Memory Corruption at the Border of Trusted Execution

Tobias Cloosters and **Oussama Draissi** | University of Duisburg-Essen
Johannes Willbold | Ruhr University Bochum
Thorsten Holz | CISPA Helmholtz Center for Information Security
Lucas Davi | University of Duisburg-Essen

Trusted execution environments provide strong security guarantees, like isolation and confidentiality, but are not immune from memory-safety violations. Our investigation of public trusted execution environment code based on symbolic execution and fuzzing reveals subtle memory safety issues.

Trusted execution environments (TEEs) strongly isolate sensitive code and data from an untrusted computing environment, e.g., the operating system (OS), hypervisor, and other applications. TEEs are designed to allow developers to protect small parts of their application within secure containers, often called *enclaves*, to securely handle sensitive data, such as cryptographic keys. An enclave is a strongly isolated execution environment that can be dynamically created, while the main application, known as a *host*, is running. These enclaves run inside a host application and should even remain secure after the host is compromised. In general, enclaves are as susceptible to memory corruption attacks as any other system software. In fact, many enclaves that we collected and analyzed are developed in the memory-unsafe languages C and C++. However, given that the main purpose of enclaves is the protection of security-critical code and data, as well as their small code size, we would expect that the code has been thoroughly tested and validated. In addition, memory-safe languages, such as Rust, are supported to program enclaves.

In this article we provide a deep dive into analyzing public enclaves to validate whether they are sufficiently protected against memory corruption attacks. In particular, we examine the host-to-enclave boundary as this interface is used to send untrusted (i.e., potentially malicious) input to a trusted code zone. To perform this analysis, we reverse-engineer public enclave code and develop automated analysis techniques based on symbolic execution and fuzzing to assess the security of enclaves. Our findings demonstrate that an erroneous implementation of the application programming interface (API) at the host-to-enclave boundary is often the root cause for memory corruption vulnerabilities in enclave code. Our investigation focuses on the popular TEE implementation of Intel, called *Intel SGX* (*software guard extensions*). For instance, Microsoft Azure offers SGX-based enclaves to their customers and features sophisticated memory protection techniques, e.g., enclave code and data are encrypted and integrity-protected. We also discuss the extent to

Digital Object Identifier 10.1109/MSEC.2024.3381439
Date of publication 10 April 2024; date of current version: 17 July 2024.

which our results apply to other TEE technologies on ARM, RISC-V, or AMD.

SGX Threat Model

SGX enclaves are loaded into the virtual address space of the host process. However, only the enclave can access its own private and secure memory. The enclave memory is stored in an encrypted and integrity-protected memory region in RAM to provide strong isolation and to safeguard the enclave against hardware-level attacks.

An enclave can directly use the host data for its computation, as it is assigned the same read and write permissions to the virtual address space. However, enclave access to the host's virtual address space should be used with care, as there is a high change for race conditions or time-of-check/time-of-use bugs. Hence, this is usually only done when copying input data to private memory or for returning output data upon completion of an enclave call. In contrast, code execution permissions are more restrictive, i.e., using branch instructions (x86: jmp/call) across the host-to-enclave boundary is prohibited. Instead, dedicated SGX instructions have to be used.

These instructions limit enclave execution as follows: The host process starts executing enclave code using the special EENTER instruction, which enables enclave mode and jumps to a selected entry point. The entry points of enclaves are defined in the so-called *thread control structures* (*TCSs*), which are locked while in use by a thread. This makes the number of TCSs also the maximum number of threads that can enter an enclave concurrently. Defining only one TCS is an effective means to mitigate time-of-check/time-of-use bugs. Further, the enclave must explicitly leave enclave mode by using the EEXIT instruction before the thread can execute any host application code.

Pointer Handling

Unsafe pointer operations are the root cause of countless vulnerabilities. However, pointers are also crucial to implement memory access. The threat model of TEEs defines a powerful attacker that controls most of the system; thus, pointers have to be handled with extreme care.

On the assembly level, chunks of data are always handled using pointers. This is also true when passing chunks of data across the host-to-enclave boundary. Since the whole host memory is shared with the enclave, these pointers are per se indistinguishable from other code pointers. As a result, SGX enclaves may unintentionally operate on untrusted data when handling pointers. Therefore, SGX enclaves should ideally check if pointers reference untrusted (outside) or private data (inside) before use. As we will show later, performing this check accurately is not always straightforward.

In particular, it is important that enclaves verify that a given input is not part of their own private memory. For example, an enclave may serve as a transport layer security (TLS) client that stores and protects the encryption secrets in its private memory. When an attacker calls the "send data" method using a pointer to the internal secrets, and if the pointer is not validated, the enclave would send the secrets to a remote party. The same applies in the other direction: Failure to validate an attacker-provided destination pointer being outside may result in a "receive-data" function writing the data intended for the host application into its private memory, thereby corrupting its own integrity.

