A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes

Jo Van Bulck
imec-DistriNet, KU Leuven
jo.vanbulck@cs.kuleuven.be

David Oswald
The University of Birmingham, UK
d.f.oswald@cs.bham.ac.uk

Eduard Marin
The University of Birmingham, UK
e.marin@cs.bham.ac.uk

Abdulla Aldoseri
The University of Birmingham, UK
axa1170@student.bham.ac.uk

Flavio D. Garcia
The University of Birmingham, UK
f.garcia@cs.bham.ac.uk

Frank Piessens
imec-DistriNet, KU Leuven
frank.piessens@cs.kuleuven.be

ABSTRACT

This paper analyzes the vulnerability space arising in Trusted Execution Environments (TEEs) when interfacing a trusted enclave application with untrusted, potentially malicious code. Considerable research and industry effort has gone into developing TEE runtime libraries with the purpose of transparently *shielding* enclave application code from an adversarial environment. However, our analysis reveals that shielding requirements are generally not well-understood in real-world TEE runtime implementations. We expose several sanitization vulnerabilities at the level of the Application Binary Interface (ABI) and the Application Programming Interface (API) that can lead to exploitable memory safety and side-channel vulnerabilities in the compiled enclave. Mitigation of these vulnerabilities is not as simple as ensuring that pointers are outside enclave memory. In fact, we demonstrate that state-of-the-art mitigation techniques such as Intel's `edger8r`, Microsoft's "deep copy marshalling", or even memory-safe languages like Rust fail to fully eliminate this attack surface. Our analysis reveals 35 enclave interface sanitization vulnerabilities in 8 major open-source shielding frameworks for Intel SGX, RISC-V, and Sancus TEEs. We practically exploit these vulnerabilities in several attack scenarios to leak secret keys from the enclave or enable remote code reuse. We have responsibly disclosed our findings, leading to 5 designated CVE records and numerous security patches in the vulnerable open-source projects, including the Intel SGX-SDK, Microsoft Open Enclave, Google Asylo, and the Rust compiler.

KEYWORDS

Trusted execution, TEE, Intel SGX, memory safety, side-channels

ACM Reference Format:
Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, UK*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3363206

CCS '19, November 11–15, 2019, London, UK

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363206

1 INTRODUCTION

Minimization of the Trusted Computing Base (TCB) has always been one of the key principles underlying the field of computer security. With an ongoing stream of vulnerabilities in mainstream operating system and privileged hypervisor software layers, Trusted Execution Environments (TEEs) [28] have been developed as a promising new security paradigm to establish strong hardware-backed security guarantees. TEEs such as Intel SGX [8], ARM TrustZone [34], RISC-V Keystone [21], or Sancus [32] realize isolation and attestation of secure application compartments, called *enclaves*. Essentially, TEEs enforce a dual-world view, where even compromised or malicious system software in the normal world cannot gain access to the memory space of enclaves running in an isolated secure world on the same processor. This property allows for drastic TCB reduction: only the code running in the secure world needs to be trusted for enclaved computation results. Nevertheless, TEEs merely offer a relatively coarse-grained memory isolation primitive at the hardware level, leaving it up to the enclave developer to maintain useful security properties at the software level. This can become particularly complex when dealing with interactions between the untrusted host OS and the secure enclave, e.g., sending or receiving data to or from the enclave. For this reason, recent research and industry efforts have developed several TEE runtime libraries that transparently shield enclave applications by maintaining a secure interface between the normal and secure worlds. Prominent examples of such runtimes include Intel's SGX-SDK [19], Microsoft's Open Enclave SDK [29], Graphene-SGX [43], SGX-LKL [35], Google's Asylo [13], and Fortanix's Rust-EDP [11].

There are some differences in the way each trusted runtime handles input/output data to and from the enclave. At the system level, all TEEs offer some form of `ecall`/`ocall` mechanism to switch from the normal to the secure word (and vice versa). Building on this hardware-level isolation primitive, TEE runtimes aim to ease enclave development by offering a higher level of abstraction to the enclave programmer. Particularly, commonly used production-quality SDKs [19, 29] offer a secure function call abstraction, where untrusted code is allowed to only call explicitly annotated `ecall` entry points within the enclave. Furthermore, at this level of abstraction the enclave application code can call back to the untrusted world by means of specially crafted `ocall` functions. It is the TEE

runtime's responsibility to safeguard the secure function call abstraction by sanitizing low-level ABI state and marshalling input and output buffers when switching to and from enclave mode. However, the SDK-based approach still leaves it up to the developer to manually partition secure application logic and design the enclave interface. As an alternative to such specifically written enclave code, one line of research [1, 2, 42, 43] has developed dedicated enclave library OSs that seamlessly enforce the `ecall`/`ocall` abstraction at the system call level. Ultimately, this approach holds the promise to securely running unmodified executables inside an enclave and fully transparently applying TEE security guarantees.

Over the last years, security analysis of enclaved execution has received considerable attention from a microarchitectural side-channel [24, 26, 30, 45, 46] and more recently also transient execution perspective [5, 20, 44]. However, in the era where our community is focusing on patching enclave software against very advanced Spectre-type attacks, comparably little effort has gone into exploring how resilient commonly used trusted runtimes are against plain *architectural* memory-safety style attacks. Previous research [3, 22] has mainly focused on developing techniques to efficiently exploit traditional memory safety vulnerabilities in an enclave setting, but has not addressed the question how prevalent such vulnerabilities are across TEE runtimes. More importantly, it remains largely unexplored whether there are *new* types of vulnerabilities or attack surfaces that are specific to the unique enclave protection model (e.g., ABI-level misbehavior, or API-level pointer poisoning in the shared address space). Clearly, the enclave interface represents an important attack surface that so far has not received the necessary attention and thus is the focus of this paper.

Our contribution. In this paper, we study the question of how a TEE trusted runtime can securely "bootstrap" from an initial attacker-controlled machine state to a point where execution can be safely handed over to the actual application written by the enclave developer. We start from the observation that TEE runtimes hold the critical responsibility of shielding an enclave application at all times to preserve its intended program semantics in a hostile environment. As part of our analysis, we conclude that the complex shielding requirement for an enclave runtime can be broken down into at least two distinct tiers of responsibilities.

