclave binaries to tackle the general problem of path explosion. On the other hand, applying fuzzing to SGX enclaves is not straightforward: (1) To ensure efficiency, fuzzing requires a sophisticated mutation strategy. However, mutation for the complex ECALL interface requires significant engineering effort. (2) Fuzzing relies on dynamic analysis tools to instrument binaries [9, 51], which are currently not available for SGX enclaves. In particular, integrating dynamic analysis tools is highly challenging when analyzing proprietary enclave binaries. Note that the protection mechanisms provided by SGX impede dynamic binary instrumentation. Further, static binary instrumentation often fails to accurately rewrite binaries. Consequently, we decided to rely on symbolic execution as it allows us to fully control the simulated environment. Further, it comes with additional flexibility significantly simplifying implementation and integration of symbolic vulnerability detectors. However, enabling hybrid fuzzing/concolic execution in TEEREX is worthwhile investigating for future work.

8 Related Work

The security research on privilege separation lead to system architectures that separate user from kernel space. However, several kernel vulnerabilities bypassed this separation simply because the kernel is not strictly separated from user space [14, 19, 28, 29, 45]. As a response, CPU vendors introduced hardware-based mitigation mechanisms, such as SMAP or SMEP [39], to enforce stricter separation. In fact, there are many parallels between the user/kernel space interface and the SGX host-to-enclave interface. That is, a higher privileged partition (the enclave) must carefully parse and validate any data that is written by the untrusted partition (the host application).

Prior work in this area introduced mechanism allowing a user space program to reliably execute in the presence of a compromised operating system [16, 50, 54, 59]. However, Checkoway et al. [13] have shown that existing legacy software cannot be simply retrofitted to such environments mainly because many kernel and operating system APIs implicitly assume that the kernel is the most trusted part of the system, e.g., in the threat model of a traditional Unix-like system the kernel is assumed to have full control over the code and data areas of any user space process. As such, existing software, such as most implementations of the C standard library, lack any validation of data passed from the kernel. So-called *Iago* attacks exploit this fact and show that a malicious kernel can easily corrupt memory of a user space process by returning bogus arguments from system calls. As we show in this paper, very similar issues apply to SGX enclaves; especially when legacy code is retrofitted to run inside SGX enclaves.

Hu et al. [36] showed that any software that is separated into equally-privileged but mutually untrusted partitions can be vulnerable to similar attacks. They presented an approach based on taint tracking and constraint solving to detect arbitrary write and possible TOCTOU vulnerabilities for a limited number of execution paths. In contrast, TEEREX utilizes full symbolic execution to identify arbitrary write primitives. Furthermore, TEEREX also discovers control-flow hijacking and NULL-pointer dereferences. TEEREX' analysis also includes scenarios, where the exploit depends on the global state of the target.

Recently, Van Bulck et al. [69] presented a security analysis of several TEE SDKs, whereas we focus on analyzing enclaves. They identified vulnerabilities in TEE SDKs using only manual code review. In contrast, we introduce an *automated* vulnerability detection framework for SGX enclave binaries, which additionally assists an analyst in assessing the vulnerability and constructing an exploit.

Many Android phones using ARM processors utilize the TrustZone trusted execution environment (TEE) to protect critical software. In contrast to SGX, TrustZone splits all privilege levels into a trusted and untrusted world, where the trusted OS has the highest privilege on the system. Machiry et al. [52] analyzed the attack surface of the privilege boundary between normal world and TEE. They identified a class of vulnerabilities caused by to the semantic gap between normal world and TEE. They allow unprivileged, untrusted user space applications (e.g., a sandboxed Android app) to abuse the TEE to compromise the normal OS (the Linux kernel). This type of vulnerability does not apply to SGX as enclaves have little privileges and are prohibited to interact with the OS. Harrison et al. [31] implemented a fuzzer based on full-system emulation of the TrustZone TEE including the trusted OS and trusted applications. The main challenge for analyzing ARM-based TEEs is the fact that a custom trusted OS, including required hardware, must be emulated. In contrast, SGX enclaves generally lack direct hardware access. Further, as discussed in Section 7, symbolic execution offers several advantages over fuzzing when analyzing SGX enclaves.