Furthermore, enclaves should never be designed to purposely take pointers to private memory as input arguments, but we have seen it abused for compatibility purposes, thereby introducing vulnerabilities. When developing C libraries, it is common practice to store state in objects and to use a pointer to this state to identify subsequent library calls to the same state, such as a TLS session. However, using this pattern in enclaves is risky because the session pointer now crosses the trust boundary, making it attacker controlled. This is even exploitable if the enclave validates that the pointer's memory is part of its private memory. Due to possible race conditions, an enclave can never operate directly on outside data; the very first action is always to copy input data into the enclave. The consequence is that—even if the data validation fails—there is now a copy in the private memory. The attacker can abuse this to construct counterfeit objects in the secure memory. We discovered this bug pattern and identified multiple vulnerable TLS enclaves.

Fortunately, Intel provides a regularly updated SGX software development kit (SDK) to ease the memory operations at the boundary. This SDK includes an interface generator that automatically generates code for secure input and output copies. However, it only supports simple and linear data types like byte buffers and strings. For more complex and, in particular, nested structures, it still up to the developer to perform a secure copy manually.

Nested Pointers

Structures that reference substructures are even harder to validate safely. The TaLoS[3] project implemented shadow copies of input data to protect the TLS session objects within the enclave, but also to allow the host application to access some attributes. These shadow objects are created using the input at the beginning of enclave functions and changes are written back afterward. Our analysis revealed several critical vulnerabilities in this code rooted in race conditions and null-pointer dereferences. Thus, the SGX SDKs opted not to support

arbitrary complex data types. Current SDKs show two primary approaches for passing data between enclaves and hosts, supporting only basic data types or completely serializing and deserializing the objects. While the latter approach avoids the need for developers to validate objects themselves, it also adds a lot of code complexity and run-time overhead to the enclave.

Null-Pointer Dereference

The memory address "zero" plays a noteworthy role in the exploitation of SGX enclaves. We have found several exploits abusing this address. Many programming languages, including C/C++, conventionally use zero to indicate uninitialized or missing data. Thus, fresh pointer variables are typically initialized to the null pointer and numerous functions from standard libraries return the null pointer to indicate an error. However, the virtual address 0 is a valid address in user space, but since modern kernels prohibit its usage, null-pointer dereferences are rather harmless. In contrast, in the SGX setting, a null-pointer dereference is critical. Since the kernel is untrusted, an attacker can easily disable the kernel safeguards for address 0. As a consequence, enclaves that erroneously read a null pointer may not fail but rather read untrusted data.

Exploitation of SGX Enclaves

Memory corruption vulnerabilities are a major security concern, as they allow attackers to modify memory in a way that leads to malicious behavior. These vulnerabilities can be exploited to manipulate control-flow information, such as return addresses and function pointers, or to corrupt noncontrol data, such as decision-making variables. In both cases, the attacker can influence the program's execution flow and execute malicious code.

Over the past years, we have witnessed an ongoing battle between defenses and memory corruption attacks. Data execution prevention effectively thwarts injection of malicious code into data memory, in particular for SGX because *mprotect* is not available. However, it can be bypassed through return-oriented programming (ROP) attacks, which exploit existing code residing in code memory without injecting new code. ROP attacks are more challenging against SGX enclaves because the enclaves can be shipped as encrypted binaries, which hinders the static analysis to find ROP gadgets. This is addressed by Dark-ROP,[4] a technique to dynamically identify ROP gadgets by iteratively executing the enclave and analyzing memory accesses. This method enables the identification of ROP gadgets in real time by dynamically observing the memory behavior during execution. On the other hand, Dark-ROP attacks require a consistent memory layout of an enclave. Hence, code randomization (address space layout randomization or ASLR) schemes like SGX-Shield can be used to effectively mitigate Dark-ROP attacks.

Nevertheless, current SGX randomization schemes fall short in providing comprehensive protection against all types of ROP attacks. Biondo et al.[5] show that the Intel SGX SDK contains several powerful ROP gadgets at locations that cannot be randomized as part of the fixed-entry points of SGX enclaves. These gadgets are invoked when resuming the context of an SGX enclave (OCALL-return). Therefore, an attacker can still hijack a vulnerable enclave by launching a memory-corruption attack and providing counterfeit context information. Even if we assume the existence of a perfect control-flow integrity (CFI) scheme, noncontrol data attacks still pose a threat. These attacks manipulate data without breaking the benign control flow and, therefore, cannot be prevented by CFI.

The current state of exploitation techniques indicates that all vulnerabilities in enclaves must be considered critical. If an enclave contains a memory-corruption vulnerability, defenses are unlikely to prevent compromise.

In summary, memory corruption attacks against SGX typically exploit the host-to-enclave boundary, as this serves as an entry point to trigger and halt enclave execution.

Vulnerability Analysis

In the following, we summarize our approaches to vulnerability detection.

Symbolic Execution

Symbolic execution was first proposed in the 1970s as a generalization of testing and has become one of the standard tools for high-coverage testing and vulnerability analysis. However, the modeling of side effects caused by the OS is highly challenging, e.g., symbolic execution must typically simulate and support all OS system calls and manage a simulated file system. 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 isolated from the rest of the system. SGX enclaves are prohibited to perform any system calls and any interaction with the OS is handled through an OCALL to the untrusted host application.

The high-level architecture of TeeRex[1] is shown in Figure 1. The main goal of our method is to find vulnerable states during the symbolic exploration. In particular, TeeRex uncovers unsafe usage of pointers in the input, and null pointers, which in the consequence enables an attacker to control the enclave's private memory and its execution. Further, it aims to

collect metadata 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 the protected enclave memory. Note that TeeRex does not detect vulnerabilities caused by multithreading.