In a first ABI-level tier, we consider that upon enclave entry, the adversary usually controls a significant portion of the low-level machine state (e.g., CPU registers). This requires sanitization, typically implemented through a carefully crafted enclave entry assembly routine to establish a trustworthy ABI state as expected by the compiled application code. Examples of trusted runtime responsibilities at this level include switching to a private call stack, clearing status register flags that may adversely affect program execution, or scrubbing residual machine state before enclave exit.

Secondly, we consider that the enclaved binary itself makes certain API-level assumptions. Here we pay particular attention to pointers and size arguments, because in many TEE designs [8, 21, 32], at least part of the enclave's address space is shared with untrusted adversary-controlled code. Hence, the enclaved binary may assume that untrusted pointer arguments are properly sanitized to point outside of trusted memory, or that `ocall` return values have been scrutinized. Our main contributions are:

- We categorize enclave interface shielding responsibilities into 10 distinct classes, across the ABI and API tiers (cf. Table 1).
- We analyze 8 widely used enclave runtimes, revealing a recurring vulnerability landscape, ranging from subtle side-channel leakage to more grave types of memory safety infringements.
- We practically demonstrate according attacks in various application scenarios by extracting full cryptographic keys, and triggering controlled enclave memory corruptions.
- We show that state-of-the-art automated enclave interface sanitization approaches such as `edger8r`, or even the use of safe languages like Rust, fail to fully prevent our attacks, highlighting the need for more principled mitigation strategies.

Responsible disclosure. All of the security vulnerabilities described in this work have been responsibly disclosed through the proper channels for each affected TEE runtime. In each case, the issues have been verified and acknowledged by the developers. In the case of Intel, this can be tracked via CVE-2018-3626 and CVE-2019-14565, and for Microsoft via CVE-2019-0876, CVE-2019-1369, and CVE-2019-1370. The weakness found in Fortanix-EDP led to a security patch in the Rust compiler. For other open-source projects, our reports have been acknowledged in the respective commits or issues on GitHub. We worked with the maintainers of said projects to ensure mitigation of the problems reported in this paper.

To ensure the reproducibility of our work, and to provide the community with a relevant sample of vulnerable enclave programs for evaluating future attacks and defenses, we published all of our attack code at https://github.com/jovanbulck/0xbadc0de.

2 BACKGROUND AND RELATED WORK

This section reviews enclave operation and TEE design, introduces the trusted runtime libraries we analyzed in this work, and finally summarizes related work on TEE memory corruption attacks.

2.1 Enclave entry and exit

TEE design. The mechanisms to interface with enclaves vary depending on the underlying TEE being used. Figure 1 shows how, from an architectural point of view, we distinguish two types of TEE designs: those that rely on a single-address-space model (e.g., Intel SGX [8] and Sancus [32]) vs. the ones that follow a two-world view (e.g., ARM TrustZone [34] and Keystone [21]). In the former case, enclaves are embedded in the address space of an unprivileged host application. The processor orchestrates enclave entry/exit events, and enforces that enclave memory can never be accessed from outside the enclave. Since the trusted code inside the enclave is allowed to freely access unprotected memory locations outside the enclave, bulk input/output data transfers are supported by simply passing pointers in the shared address space.

In the case of a two-world design, on the other hand, the CPU is logically divided into a "normal world" and a "secure world". A privileged security monitor software layer acts as a bridge between both worlds. The processor enforces that normal world code cannot access secure world memory and resources, and may only call a predefined entry point in the security monitor. Since the security monitor has unrestricted access to memory of both worlds, an explicit "world-shared memory" region can typically be setup to pass data from the untrusted OS into the enclave (and vica versa).

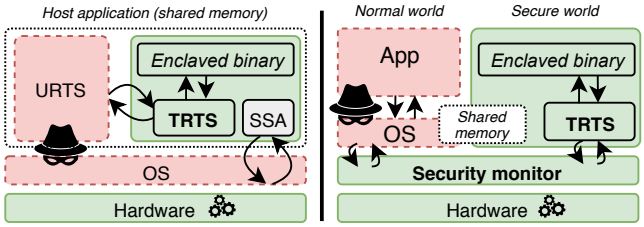

Figure 1: Enclave interactions in a single-address-space TEE design (left) vs. two-world design (right). The software components we study are bold, and the TCB is green (solid lines).

Enclave entry/exit. Given that the runtimes we studied focus mainly on Intel SGX (cf. Section 3.2), we now describe `ecall`/`ocall` and exception handling following SGX terminology [8]. Note that other TEEs feature similar mechanisms, the key difference for a two-world design being that some of the enclave entry/exit functionality may be implemented in the privileged security monitor software layer instead of in the processor.

In order to enter the enclave, the untrusted runtime executes the `eenter` instruction, which switches the processor into enclave mode and transfers execution to a predefined entry point in the enclave's Trusted Runtime System (TRTS). Any meta data information, including the requested `ecall` interface function to be invoked, can be passed as untrusted parameters in CPU registers. TRTS first sanitizes CPU state and untrusted parameters before passing control to the `ecall` function to be executed. Subsequently, TRTS issues an `eexit` instruction to perform a synchronous enclave exit back to the untrusted runtime, again passing any parameters through CPU registers. The process for `ocalls` takes place in reverse order. When the enclave application calls into TRTS to perform an `ocall`, the trusted CPU context is first stored before switching to the untrusted world, and restored on subsequent enclave re-entry.

When encountering interrupts or exceptions during enclaved execution, the processor executes an Asynchronous Enclave eXit (AEX) procedure. AEX first saves CPU state to a secure Save State Area (SSA) memory location inside the enclave, before scrubbing registers and handing control to the untrusted OS. The enclave can subsequently be resumed through the `eresume` instruction. Alternatively, the untrusted runtime may optionally first call a special `ecall` which allows the enclave's TRTS to internally handle the exception by inspecting and/or modifying the saved SSA state.