9 Conclusion

Intel SGX is a promising security technology to strongly isolate sensitive code and data into enclaves. However, implementing the host-to-enclave boundary securely is highly critical as the enclave processes and operates on input originating from untrusted memory space. To allow thorough security testing of this interface, we perform a systematic investigation on publicly available SGX enclaves. A major contribution of this paper is to introduce an automated analysis approach to determine vulnerabilities in enclaves. To do so, our approach develops a sophisticated symbolic execution framework that is able to analyze enclave binaries and produce detailed vulnerability reports to significantly simplify the construction of proof-of-concept (PoC) exploits. Our findings on public enclaves reveal vulnerabilities in two fingerprint drivers (by Synaptics and by Goodix), three TLS libraries, and a project published by Intel. For each, we constructed PoC exploits to confirm the severity of the vulnerability and perform control-flow hijacking allowing an attacker to subvert any confidentiality or integrity guarantees offered by the SGX enclaves. We analyzed the root causes of the vulnerabilities and identified vulnerability patterns that likely also affect privately deployed enclaves. Addressing our findings is crucial to allow secure deployment of SGX enclaves.

Acknowledgment

We would like to thank the affected vendors, Intel, Baidu, WolfSSL, Synaptics, Goodix, and the enclave developers for promptly acting upon our reports and developing patches. Furthermore, we especially thank our shepherd, Nathan Dautenhahn, for helping us to improve this work. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and under SFB 1119 – 236615297.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (2009). DOI: 10.1145/1609956.1609960.

[2] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. *TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves*. en. Tech. rep. 2017/5. Imperial College London, Mar. 2017. URL: https://www.doc.ic.ac.uk/research/technicalreports/2017/DTRS17-5.pdf.

[3] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. "AEG: Automatic Exploit Generation". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS*. 2011. URL: https://www.ndss-symposium.org/ndss2011/aeg-automatic-exploit-generation.

[4] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing symbolic execution with veritesting". In: *Commun. ACM* 59.6 (2016), pp. 93–100. DOI: 10.1145/2927924.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657.

[6] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[7] 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. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/biondo.

[8] Robert S Boyer, Bernard Elspas, and Karl N Levitt. "SELECT—a formal system for testing and debugging programs by symbolic execution". In: *ACM SigPlan Notices* 10.6 (1975). URL: https://dl.acm.org/citation.cfm?id=808445.

[9] Bryan Buck and Jeffrey K Hollingsworth. "An API for Runtime Code Patching". In: *Int. J. High Perform. Comput. Appl.* 14.4 (Nov. 2000). ISSN: 1094-3420. DOI: 10.1177/109434200001400404.

[10] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2008. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

[11] Nicholas Carlini and David Wagner. "ROP is Still Dangerous: Breaking Modern Defenses". In: *23rd USENIX Security Symposium, USENIX Security*. 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE, May 2012. DOI: 10.1109/SP.2012.31.

[13] Stephen Checkoway and Hovav Shacham. "Iago attacks: why the system call API is a bad untrusted RPC interface". In: *ASPLOS*. Vol. 13. 2013. DOI: 10.1145/2499368.2451145.

[14] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. "Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems". In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys '11. ACM, 2011. DOI: 10.1145/2103799.2103805.

[15] Shuo Chen, Jun Xu, and Emre Can Sezer. "Non-Control-Data Attacks Are Realistic Threats". In: *Proceedings of the 14th USENIX Security Symposium*. 2005. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats.

[16] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey S. Dwoskin, and Dan R. K. Ports. "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*. 2008. DOI: 10.1145/1346281.1346284.

[17] James A. Clause, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. 2007. DOI: 10.1145/1273463.1273490.

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

[19] Mark Cox. *Red Hat's Top 11 Most Serious Flaw Types for 2009*. Feb. 2010. URL: https://awe.com/mark/blog/20100216.html.

[20] *CVE-2019-18619*. July 2020. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=2019-18619 (visited on 07/16/2020).

[21] *CVE-2020-11667*. July 2020. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-11667 (visited on 07/16/2020).

[22] CyberLink. *PowerDVD Ultra Requirements*. URL: https://www.cyberlink.com/products/powerdvd-ultra/spec_en_US.html (visited on 11/14/2019).

[23] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". In: *Proceedings of the 23rd USENIX Security Symposium, USENIX Security*. 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi.

[24] Dell. *DSA-2020-138: Dell Client Platform Security Update for Goodix Fingerprint Sensor Driver Vulnerability*. July 2020. URL: https://www.dell.com/support/article/de-de/SLN321807 (visited on 07/16/2020).