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.

Fuzzing

Fuzzing (short for "fuzz testing") is a dynamic program-analysis technique where a fuzzer feeds seemingly random input into a program. In our SGXFuzz[2] approach, we use feedback-driven fuzzing and present a novel method for fuzzing SGX enclaves that efficiently synthesizes nested input structures. Our approach consists of an enclave transplantation technique and a data structure synthesis method. A high-level overview of our approach is shown in Figure 2.

Enclave Transplantation

While multiple approaches to fuzzing SGX enclaves were explored, the main problem was to obtain feedback, usually in the form of code coverage, indicating which code branches were covered during execution. To address this, we extract the memory, including the code part of an enclave, and transplant it into a user space application. We extract the enclave memory by introducing changes to the SGX drivers for Linux and Windows to automatically dump the enclave's code. The specially crafted user space application mimics the SGX environment and instructions, most importantly, the context switches (i.e., EENTER and EEXIT). A notable exception are the instructions for attestation, for which we do not have access to the processor's internal key. We intercept SIGILL signals to detect SGX' ENCLU instruction without the need for SGX-capable hardware. This allows us to mimic an SGX environment without emulation.

Data Structure Synthesis

With this user space application, we can execute enclaves natively without any form of emulation and the inherent constraints to introspection imposed by regular SGX setups. The ability to natively execute the enclave as user space application also enables efficient fuzzing using readily available fuzzing tools. While previous SGX fuzzing approaches were either limited in compatibility (i.e., requiring specific source code changes) or feedback (i.e., not having efficient code coverage feedback), our approach solves both problems and is binary-only compatible. Apart from requiring an efficient execution setup, enclaves also pose the challenge

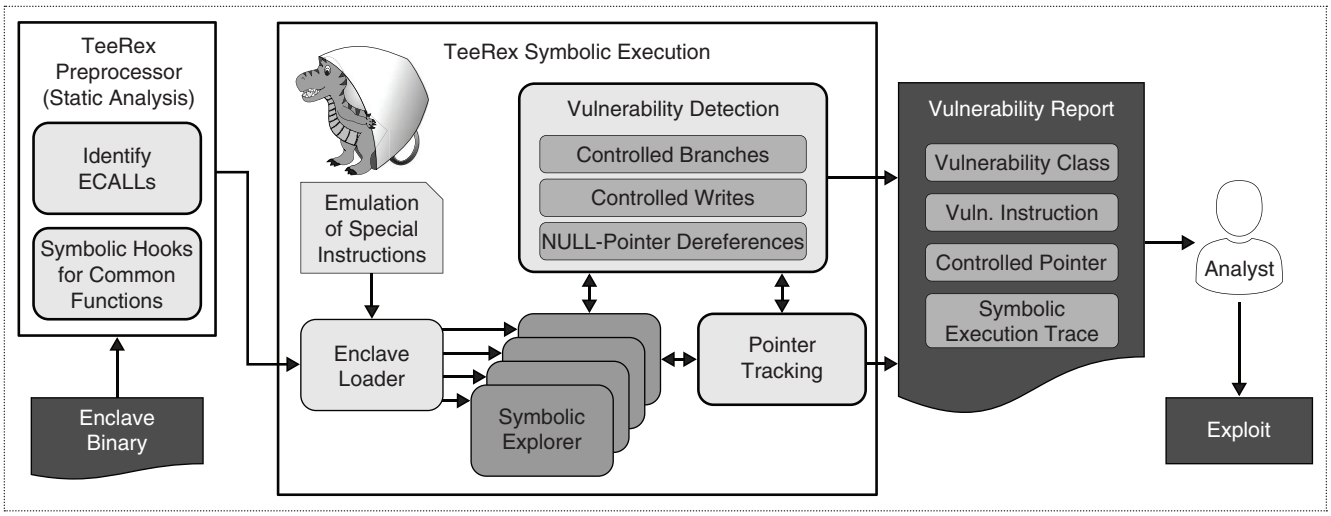

Figure 1. The architecture of TeeRex is split into several major components. The preprocessor identifies instructions and functions that cannot be executed symbolically and locates the ECALL table. The enclave loader sets up the environment and creates the argument structure for the ECALL with unconstrained symbolic values. The vulnerability detection analyzes the symbolic states during symbolic exploration for vulnerabilities, and specifically analyzes instructions that access memory and jumps. TeeRex also implements pointer tracking to determine whether data are loaded from enclave or host memory, or loaded via an ECALL parameter.

of feeding the correct input format. Usually, when fuzzing a program, the target receives data in form of a single buffer (e.g., via standard input), making it trivial for a fuzzer to generate this linear buffer. However, since functions expose regular C functions as their APIs, they might receive any number and type of parameters. To know these signatures in advance, header files or static analysis are required to reverse-engineer them. Instead, we implement an analysis-free binary-only data structure synthesis approach, which incrementally "learns" the layout of the input structure and uncovers possible size fields of variable-sized arrays. Thus, any access beyond our buffer causes a memory fault and prompts us to adjust the layout.