2.2 TEE shielding runtimes

Intel SGX-SDK. With the release of the open-source SGX-SDK, Intel [19] supports a secure function call abstraction to enable production enclave development in C/C++. Apart from pre-built trusted runtime libraries, a key component of the SDK is the `edger8r` tool, which parses a developer-provided Enclave Description Language (EDL) file in order to automatically generate trusted and untrusted proxy functions to be executed when crossing enclave boundaries.

Microsoft Open Enclave SDK. Microsoft developed the open-source Open Enclave (OE) SDK with the purpose of facilitating TEE-agnostic production enclave development [29]. Currently, OE only supports Intel SGX applications, but in the future TrustZone-based TEEs will also be supported through OP-TEE bindings [29]. The OE runtime includes a custom fork of Intel's `edger8r` tool.

Google Asylo. Google aims to provide a higher-level, platform-agnostic C++ API to develop production enclaves in a Remote Procedure Call (RPC)-like fashion [13]. While the Asylo specification aims to generalize over multiple TEEs, presently only a single SGX back-end is supported, which internally uses Intel's SGX-SDK. From a practical perspective, the Asylo runtime can thus be regarded as an additional abstraction layer on top of the Intel SGX-SDK.

Fortanix Rust-EDP. As an alternative to Intel's and Microsoft's SDKs written in C/C++, Fortanix released a production-quality SGX toolchain to develop enclaves in the safe Rust language [11]. The combination of SGX's isolation guarantees with Rust's type system aims to rule out memory safety attacks against the trusted enclave code. Similar to libOS-based approaches, Rust-EDP hides the enclave interface completely from the programmer and transparently redirects all outside world interactions in the standard library through a compact and scrutinized `ocall` interface.

Graphene-SGX. This open-source library OS approach allows to run unmodified Linux binaries inside SGX enclaves [43]. The trusted Graphene-SGX runtime transparently takes care of all enclave boundary interactions. For this, the libOS offers a limited `ecall` interface to launch the application, and translates all system calls made by the shielded application binary into untrusted `ocalls`. While Graphene was originally developed as a research project, it is currently meeting increasing industry adaption and thrives to become a standard solution in the Intel SGX landscape [36].

SGX-LKL. This open-source research project offers a trusted in-enclave library OS that allows to run unmodified Linux binaries inside SGX enclaves [35]. Similarly to Graphene-SGX, SGX-LKL intercepts all system calls in the shielded application binary, but the libOS layer is internally based on the Linux Kernel Library (LKL).

Keystone. Keystone [21] is an open-source research framework for developing customized TEEs in RISC-V processors. Keystone adopts a "secure world" view similar to ARM TrustZone [34] where a privileged security monitor software layer separates enclaves in their own address spaces, potentially including explicit shared memory regions. Keystone enclaves feature a trusted runtime which intercepts system calls and transparently tunnels all untrusted world interactions through the underlying security monitor.

Sancus. The Sancus research TEE [32] offers lightweight enclave isolation and attestation on an embedded 16-bit TI MSP430 processor featuring a plain single-address-space without virtual memory. A dedicated C compiler automates enclave creation and includes a small trusted runtime library that is transparently invoked on enclave entry/exit. Trusted software may additionally provide code confidentiality [14] or authentic execution [31] guarantees.

2.3 Related work

OS system call interface. During the last decade, significant research efforts have been made to discover and mitigate vulnerabilities in OS kernels, such as missing pointer checks, uninitialized

data leakage, or buffer and integer overflows [6]. By exploiting a single vulnerability in a kernel, unprivileged adversaries may read or write arbitrary memory and gain root access. While these vulnerabilities continue to be relevant in modern kernels, they are generally well understood by the OS security community. However, they have received less attention in the context of TEEs.

Checkoway et al. [4] first demonstrated that an *untrusted* OS can perform so called Iago attacks to compromise legacy applications by supplying maliciously crafted pointers or lengths as the return value of a traditionally trusted system call like `malloc()`. These attacks are closely related to a small subset of the vulnerabilities described in this work, specifically attack vector #9, which exploits that pointers or buffer sizes returned by untrusted `ocalls` may not be properly sanitized (cf. Section 5.5). Our work generalizes Iago attacks from the OS system call interface to `ocalls` in general, and more broadly shows that Iago attacks are but one instance of adversarial OS interactions. We show for instance that legacy applications may also make implicit assumptions on the validity of `argv` and `envp` pointers, which are not the result of system calls.

Memory corruption attacks on ARM TrustZone. ARM TrustZone [34] was one of the first widely deployed TEEs, particularly in mobile devices, and hence received considerable attention from security researchers. The code running in the secure world largely depends on the device manufacturer, with widely used runtimes including Trustonic Kinibi, Qualcomm's QSEE, Google's Trusty, and the open-source project OP-TEE. Over the past years, several vulnerabilities [33, 34] have been discovered in TrustZone runtimes caused by e.g., missing or incorrect pointer range or length checks, or incorrect handling of integer arithmetic. Often, these vulnerabilities rely on the existence of a shared memory region for data exchange between the normal and secure worlds: if an adversary passes a pointer into trusted memory where a pointer to shared memory is expected, memory corruption or disclosure may occur when the pointer is not properly validated by the trusted runtime.

Machiry et al. [27] presented a related class of Boomerang attacks, which leverage the fact that TrustZone's secure world OS has full access to untrusted memory, including the regions used by the untrusted OS. Boomerang exploits that trusted pointer sanitization logic may only validate that pointers lie outside of secure memory, allowing unprivileged code executing in the normal world to read or write memory locations belonging to other applications or the untrusted OS. In a sense, Boomerang vulnerabilities are orthogonal to a subset of the vulnerabilities described in this paper: both target incorrect pointer checks within trusted code, but while Boomerang attacks relate to checks of pointers into *untrusted* memory, we focus on pointers into *trusted* memory.