[25] Ran Duan, Long Li, Shi Jia, Yu Ding, Yulong Zhang, Yueqiang Cheng, Lenx Wei, and Tanghui Chen. *Apache Teaclave Rust-SGX SDK - Samplecode "tls/tlsclient"*. URL: https://github.com/apache/incubator-teaclave-sgx-sdk/tree/master/samplecode/tls/tlsclient (visited on 02/28/2020).

[26] Tyler Durden. "Bypassing PaX ASLR protection". In: *Phrack Magazine* 59.9 (2002). URL: http://phrack.org/issues/59/9.html.

[27] *Efficient TLS termination inside Intel SGX enclaves for existing applications: lsds/TaLoS*. Aug. 7, 2019. URL: https://github.com/lsds/TaLoS (visited on 08/27/2019).

[28] Przemyslaw Frasunek. *Full Disclosure Mailing List Archives: FreeBSD 7.0 - 7.2 pseudofs null pointer dereference*. Sept. 2010. URL: https://seclists.org/fulldisclosure/2010/Sep/107 (visited on 11/13/2019).

[29] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. "K-Miner: Uncovering Memory Corruption in Linux". In: *Proceedings 2018 Network and Distributed System Security Symposium, NDSS*. 2018. DOI: 10.14722/ndss.2018.23326.

[30] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. "Out of Control: Overcoming Control-Flow Integrity". In: *2014 IEEE Symposium on Security and Privacy, S&P*. 2014. DOI: 10.1109/SP.2014.43.

[31] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, Michael Grace, Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, Hayawardh Vijayakumar, et al. "PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation". In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020) (To Appear)*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/harrison.

[32] Sean Heelan, Tom Melham, and Daniel Kroening. "Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2019. DOI: 10.1145/3319535.3354224.

[33] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. "Using innovative instructions to create trustworthy software solutions". In: *The Second Workshop on Hardware and Architectural Support for Security and Privacy, HASP*. 2013. DOI: 10.1145/2487726.2488370.

[34] HP. *HPSBHF03675 rev. 1 - Synaptics Fingerprint Drivers that use SGX*. July 2020. URL: https://support.hp.com/hk-en/document/c06696568 (visited on 07/16/2020).

[35] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. "Automatic Generation of Data-Oriented Exploits". In: *24th USENIX Security Symposium, USENIX Security*. 2015. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu.

[36] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. "Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software". In: *Computer Security - 20th European Symposium on Research in Computer Security, Proceedings, Part II, ESORICS*. 2015. DOI: 10.1007/978-3-319-24177-7_16.

[37] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks". In: *IEEE Symposium on Security and Privacy, S&P*. 2016. DOI: 10.1109/SP.2016.62.

[38] Intel. *Demo Programs for the GNU* Multiple Precision Arithmetic Library* for Intel® Software Guard Extensions*. URL: https://github.com/intel/sgx-gmp-demo/ (visited on 10/10/2019).

[39] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes 3 (3A, 3B, and 3C): System Programming Guide*. 2019. URL: https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf.

[40] Intel. *Intel® Software Guard Extensions SDK for Linux**. URL: https://01.org/intel-software-guard-extensions (visited on 08/20/2019).

[41] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*. Order Number 332831-065US. Intel. Dec. 2017.

[42] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. "Block Oriented Programming: Automating Data-Only Attacks". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2018. DOI: 10.1145/3243734.3243739.

[43] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. "Compiler-Generated Software Diversity". In: *Moving Target Defense*. Vol. 54. Advances in Information Security. 2011. DOI: 10.1007/978-1-4614-0977-9_4.

[44] Simon Johnson. *Intel® SGX and Side-Channels*. Feb. 2018. URL: https://software.intel.com/en-us/articles/intel-sgx-and-side-channels (visited on 10/10/2019).

[45] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. "kGuard: Lightweight Kernel Protection against Return-to-User Attacks". In: *Proceedings of the 21th USENIX Security Symposium*. 2012. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kemerlis.

[46] James C King. "Symbolic execution and program testing". In: *Commun. ACM* 19.7 (July 1976). ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[47] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. "SoK: Automated Software Diversity". In: *Proceedings of the 35th IEEE Symposium on Security and Privacy*. 2014. DOI: 10.1109/SP.2014.25.