Finally, to detect enclave-specific access violations, i.e., the enclave working on untrusted memory, we probe different types of pointer in the previously determined input layout. More specifically, we test whether pointers inside, outside, or on the enclave's memory boundary lead to distinctive code coverage.

In total, we found 79 vulnerabilities, of which three have been assigned common vulnerabilities and exposure identifiers and a bug bounty of US$ 13k was issued. Notably, we did not encounter any false positives in our evaluation of SGXFuzz. We manually verified that each report is caused by an actual bug, only some bugs occurred duplicated, resulting in more reports than verified bugs.

Examining current and ongoing research, the generation of structures for SGX fuzzing remains a persistent topic. Khan et al.[6] use a combination of static and symbolic program analysis of the enclave and its host application to infer the enclave interface.

Symbolic Execution Versus Fuzzing

An overview of all vulnerabilities found and the root causes identified is provided in Table 1. In the following, we discuss the strengths and weakness of the two analysis techniques we used. First, there are the fundamental advantages of fuzzing and symbolic execution, respectively. Fuzzing provides complete and reproducible test cases with a low false-positive rate. However, every test case starts at the entry point and tests a full execution path. Further, fuzzers have to test identical code paths multiple times until they find a new path because the exact relation between input bytes and branches taken is unknown. At some point, this will reach roadblocks that the fuzzer cannot solve.

Symbolic execution, on the other hand, is a comprehensive approach to explore all possible program states in a single analysis. However, a complete analysis of non-negligible code sizes is infeasible due to the amount of possible states. Hence, practical symbolic execution approaches focus on specific parts and approximate or

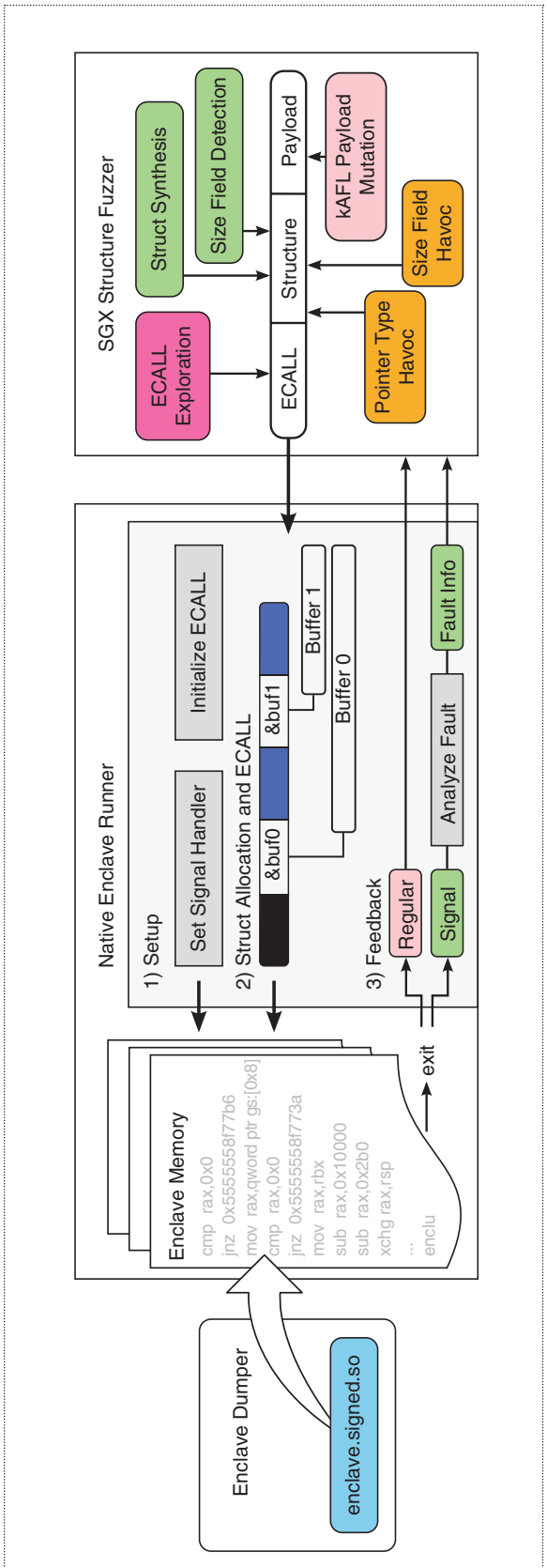

Figure 2. High-level overview of SGXFuzz architecture.[2] We first extract the enclave's memory from the enclave's distribution format. The enclave runner (or harness) is capable of executing any ECALL outside SGX using the dumped enclave, a serialized structure description, and the payload bytes. Our SGX structure fuzzer uses the signals emitted during the enclave's execution to generate and adapt the layouts of the structures.

Table 1. Vulnerabilities found by our analysis using symbolic execution and fuzzing, respectively.