Memory corruption attacks on Intel SGX. Lee et al. [22] were the first to execute a completely blind memory corruption attack against SGX by augmenting code reuse attack techniques [41] with several side-channel oracles. To successfully mount this attack, adversaries require kernel privileges and a static enclave memory layout. Recently, these techniques were improved by Biondo et al. [3] to allow even non-privileged adversaries to hijack vulnerable enclaves in the presence of fine-grained address space randomization [40]. Their approach is furthermore made application-agnostic by leveraging gadgets found in the trusted runtime library of the official Intel SGX-SDK. In a perpendicular line of research, Schwarz et al. [38] criticized SGX's design choice of providing enclaves with unlimited access to untrusted memory outside the enclave. They demonstrated that malware code executing inside an SGX enclave can mount stealthy code reuse attacks to hijack control flow in the untrusted host application.

Importantly, all previous SGX memory safety research focused on contributing novel exploitation techniques while assuming the prior presence of a vulnerability in the enclave code itself. Hence, those results are *complementary* to the vulnerabilities described in this work. We have indeed demonstrated control flow hijacking for some of the pointer sanitization issues below, and these may further benefit from exploitation techniques developed in prior work.

3 METHODOLOGY AND ADVERSARY MODEL

3.1 Attacker model

We consider systems with hardware support for a TEE and where a trusted runtime supports the secure, shielded execution of an enclaved binary produced by the application developer. With *enclaved binary*, we specifically mean that the binary is the output of a standard compiler, which is not aware of the TEE. It is the responsibility of the shielding runtime to preserve intended program semantics in a hostile environment. We focus exclusively on vulnerabilities in the TEE runtime and assume that there are no application-level memory safety vulnerabilities in the enclaved binary.

We assume the standard TEE attacker model [28], where adversaries have full control over *all* software executing *outside* the hardware-protected memory region. This is a powerful attacker model, allowing the adversary to, for instance, modify page table entries [47, 54], or precisely execute the victim enclave one instruction at a time [45]; yet, this is the attacker that TEEs are designed to defend against. It is important to note that some of the attacks we discuss can also be launched by significantly less privileged attackers, *i.e.*, with just user-level privileges to invoke the enclave.

3.2 Research methodology

Our objective is to pinpoint enclave shielding responsibilities, and to find vulnerabilities where real-world TEE runtimes fail to safeguard implicit interface assumptions made by the enclaved binary.

TEE runtime code review. We base our research on manual code review, and hence limited our study to open-source TEE runtimes. After reviewing the literature and code repositories, we selected 8 popular runtimes to be audited. Our resulting selection allows to compare tendencies in (*i*) production vs. research code bases; (*ii*) SDK vs. libOS-based shielding abstractions; (*iii*) unsafe C/C++ vs. safe Rust programming languages; and (*iv*) underlying TEE design dependencies. Note that we opted not to include `baidu-rust-sgx`, as it is merely a layer on top of Intel SGX-SDK (and hence inherits all vulnerabilities of the latter). After reviewing prior research [33] and relevant code, we found that sanitization in the TrustZone runtime OP-TEE has already been thoroughly vetted and we hence decided not to systematically audit this runtime. For each of the selected TEE runtime implementations, we then reviewed the sanitizations and defensive checks implemented by the trusted runtime between entering the TEE and transferring control

Table 1: Enclave runtime vulnerability assessment (our contribution, highlighted) and comparison to related work on OSs and TEEs. Symbols indicate whether a vulnerability was successfully exploited (★); acknowledged but without proof-of-concept (●); or not found to apply (○). Half-filled symbols (⯪, ◐) indicate that improper sanitization only leads to side-channel leakage.

Vulnerability \ Runtime		SGX-SDK	OpenEnclave	Graphene	SGX-LKL	Rust-EDP	Asylo	Keystone	Sancus	Linux	Prior TEE attack research
Tier1 (ABI)	#1 Entry status flags sanitization	★	★	◐	●	◐	●	○	○	[9]	
	#2 Entry stack pointer restore	○	○	★	●	○	○	○	★	○	
	#3 Exit register leakage	○	○	○	★	○	○	○	○	○	SGX Dark-ROP exploitation [3, 22]
Tier2 (API)	#4 Missing pointer range check	○	★	★	★	○	●	○	★	[6]	TrustZone exploits [33, 34]
	#5 Null-terminated string handling	⯪	★	○	○	○	○	○	○	[6]	
	#6 Integer overflow in range check	○	○	●	○	●	○	●	●	[6]	TrustZone exploits [33, 34]
	#7 Incorrect pointer range check	○	○	●	○	○	●	○	●	○	
	#8 Double fetch untrusted pointer	○	○	●	○	○	○	○	○	[37, 53]	SGX AsyncShock framework [50]
	#9 Ocall return value not checked	○	★	★	★	○	●	★	○	–	Iago attacks (Linux system call interface) [4]
	#10 Uninitialized padding leakage	[23]	★	○	●	○	●	★	★	[7]	SGX-SDK `edger8r struct` leakage [23]

to the enclaved binary, and the symmetrical path when exiting the TEE. We found new vulnerabilities in all studied runtimes. Table 1 summarizes our findings, structured according to the respective vulnerability classes, and relating to similar vulnerabilities in the Linux kernel and prior TEE research. Our systematization revealed 10 distinct attack vectors across 2 subsequent tiers of TEE shielding responsibilities, explored in Sections 4 and 5, respectively.

In our code review, we focus our attention on the assumptions that an enclaved binary makes about two key interfaces, and we consider both integrity and confidentiality concerns. A first level of interface sanitization we inspect is the ABI, which unambiguously specifies function calling conventions regarding the low-level machine state expected by the compiler [10]. We manually locate the trusted runtime entry point, and review how the compact assembly routine establishes a trustworthy ABI state on entry, and similarly scrubs residual CPU state on exit. The second key interface, that we refer to as the API, is the functional interface of the enclaved binary. We review how the TEE runtime validates different kinds of arguments passed in through an `ecall` or as the return value of an `ocall`. We focus in particular on the handling of pointers and strings, where it is the TEE runtime's responsibility to ensure that variable-sized buffers lie entirely outside the enclave before copying them inside and transferring execution to the enclaved binary. For confidentiality, we check again that all memory copied outside the TEE only contains explicit return values, and that no avoidable side-channel leakage is introduced.