[48] Jae-Hyuk Lee, Jin Soo 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*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

[49] Lenovo. *Lenovo Security Advisory: LEN-31372*. July 2020. URL: https://support.lenovo.com/de/en/product_security/len-31372 (visited on 07/16/2020).

[50] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. "Implementing an untrusted operating system on trusted hardware". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP*. 2003. DOI: 10.1145/945445.945463.

[51] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. 2005. DOI: 10.1145/1065010.1065034.

[52] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments". In: *24th Annual Network and Distributed System Security Symposium, NDSS*. 2017. DOI: 10.14722/ndss.2017.23227.

[53] Moxie Marlinspike. *Technology preview: Private contact discovery for Signal*. Sept. 26, 2017. URL: https://signal.org/blog/private-contact-discovery/ (visited on 10/10/2019).

[54] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. "Flicker: An execution infrastructure for TCB minimization". In: *ACM SIGOPS Operating Systems Review*. Vol. 42. 4. ACM. 2008. DOI: 10.1145/1357010.1352625.

[55] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative instructions and software model for isolated execution". In: *The Second Workshop on Hardware and Architectural Support for Security and Privacy, HASP*. 2013. DOI: 10.1145/2487726.2488368.

[56] Microsoft. *Data Execution Prevention (DEP)*. 2006. URL: http://support.microsoft.com/kb/875352/EN-US/.

[57] PaX Team. *PaX: PAGEEXEC Design*. URL: https://pax.grsecurity.net/docs/pageexec.txt (visited on 08/23/2019).

[58] Jannik Pewny, Philipp Koppe, and Thorsten Holz. "STEROIDS for DOPed Applications: A Compiler for Automated Data-Oriented Programming". In: *IEEE European Symposium on Security and Privacy, EuroS&P*. 2019. DOI: 10.1109/EuroSP.2019.00018.

[59] Dan R. K. Ports and Tal Garfinkel. "Towards Application Security on Untrusted Operating Systems". In: *3rd USENIX Workshop on Hot Topics in Security, HotSec*. 2008. URL: http://www.usenix.org/events/hotsec08/tech/full%5C_papers/ports/ports.pdf.

[60] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. "VC3: Trustworthy Data Analytics in the Cloud Using SGX". In: *2015 IEEE Symposium on Security and Privacy, S&P*. 2015. DOI: 10.1109/SP.2015.10.

[61] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs". In: *24th Annual Network and Distributed System Security Symposium, NDSS*. 2017. DOI: 10.14722/ndss.2017.23037.

[62] 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. DOI: 10.1145/1315245.1315313.

[63] Yan Shoshitaishvili et al. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy, S&P*. 2016. DOI: 10.1109/SP.2016.17.

[64] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization". In: *Proceedings of the 34th IEEE Symposium on Security and Privacy, S&P*. 2013. DOI: 10.1109/SP.2013.45.

[65] Synaptics. *Synaptics Security Advisory: Synaptics Fingerprint Drivers that use SGX*. July 2020. URL: https://www.synaptics.com/sites/default/files/fingerprint-driver-SGX-security-brief-2020-07-14.pdf (visited on 07/16/2020).

[66] 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. DOI: 10.1109/SP.2013.13.

[67] 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. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[68] 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*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[69] 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. DOI: 10.1145/3319535.3363206.

[70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2Nd Workshop on System Software for Trusted Execution, SysTEX*. 2017. DOI: 10.1145/3152701.3152706.

[71] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: *26th USENIX Security Symposium, USENIX Security*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[72] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. "The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2017. DOI: 10.1145/3133956.3134026.

[73] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. "Towards Memory Safe Enclave Programming with Rust-SGX". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2019. DOI: 10.1145/3319535.3354241.

[74] *wolfSSL Linux Enclave Example*. URL: https://github.com/wolfSSL/wolfssl-examples/tree/master/SGX_Linux (visited on 10/10/2019).

[75] *wolfSSL: a small, fast, portable implementation of TLS/SSL for embedded devices to the cloud*. Oct. 10, 2019. URL: https://github.com/wolfSSL/wolfssl (visited on 08/27/2019).

[76] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy, S&P*. 2015. DOI: 10.1109/SP.2015.45.

[77] Michal Zalewski. *American Fuzzing Lop (AFL)*. 2019. URL: http://lcamtuf.coredump.cx/afl/ (visited on 11/13/2019).