Enclaves	Unchecked Address in Input	Race Condition	Uninitialized/ Null Pointers	Out-of-Bounds Access	Inside Pointers	Overlapping Pointers
Symbolic execution						
Intel GMP example	–	–	–	•	–	•
Rust SGX SDK's *tlsclient*	–	–	–	•	•	–
TaLoS TLS client	•	•	–	•	–	•
WolfSSL example enclave	–	–	–	•	–	–
Synaptics SynaTEE driver	•	–	–	•	–	–
Goodix fingerprint driver	•	–	–	•	–	–
Fuzzing						
Goodix fingerprint driver	•	–	•	–	–	–
sgxwallet	•	–	•	–	–	–
Gingytech fingerprint driver	–	–	–	–	–	•
STANlite	•	–	–	–	–	•
ELAN fingerprint biometric SSL	•	–	–	–	–	–
Town Crier	•	–	–	–	–	–
Synaptics fingerprint driver enclave	–	–	•	–	–	–
OMEC Project's C3P0	•	–	–	–	–	–
lockbox	•	–	–	–	–	–
SGX Darknet	–	–	•	–	–	–
Plinius	–	–	–	–	–	•

exclude others. Moreover, there are conceptional no-ops in programs that introduce high complexity for a symbolic solver. For example, SGX enclaves create secure copies of input data using *malloc* and *memcpy*. While this hardly affects possible memory corruptions, these memory management functions consist of many read/write instructions that impose a burden for the symbolic memory. As a countermeasure, symbolic execution engines try to replace these functions with mocks that are optimized for symbolic engines, but this is not trivial for stripped closed-source binaries, whereas the overhead of these functions is negligible for a fuzzer. As a result, it is difficult for TeeRex to analyze ECALLs with complex input structures in stripped closed-source binaries.

Race Conditions, OCALLs, and Call Order

In addition to the blackbox approaches of TeeRex and SGXFuzz, source-code based analysis can reveal more complex vulnerabilities. COIN attacks[7] analyze vulnerabilities that arise from the interaction of multiple ECALLs. First, if enclaves opt to enable multithreading, race conditions become a serious threat and enclaves have to implement locking mechanisms for their private memory access. SGXRacer[8] is also a blackbox approach to specifically detect race conditions in SGX enclaves. However, it unfortunately reports many false positives. Second, as also shown by the presented blackbox tools, the call order is significant for SGX, e.g., skipping the initialization ECALL may lead to null-pointer dereferences. COIN is able to reveal more subtle cases due to source-code analysis.

Third, OCALLs extend the enclave API and introduce further data input to ECALLs. That is, the attacker-controlled data for a single ECALL now consists of the original input data plus all return data every time the enclave function performs an OCALL. Although TeeRex and SGXFuzz support the concept, it is not yet implemented in the current automated analysis process. Attacks using manipulation of return values were introduced for kernel system calls, i.e., Iago[9] attacks. Cui et al.[10] show that Iago attacks are also a threat to SGX enclaves, in particular, when legacy code is ported to enclaves using compatibility wrappers.

Finally, OCALLs may also introduce reentrancy vulnerabilities, depending on the enclave configuration. This scenario considers one additional level of call, i.e., an ECALL leaves the enclave for an OCALL that again calls into the enclave before returning. In this case, the first (outer) ECALL is not finished when the second ECALL is executed, which may lead to state inconsistencies. The attack of this nature is explained in COIN,[7] but it has not been assessed.

Toward Memory-Safe SGX Enclaves

The vulnerabilities we discovered showed similar patterns of development practices that easily introduce

bugs in SGX enclaves. These are usually code patterns from C/C++ libraries, where either the pattern or even legacy code bases were taken into the development of enclaves. After reporting our findings, developers took our advice into consideration to change and secure the interaction between the host and the enclave.

The first vulnerability pattern we found is the (private) state of SGX enclaves. C libraries commonly let the main application allocate and delete the memory for state objects. Further, the libraries trust the main application to only use the correct type of state objects that are valid for calling a library function. In other words, the library code typically assumes that the state objects are valid and trusted. In SGX, state management has to happen within the enclave and state objects have to be validated for specific enclave functions. While developers tend to take validation of input into account, it is often missed for internal state objects leading to null-pointer dereferences. We encountered multiple enclaves that use one function for initializing the internal state, while other functions operate on this state. Since the state is purely internal, other functions trust this state. However, in SGX, an attacker can also call these functions without calling the initialization first. Hence, the enclave functions operate on the attacker-controlled address zero. Most prominently, we found this type of vulnerability in all fingerprint driver enclaves we analyzed. Since these enclaves are closed source, we cannot analyze code patterns. However, binary analysis reveals that Synaptics added various checks on input and state objects after we reported our findings.

While missing state initialization is the most obvious kind of violation of the state invariants, there also may be other constraints. For example, a cryptographic library may require session initialization and key exchange before sending data. An attacker that initializes a session but skips key exchange may send data using an uninitialized key, thus breaking the encryption. For one fingerprint driver enclave, we constructed an exploit that reads arbitrary private enclave data because the enclave can be tricked to encrypt its memory using an uninitialized key.

The second insecure coding pattern addresses the state references. We noticed that TLS enclaves of the Rust SDK and WolfSSL return pointers to the private memory to track different TLS sessions. As discussed above, this is exploitable because it is impossible for an enclave to check whether the referenced session is genuine. An attacker can easily inject arbitrary data into an enclave as part of a legitimate call because enclaves always have to copy the input data before validation. Therefore, the only option is to verify not the session but whether the pointer itself is a known valid session. This requires the enclave to keep a list of valid session pointers and effectively make the pointer an arbitrary resource identifier. The developers of the TLS enclaves of the Rust SDK and WolfSSL followed our advice and replaced the session pointer with an ID serving as an index in a list of internal session objects.