TEE design considerations. The communication between enclave and untrusted code for all TEE runtimes considered in this paper relies on some form of "world-shared memory", *i.e.*, a memory region that is accessible to both trusted and untrusted code. Depending on the specific TEE design (cf. Fig. 1), this can be realized by either embedding the enclave in the address space of a surrounding host process, as in Intel SGX [8] or Sancus [32], or by explicitly mapping a dedicated virtual memory region into both worlds as in ARM TrustZone [34] and Keystone [21]. Prior research has mainly explored interface sanitization vulnerabilities in ARM TrustZone TEEs (cf. Section 2.3). Given the prevalence of SGX in contemporary Intel processors, our study focuses largely on SGX-style single-address-space TEE designs as used in 7 out of 8 considered runtimes. However, the example of Keystone, and prior research on ARM TrustZone [33, 34], shows that the attack surface studied here is not necessarily limited to TEEs using the single-address-space approach taken by SGX. As part of our analysis, we found that certain TEE-specific design considerations may sometimes significantly impact exploitability. When applicable, such TEE design considerations are discussed throughout the paper.

4 ESTABLISHING A TRUSTED ABI

Similarly to traditional user/kernel isolation, TEE-enabled processors typically only take care of switching to a fixed entry point and thereafter leave it up to trusted runtime software to securely bootstrap the enclaved execution. In practice, this implies that adversaries may still control a large fraction of the low-level machine state (e.g., CPU registers) on enclave entry. Hence, a trusted assembly entry routine is responsible to establish an ABI-compliant machine state when transferring control to the shielded application, and to save and scrub low-level machine state on enclave exit.

4.1 Sanitizing machine state on entry

After reviewing well-documented ABI-level calling conventions [10] expected by popular C compilers, we concluded that most CPU registers can be left unmodified, apart from the stack pointer explored in the next section. However, a more subtle concern relates to the expected state of certain status register flags on function entry.

> **Attack vector #1 (status flags):** Entry code should sanitize register flags that may adversely impact program execution. ▷ Prevalent in production and research runtimes, but exclusively Intel SGX (x86 CISC).

TEE design. The underlying processor architecture used in the specific TEE design may greatly impact the resulting ABI-level attack surface. That is, in comparison to Intel's notoriously complex x86 CISC architecture [8], simpler RISC-based TEEs such as Sancus [32], Keystone [21], or ARM TrustZone [34] tend to impose less obligations for trusted software to sanitize low-level machine state. For instance, we found that the Sancus runtime should only take care to clear the interrupt flag. Likewise, TrustZone even transparently takes care to save/restore secure world stack pointer registers.

Our analysis further reveals the trade-offs for implementing register and status flag clearing in either hardware or software. For instance, we show that the Intel SGX design leaves this responsibility largely to software, exposing a larger attack surface.

We methodically examined all the software-visible flags in the x86 `flags` register [17] and discovered two potentially dangerous flags that may adversely impact enclaved execution if not properly cleared. First, the Alignment Check (AC) flag may be set before entering the enclave in order to be deterministically notified of every unaligned memory access performed by the trusted enclave software. This novel side-channel attack vector is closely related to well known page fault [54] or segmentation fault [15] controlled-channels, but this time abuses x86 `#AC` alignment-check exceptions. Also, note that `#PF` side-channels ultimately reflect fundamental hardware-level TEE design decisions that cannot be avoided in software, whereas we argue that `#AC` leakage originates from the trusted runtime's failure to clear the associated status register control flag. A second and more dangerous ABI-level attack vector arises from the Direction Flag (DF), which can be set to change the loop behavior of x86 string instructions (e.g., `rep movs`) from auto-increment to auto-decrement. Commonly used x86 ABIs [10] allow for compiler optimizations by mandating that DF shall always be cleared on function call/return. However, in case this subtle ABI requirement is not explicitly enforced in the assembly entry routine, SGX adversaries may change DF to an unexpected "decrement" direction before the `ecall` and thereby hijack the intended direction of all subsequent x86 string instructions executed by the enclave. This opens a severe vulnerability that can be successfully exploited to trigger enclave memory corruption and erroneous computation results.

Intel SGX-SDK. We experimentally confirmed that the trusted runtime in Intel's official SGX-SDK [19] does *not* clear AC or DF on enclave entry. The latter can be tracked via CVE-2019-14565 (Intel SA-00293), leading to enclave TCB recovery.

While unaligned data accesses (e.g., fetching a 16-bit word at an odd byte address) are explicitly supported in the x86 architecture, the processor may optionally be forced to generate an exception for such accesses when software sets the AC bit in the `flags` register. We developed a minimal sample enclave to showcase how `#AC` exceptions may in certain scenarios reveal secret-dependent data accesses at an enhanced byte-level granularity as compared to state-of-the-art SGX side-channel attacks that are restricted to a coarser-grained 64 B cacheline [39] or 4 KiB page-level [47, 54] granularity. Figure 2 illustrates the key idea behind the attack, where a 16-bit word is loaded by specifying a byte-granular index in a small lookup table that has been explicitly aligned to a cacheline boundary (e.g., as might also be performed in a streamed data or string processing enclave application). In the example, secret index 0 returns the data `AB`, whereas secret index 1 returns `BC`. Our exploit deterministically reconstructs the intra-cacheline secret-dependent data access by observing whether or not the enclaved execution generates an `#AC` alignment-check exception. One of the challenges we encountered is to make the enclave progress after returning from the untrusted signal handler. Since the processor automatically restores the previous value of the `flags` register (including the set AC bit) from enclave-private SSA memory when resuming the enclave [8], the unaligned data access will never be allowed to complete. To overcome this challenge, we make use of the adversary's root privileges to load a simple kernel module that clears the processor's Alignment Mask (`CR0.AM`) to temporarily disable alignment checking. Combined with a single-stepping attack primitive like SGX-Step [45], this approach allows to determine noise-free alignment side-channel information for *every* single instruction in the victim enclave.