The third pattern is related to nested data structures. The Intel GMP example code demonstrates GMP math usage inside enclaves by adding two numbers. To do so securely, the code first creates copies of the input numbers in the private memory. However, as it turns out, these copies are flat copies and still reference outside memory. We demonstrated that this vulnerability allows arbitrary writes in the enclave. This bug was probably missed by the enclave developers because the internal structure of GMP numbers is not visible to the users of GMP. Thus, they were not aware of the nested references while using GMP for the enclave. After we reported our findings, the example was changed to only pass serialized data across the enclave boundary. That is, the numbers are now represented as text strings instead of a binary representation.

In general, data serialization is the most promising way to mitigate pointer-based vulnerabilities at the host-to-enclave boundary. Intel's enclave definition language can protect strings and other flat data structures (i.e., without nested pointers). Google's enclave SDK goes one step further and serializes the data using protobuf definitions, which also supports more complex data structures.

Overlapping Memory in the Rust SGX SDK

The Rust SGX SDK is a framework for developing SGX enclaves in the Rust programming language. This promises memory safety within these enclaves due to the inherent memory safety properties of Rust. The Rust SGX SDK is shipped with a *tlsclient* enclave as an example how to run a TLS server and client inside an SGX enclave. We analyzed the *tlsclient* enclave using TeeRex and discovered a memory-corruption vulnerability that enables arbitrary code execution. This example demonstrates that Rust cannot protect against memory-corruption at the host-to-enclave boundary since it is unaware of this border.

The API of the enclave consists of functions to create TLS sessions, and then utilize the session to send and receive data securely. Since this used pointers to private memory for the session objects, this enclave is vulnerable to an object-injection attack, where counterfeit objects are written to enclave memory as part of legitimate data in another enclave function. However, this vulnerability can even be exploited without object injection due to another misconception. The developers who used the pointer-as-session reference intended to disallow sessions outside the secure memory, thus ensuring that the reference is not in the untrusted memory region of the host application. Here, the semantics are crucial because the Intel SGX SDK provides two

functions to check whether a memory area is either strictly outside or inside the enclave memory. Both functions return false for memory that is partially inside and partially outside the secure memory. Since the developers chose to disallow outside memory, they still allowed for two types of memory: inside and partially inside (see Figure 3 for an illustration).

The enclave's `tls_client_write` function receives a session pointer as parameter, which references a structure representing a secure TLS session. Since Rust allows polymorphism, there is a pointer to the virtual method table (vtable) embedded within the object structure. The vtable is a crucial component that stores addresses of all virtual methods associated with a class, which enables the dynamic function calls of polymorphic data types. Hence, if an attacker gains the ability to modify the vtable pointer, this allows the attacker to redirect the enclave's execution flow to arbitrary code within the enclave.

The exploitation of this vulnerability is achieved by constructing a counterfeit sessions object that overlaps enclave and untrusted host memory. This object is placed just before the first page of enclave memory, so that the last byte of the object overlaps into the enclave memory. This is possible because host and enclave share the address space. The single byte is sufficient to be partially inside, i.e., not strictly outside, thus passing the validation. While the value of this last byte is obviously not attacker-controlled, an attacker still maintains control over most bytes of the object, including the vtable address, which can be exploited to execute arbitrary code within the enclave. The Rust SGX SDK developers promptly addressed the issue by introducing session identifiers instead of pointers for the TLS sessions. This mitigates both exploits, the object injection attacks and the edge cases of overlapping memory.

In summary, this example illustrates that Rust, a memory-safe language, is not a silver bullet for enclave security because it does not incorporate the paradigms of SGX into its model. We conclude that the entire software stack, including the SDK itself, must be rigorously analyzed for memory safety vulnerabilities.

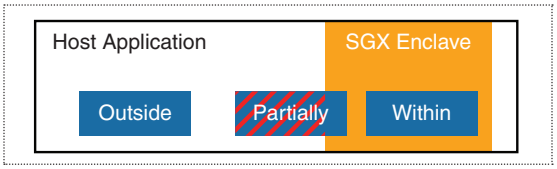

Figure 3. The memory layout of SGX enclaves leads to a third security state. Developers have to be aware that the logical inverse of outside is not only within. Thus, partially overlapping chunks are not (strictly) outside, but still largely attacker-controlled. This bug enabled exploitation of the Rust SDK's TLS client enclave.

Memory Corruption Outside SGX

In addition to Intel SGX, there are several other TEEs, which we present below. Each differ in design choices that change the attack surface, thus mitigating some attacks, while introducing others.

ARM TrustZone

The TEE introduced by ARM, called *ARM TrustZone*, is a security architecture that is extensively used in mobile devices. For instance, Samsung's Knox platform uses ARM TrustZone for secure boot, protected storage, and remote attestation. Furthermore, Google has made TEEs mandatory for any Android device with a fingerprint reader.