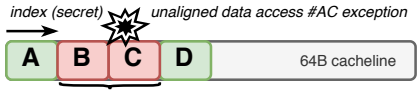

Figure 2: Misaligned, intra-cacheline secret data access.

It should be noted that the oversight of not clearing the AC flag in the trusted runtime merely leaks address-related side-channel information, which falls explicitly outside of SGX's threat model [8]. However, this is distinctly *not* the case for the DF flag, which directly intervenes with the semantics of the enclaved execution. We confirmed that the popular `gcc v5.4` compiler replaces for instance common `strlen()` and `memset()` invocations with inlined x86 string instructions at optimization level `-Os`. We developed a start-to-end attack scenario to show how forcibly inverting the direction of such string operations when entering the enclave through an `ecall` can lead to controlled heap corruption and memory disclosure. Our PoC exploit targets `edger8r` bridge code that is automatically generated to copy input and output buffers to and from the enclave (cf. Section 5.1 and Fig. 3). Particularly, we abuse that `edger8r` code allocates the output buffers on the enclave heap and thereafter uses `memset()` to securely initialize the newly allocated buffer to all-zero. However, setting DF before the `ecall` causes the `memset()` direction to be inverted and any preceding heap memory to be corrupted (*i.e.*, zeroed). Due to the way the SGX-SDK enclave heap is organized, this will ultimately lead to a crash on the next `free()` invocation in the `edger8r` code. Every heap frame is preceded by a size field and a pointer to a meta-data bookkeeping structure. Such pointers are stored in `xor`-ed form with a randomly generated secret constant to harden the code against traditional heap corruption attacks. We confirmed that after erroneously zeroing the preceding heap frames, the resulting pointer will most likely end up as a non-canonical 64-bit address and halt the enclave by means of a general protection fault. However, before finally calling `free()` and detecting the heap corruption, the trusted `edger8r`-generated code still copies the allocated output buffer outside the enclave, potentially leading to secret disclosure (as this buffer has never been properly zeroed). We note that the heap corruption in itself may also be leveraged in application-specific scenarios, e.g., zeroing out a cryptographic key residing in the preceding heap frame.

Microsoft Open Enclave SDK. We experimentally confirmed that OE suffers from the same DF vulnerability described above (tracked via CVE-2019-1370). However, we found that after entering the enclave with the DF flag set, the trusted runtime already crashes early-on in the entry path. The reason for this is that on our machines (`gcc v5.4` using the default Makefile), one of the compiled entry functions uses a `rep` string instruction to initialize a local variable on the call stack. Hence, setting DF leads to memory

```
1   cmp $RETURN_FROM_OCALL, %rdi    ; %RDI = attacker arg
2   je .Lreturn_from_ocall
3   ...
4 .Lreturn_from_ocall
5 ★ mov %gs:SGX_LAST_STACK, %rsp
6   ...
7   ret
```

Listing 1: Low-level `ocall` return path in Graphene-SGX.

corruption by overwriting a piece of the *trusted* call stack with zeroes. We have not attempted to further exploit this behavior.

Other SGX runtimes. When reviewing the assembly entry routines of the other SGX-based shielding systems (cf. Table 1), we found that *none* of them sanitizes AC, whereas interestingly both Rust-EDP and Graphene-SGX clear DF on enclave entry. Note that Google's Asylo framework is built on top of the Intel SGX-SDK and hence inherits all of the vulnerabilities described above.

4.2 Maintaining the call stack abstraction

In order to safeguard enclave confidentiality and integrity, it is essential that enclaves features their own private call stack. When exiting the TEE by means of an `ocall`, the trusted stack pointer should be stored and control flow should continue at a location outside the enclave. After having performed an `ocall`, upon receiving the next `ecall`, the private call stack should be restored so the runtime can "return" into the shielded application.

Attack vector #2 (call stack): Entry code should safeguard the call stack abstraction for `ecalls` and `ocalls`. ▷ Not applicable to TrustZone, well-understood in production SGX-SDKs, but not always in research code.

TEE design. We observed that TEE-specific design decisions may largely impact the attack surface arising from call stack switching. That is, in ARM TrustZone [34] the stack pointer CPU register is duplicated and fully transparently stored/restored on secure world context switches. More versatile TEE designs like Intel SGX [8] or Sancus [32], on the other hand, support multiple mutually distrusting enclaves and leave it up to trusted runtime software to store and restore the stack pointer across enclave boundaries. Another illustration of the trade-offs between hardware and software responsibilities arises in SGX's `eexit` instruction, which was designed to explicitly fault when supplying in-enclave continuation addresses [8]. Alternative TEE designs like Sancus [32], on the other hand, expect such continuation pointer checks to be performed by the trusted software, leaving a larger attack surface.

Graphene-SGX. After scrutinizing Graphene's low-level bootstrapping code, we discovered that `enclave_entry.S` does not properly safeguard the `ocall` return abstraction. Listing 1 shows how the code unconditionally jumps to the stack pointer restore logic after merely receiving an *unchecked* magic value in the `%rdi` register. We experimentally confirmed that this can be abused to illegally "return" into an enclave thread that is not waiting for a previous `ocall` return. An adversary can exploit this weakness to erroneously initialize the trusted in-enclave stack pointer of a newly started thread with the value of the last `ocall`. The memory content at these locations determine the values popped into registers, and ultimately `ret` control flow.

SGX-LKL. We found a highly similar vulnerability in the way SGX-LKL's low-level entry code distinguishes different `ecall` types. Specifically, we noticed that the unchecked parameter in `%rdi` can be poisoned to trick the entry routine into erroneously calling a signal handler for a thread that was never interrupted. This is especially problematic as the signal handler code will then illegally restore the stack pointer register from an uninitialized memory location.

Sancus. We reviewed the assembly code inserted at the entry point of a Sancus enclave, and noticed that the Sancus TEE suffers from similar call stack switching vulnerabilities. Particularly, we experimentally confirmed that it is possible to supply illegal CPU register arguments and trick the enclave into "returning" into a thread that was not waiting for a previous `ocall` return. In such a case, the enclave stack will be falsely restored to the value of the last valid `ocall`, leading to memory-safety violations from incorrect control flow and register values. Sancus's enclave entry assembly routine further expects a CPU register parameter to specify the address where execution is continued after leaving the enclave. The software does not properly validate this parameter. Unlike SGX's `eexit` hardware primitive, which refuses to jump to illegal continuation addresses, Sancus enclaves are exited by means of an ordinary `jmp` instruction. We experimentally confirmed the possibility of code reuse attacks [41] by forcing the vulnerable entry routine to jump to an arbitrary in-enclave continuation address.

4.3 Storing and scrubbing machine state on exit

Prior to exiting the TEE, the trusted runtime's assembly routine should save and clear all CPU registers that are not part of the calling convention, and restore them on subsequent enclave re-entry. This is highly similar to how a traditional operating system needs to context switch between processes, and hence we found this to be a generally well-understood requirement.

Attack vector #3 (register state): Exit code should save and scrub CPU registers. ▷ Generally well-understood across runtimes and architectures.

TEE design. Similar to parameter passing across traditional user/kernel boundaries, widespread TEE designs commonly preserve CPU register contents when context switching between the normal and secure worlds. Prior research [3, 22] on exploiting memory safety vulnerabilities in SGX enclaves has for instance exploited that the `eexit` instruction does *not* clear register values, leaving this as an explicit software responsibility. Further, while scrubbing CPU registers on enclave interrupt is a hardware responsibility in the Intel SGX design [8], we found that the AEX operation in current SGX processors does *not* clear the x86 DF flag (cf. Section 4.1). We experimentally confirmed that this can be exploited as a side-channel to learn the direction of private in-enclave string operations.

SGX-LKL. When reviewing the respective assembly routines, we noticed that SGX-LKL is the only SGX runtime which does not properly scrub registers before invoking `eexit`. The reason for this oversight is that LKL attempts to leverage the `setjmp/longjmp` standard C library functions to easily store and restore the execution state on enclave entry/exit without needing dedicated assembly code. While indeed functionally correct, *i.e.*, the integrity of CPU registers is preserved across enclave calls, the approach cannot guarantee confidentiality. This is because `setjmp()` still behaves

as a normal C function, which—adhering to calling conventions—does *not* clear all CPU state. We therefore advise to use a dedicated assembly routine which overwrites confidential CPU registers before invoking `eexit`. This issue highlights the necessity to explicate and properly separate ABI and API-level shielding concerns in consecutive stages of the trusted runtime (cf. Section 3). We experimentally confirmed this vulnerability by loading an elementary AES-NI application binary inside SGX-LKL, and modifying the *untrusted* runtime to dump x86 `xmm` registers—including the AES state and round keys—after enclave exit.

5 SANITIZING THE ENCLAVE API

Once a trustworthy ABI state has been established, the trusted bootstrapping assembly code can safely transfer control to machine code emitted by a compiler from a program description written in a higher-level language. Remarkably, almost all runtimes [13, 19, 21, 29, 32, 35, 43] we studied are written in C or C++, with the notable exception of Fortanix's EDP platform [11], which is written in the memory-safe Rust language. While the use of safe languages is indeed preferable to rule out an important class of application-level memory-safety vulnerabilities in the trusted runtime implementation, we show that safe languages by themselves cannot guarantee that the enclave interface is safe.

That is, it remains the responsibility of the trusted runtime implementation to marshal and scrutinize untrusted input parameters before passing them on to the shielded application written by the enclave developer. Depending on the specific runtime, developers may communicate trusted API sanitization and marshalling requirements explicitly (e.g., using a domain-specific language like in Intel's `edger8r` or Microsoft's `oeedger8r`), or the enclave interface may be completely hidden from the programmer (e.g., libOS-based approaches).

In this section, we analyze shielding requirements for API sanitization based on the different types of arguments that can be passed across the enclave boundary. We pay particular attention to pointers and (variable-sized) input buffers, given the prevalent weaknesses found in real-world code.

5.1 Validating pointer arguments

Whenever untrusted side and enclave share at least part of their address spaces, an important new attack surface arises: malicious (untrusted) code can pass in a pointer to enclave memory where a pointer to untrusted memory is expected. Therefore, it is the responsibility of the shielding system to be careful in never dereferencing untrusted input pointers that fall outside of the shared memory region and point into the enclave. In case such sanity checks are missing, the trusted enclave software may unintentionally disclose and/or corrupt enclave memory locations. This is an instance of the well-known "confused deputy" [16] security problem: the attacker is architecturally prohibited from accessing secure enclave memory, but tricks a more privileged enclaved program to inadvertently dereference a secure memory location chosen by the attacker.

Attack vector #4 (pointers): Runtimes should sanitize input pointers to lie inside the expected shared memory region. ▷ Generally understood, but critical oversights prevalent across research and production code.

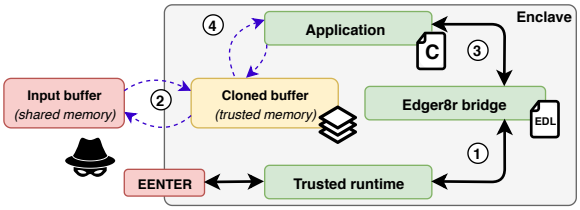

Figure 3: Automatically generated `edger8r` bridge code handles shielding of application input and output buffers.

TEE design. TEEs commonly support some form of shared memory which allows trusted in-enclave code to directly read or write an untrusted memory region outside the enclave (cf. Section 3.2). Input and output data transfers can now easily be achieved by bulk-copying into the shared memory region and passing pointers.

Pointer sanitization is a relatively well-known requirement for enclave applications, and even bears some similarity with traditional user-to-kernel system call validation concerns [6]. However, the kernel system call interface remains largely invisible, fairly stable, and is only modified by a select group of expert developers. SDK-based enclave development frameworks on the other hand expose `ecalls` and `ocalls` much more directly to the application developer by means of a secure function call abstraction.