TrustZone follows a different trust design than SGX. While the SGX enclave concept provides isolation at the application level, TrustZone partitions the OS into a secure world and an untrusted (rich) system. Both worlds use separate memory spaces to isolate the secure world from the normal world. Each world—secure and nonsecure—has its dedicated OS and applications, categorizing software as *trusted* or *untrusted*. By dividing the processor's memory into distinct regions for each world, it prevents software in one world from directly accessing or altering data in the other world's memory space.

However, since there is only one secure world for all trusted applications (TAs), this design is susceptible to attacks from one (vulnerable) TA to another. Further, TrustZone TAs have privileged access to sensitive data. This leads to a new class of vulnerabilities, namely Boomerang,[11] that exposes this semantic gap between TAs and their untrusted surroundings to manipulated and perform unauthorized operations on behalf of untrusted applications. Untrusted pointers, controlled by user-level applications, can target any memory location in the untrusted environment. However, due to limited visibility into the untrusted environment's security mechanisms, TAs cannot distinguish between safe and unsafe pointers. In a Boomerang attack, an untrusted application manipulates these pointers to read and write any memory location in the untrusted environment, exploiting the TA's full memory access.

Similar to the presented SGX attacks, Boomerang leverages the absence of checks at the host-to-enclave boundary, enabling arbitrary memory reads and writes. Attackers can overwrite kernel memory with a malicious ROP payload, allowing them to steal sensitive data from other applications, bypass security checks, or even gain full control of the untrusted OS.

RISC-V Keystone

Keystone, a RISC-V TEE, uses the RISC-V physical memory protection unit to establish secure enclaves, ensuring isolated processes resistant to manipulation. However, similar to SGX, the software within the

enclaves is still prone to memory corruption vulnerabilities and control-flow hijacks, and the binary interface of enclaves exposes a wide attack surface, which together introduces a high risk of vulnerabilities. In contrast to SGX, Keystone enclaves do not share the address space with the untrusted world, which makes null-pointer dereferences hardly exploitable. Further, Keystone enclaves cannot access the standard system calls, but only application-specific functions implemented in the trusted OS layer. This limits the privileged calls available to enclaves, and shellcode injection attacks based on manipulation of memory permissions are usually not possible. Hence, exploitation primarily relies on sophisticated ROP chains.

We developed RiscyROP,[12] a ROP gadget finding and generation tool kit for RISC-V. ROP on RISC-V is more challenging than on Intel x86 due to the architecture's lack of a stack-based return instruction, dedicated registers for function calls, and the memory alignment of instructions. RiscyROP is a tool that addresses these challenges by using symbolic execution to find and chain gadgets. We use RiscyROP to exploit the attestation demo application running within a Keystone enclave on a HiFive Unleashed RISC-V development board.

AMD Secure Encrypted Virtualization

Secure encrypted virtualization (SEV) is a TEE designed by AMD for virtualized environments. SEV encrypts the memory of each virtual machine (VM) using a secure processor (SP), a coprocessor based on the ARM architecture. The encryption performed within the SP uses individual keys for every VM, which ensures increased security as these keys never leave the SP. This approach provides robust protection against unauthorized access to sensitive data within each protected VM. Notably, the SP exclusively retains all keys to impede external entities from accessing them. Consequently, SEV fortifies systems against memory attacks, and also serves as a deterrent against malicious hypervisors attempting to access sensitive VM data. In a typical system, the hypervisor maintains control over the page tables and can arbitrarily change them. With AMD's SEV, the memory is encrypted using a set of keys known only to the processor, which prevents the hypervisor from reading the memory contents.

While AMD's SEV provides a robust mechanism to protect memory from hypervisor-level attacks, it does not address the inherent vulnerabilities of legacy software that assumes a trustworthy kernel. This assumption is prevalent in traditional Unix-like systems, but leaves existing software vulnerable to Iago[9] attacks that exploit the lack of validation of data passed from the kernel to the TEE. A malicious kernel can exploit this weakness by returning bogus values from system calls, thereby corrupting the memory of a user space process. Iago attacks are applicable to most security solutions, including TrustZone, Keystone, SGX, and AMD SEV. The Iago attack highlights the need for comprehensive system call interface protection in TEEs, encompassing memory pointer integrity to prevent arbitrary code execution and malicious kernel manipulation.

The AMD SEV-ES extension defends against memory leakage and malicious modifications of CPU state by encrypting and integrity-protecting the content of CPU registers when a VM is inactive. However, it does not guard against malicious hypervisors, similar to Iago attacks. The hypervisor can manipulate random number generation within a VM, diminishing entropy in defenses like kernel code randomization and stack canaries. Notably, the SEV-ES extension only protects CPU state and SEV still lacks integrity protection for the encryption of a VM's memory. Radev and Morbitzer[13] demonstrate an attack wherein a malicious hypervisor overwrites a VM's encrypted stack with addresses of suitable ROP gadgets.

TEEs are a promising security technology to strongly isolate sensitive code and data into enclaves. However, the secure implementation of the boundary between host and enclave is extremely critical, as the enclave processes and handles input that originates from an untrusted memory area. To enable a thorough security review of this interface, we have provided a systematic overview of existing research in this area, including an overview of the security assessments of publicly available SGX enclaves. Addressing these findings is crucial to allow secure deployment of TEEs and enclaves in practice. ■

Acknowledgment

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972 and SFB 1119–236615297 within project S2. The work was partially supported by the Ministry of Culture and Science–North Rhine Westphalia research training group *SecHuman* and the European Research Council under the consolidator grant RS^3 (101045669).

References

1. T. Cloosters, M. Rodler, and L. Davi, "TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 841–858.
2. T. Cloosters, J. Willbold, T. Holz, and L. Davi, "SGXFuzz: Efficiently synthesizing nested structures for SGX enclave fuzzing," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 3147–3164.

3. P. Aublin et al., "TaLoS: Secure and transparent TLS termination inside SGX enclaves," Imperial College London, London, U.K., Tech. Rep. 2017/5, 2017.
4. J. Lee et al., "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 523–539.
5. A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: Efficient code-reuse attacks against Intel SGX," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1213–1227.
6. A. Khan, M. Zou, K. Kim, D. Xu, A. Bianchi, and D. J. Tian, "Fuzzing SGX enclaves via host program mutations," in *Proc. 8th Eur. Symp. Secur. Privacy*, 2023, pp. 472–488.
7. M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "COIN attacks: On insecurity of enclave untrusted interfaces in SGX," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2020, pp. 971–985.
8. S. Chen, Z. Lin, and Y. Zhang, "Controlled data races in enclaves: Attacks and detection," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 4069–4086.
9. S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, 2013, pp. 253–264, doi: 10.1145/2490301.2451145.
10. R. Cui, L. Zhao, and D. Lie, "Emilia: Catching Iago in legacy code," in *Proc. Netw. Distrib. Syst. Secur. (NDSS) Symp.*, 2021, pp. 1–22. [Online]. Available: https://www.doi.org/10.14722/ndss.2021.24328
11. A. Machiry et al., "Boomerang: Exploiting the semantic gap in trusted execution environments," in *Proc. 24th ISOC Netw. Distrib. Syst. Secur. (NDSS) Symp.*, 2017, pp. 1–14. [Online]. Available: https://www.doi.org/10.14722/ndss.2017.23227
12. T. Cloosters et al., "RiscyROP: Automated return-oriented programming attacks on RISC-V and ARM64," in *Proc. 25th Symp. Res. Attacks, Intrusions Defenses (RAID)*, 2022, pp. 30–42, doi: 10.1145/3545948.3545997.
13. M. Radev and M. Morbitzer, "Exploiting interfaces of secure encrypted virtual machines," in *Proc. ACM Reversing Offensive-Oriented Trends Symp. (ROOTS)*, 2020, pp. 1–12, doi: 10.1145/3433667.3433668.

Tobias Cloosters is a research assistant at the University of Duisburg-Essen, 45127 Essen, Germany. His research interests include discovery and exploitation of memory-corruption vulnerabilities, including trusted execution on x86 and RISC-V, as well as smart contracts. Cloosters received a M.Sc. from the University of Duisberg-Essen. His work on vulnerability discovery in enclaves on the Intel software guard extensions platform using symbolic execution was awarded with the third prize of the German IT-Security Award 2020. Contact him at tobias.cloosters@uni-due.de.

Oussama Draissi is a research assistant at the University of Duisburg-Essen, 45127 Essen, Germany, specializing in system and software security. His research interests include RISC-V architecture and smart contracts. Draissi received an M.Sc. from the University of Duisburg-Essen. His recent work on detecting bugs in Solana programs was accepted at the Association for Computing Machinery Conference on Communications Security 2023. Contact him at oussama.draissi@uni-due.de.

Johannes Willbold is a research assistant at the chair for systems security at the Ruhr University Bochum, 44801 Bochum, Germany. His research interests include security of space and satellite systems, with an emphasis on firmware security aspects of space systems. Willbold received an M.Sc. from the Ruhr University of Bochum. He recently published a security analysis of low Earth orbit satellites at the 44th IEEE Symposium on Security & Privacy and is the general chair of the SpaceSec workshop. He is a Graduate Student Member of IEEE. Contact him at johannes.willbold@rub.de.

Thorsten Holz is a tenured faculty member at the CISPA Helmholtz Center for Information Security, 66123 Saarbrüken, Germany. His research interests include technical aspects of secure systems, with a specific focus on systems security. Holz received a Ph.D. in computer science from the University of Mannheim. In 2011, he received the Heinz Maier-Leibnitz Prize from the German Research Foundation (DFG), a European Research Council Starting Grant in 2014, and a European Research Council Consolidator Grant in 2022. He is a Member of IEEE. Contact him at holz@cispa.de.

Lucas Davi is a full professor for computer science at University of Duisburg-Essen, 45127 Essen, Germany. His research interests are system and software security, especially focusing on memory exploitation techniques, embedded security, smart contract security, and trusted computing. He received a Ph.D. in computer science from TU Darmstadt. His Ph.D. dissertation on code-reuse attacks and defenses was awarded with the Association for Computing Machinery (ACM) Special Interest Group on Security, Audit, and Control Dissertation Award in 2016. He also received Best and Distinguished Paper awards at IEEE Security & Privacy, Design Automation Conference, and ACM Asia Conference on Communications Security. He currently holds a European Research Council Starting Grant on smart contract security. He is a Member of IEEE. Contact him at lucas.davi@uni-due.de.