Intel SGX-SDK. In line with trusted runtime shielding requirements, pointer sanitization should preferably not be left to the application developer's end responsibility. As part of the official SGX-SDK, Intel [19] therefore developed a convenient tool called `edger8r`, which transparently generates trusted proxy bridge code to take care of validating pointer arguments and copying input and output buffers to/from the enclave. The tool automatically generates C code based on `ecall`/`ocall` function prototypes and explicit programmer annotations that specify pointer directions and sizes in a custom, domain-specific Enclave Definition Language (EDL).

Figure 3 gives an overview of the high-level operation of the trusted `edger8r` bridge code. After entering the enclave, the trusted runtime establishes a trusted ABI (cf. Section 4), locates the `ecall` function to be called, and finally ① hands over control to the corresponding `edger8r`-generated bridge code. At this point, all input buffer pointers are validated to fall completely outside the enclave, before being copied ② from untrusted shared memory to a sufficiently-sized shadow buffer allocated on the enclave heap. Finally, the `edger8r` bridge transfers control ③ to the code written by the application developer, which can now safely operate ④ on the cloned buffer in enclave memory. A symmetrical path is followed when returning or performing `ocalls` to the untrusted code outside the enclave.

Microsoft Open Enclave SDK. Microsoft [29] adopted the sanitization strategy from the Intel SGX-SDK by means of their own `oeedger8r` fork. Interestingly, OE uses a "deep copy" marshalling scheme to generalize to TEEs where the enclave cannot directly access host memory and every interaction needs to be mediated in a security kernel with access to an explicit shared memory region (cf. Fig. 1). With deep copy marshalling, instead of passing the enclave pointers to the input buffer, the contents of the buffer are first

```
1 OE_ECALL void ecall_hello(hello_args_t* p_host_args) {
2   oe_result_t __result = OE_FAILURE;
3   if (!p_host_args || !oe_is_outside_enclave(p_host_args,
4                                 sizeof(*p_host_args)))
5     goto done;
6   ...
7   done:
8 ★   if (p_host_args) p_host_args->_result = __result;
9 }
```

Listing 2: Proxy function generated by `oeedger8r` (simplified) with illegal write to arbitrary in-enclave pointer on failure.

copied into the marshalling structure and then cloned into enclave memory. The pointers in the argument structure are then modified such that they point to the corresponding (cloned) memory buffer.

Nevertheless, we discovered several flaws in the way OE handles pointer validation (tracked via CVE-2019-0876). A first subtle issue was found by reviewing the `oeedger8r`-generated code skeleton itself. Listing 2 shows a simplified snippet of the trusted bridge code generated for an elementary `hello()` entry point. The code attempts to properly verify that the untrusted `p_host_args` structure lies outside the enclave, and indeed rejects the `ecall` when detecting a pointer poisoning attempt. However, in the `done` branch at line 8, an error code is still written into the `p_host_args` structure, even if it was found earlier to illegally point inside the enclave. At the time of our review, this could only be exploited when calling the enclave through a legacy `ecall` dispatcher that had unfortunately not been removed from OE's trusted code base (cf. Appendix A.1).

Secondly, we found that enclaves built with OE feature a small number of "built-in" `ecall` entry points for infrastructural functionality directly serviced in the trusted runtime without forwarding to the shielded application. Notably, OE developers decided *not* to route these entry points through `oeedger8r`-generated bridges, but instead opted to manually scrutinize arguments for these special `ecalls`. We audited all eight built-in entry points, and confirmed that most of them were carefully written to prevent pointer sanitization issues, as well as more subtle attack vectors like TOCTOU and speculative execution side-channels. However, we found a critical issue in the built-in `_handle_get_sgx_report()` `ecall` involved in crucial attestation functionality (see Appendix A.2 for full code). This function copies the untrusted report input buffer into enclave memory, but never validates whether the argument pointer passed by the untrusted runtime actually lies outside the enclave. This evidently leads to corruption of trusted memory, e.g., when writing the return value in the fall-through branch similar to the `oeedger8r`-generated code discussed above.

Both of the above vulnerabilities allow to write a fixed failure code (`0x03000000` and `0x01000000`) to an arbitrary in-enclave memory location. We developed a PoC based on an existing file-encryptor OE example application, and successfully exploited the above vulnerabilities to forcefully overwrite the first round keys of the AES cipher. This could be extended by overwriting all but the final round keys with known values to perform full key extraction.

Google Asylo. Because Google's Asylo [13] framework is built on top the existing Intel SGX-SDK, it also inherits Intel's `edger8r`-based input sanitization scheme. Particularly, the Asylo trusted runtime features a small number of predefined `ecall` entry points, specified in EDL, that implement the necessary functionality to present a higher-level, RPC-like message passing abstraction to the application programmer. Considering that Asylo's runtime extends the trusted computing base on top of Intel's existing SGX-SDK, we were interested to assess whether the extra abstraction level may also bring *additional* attack surface. This may for instance be the case when making use of the unsafe `[user_check]` EDL attribute [19] that explicitly weakens `edger8r` guarantees and puts the burden of pointer validation on the programmer (e.g., to allow for application-specific optimizations in performance-critical scenarios). Manually scrutinizing the EDL specifications of Asylo's trusted runtime, we found 14 instances of the problematic `[user_check]` attribute. We reviewed these instances and alarmingly found that several of them lacked proper pointer validation, leaving critical vulnerabilities in the compiled enclave (e.g., a write-zero primitive). Notably, the developers took care to validate *second-level* input buffers in the untrusted argument structure, but failed to validate the argument pointer itself (cf. Appendix A.3 for a relevant sample).

Graphene-SGX. While Graphene-SGX's [43] untrusted world interaction and pointer validation concerns are largely limited to `ocalls` (cf. Sections 5.3 and 5.5), our inspection of the narrow `ecall` interface revealed a rather subtle type of implicit pointer passing that was overlooked. Namely, Graphene's trusted runtime never validates the `argv` and `envp` pointers, which are passed from the untrusted runtime all the way into the main function of the shielded application binary. As a result, adversaries can for instance leak arbitrary in-enclave memory when the trusted application outputs `argv` values (e.g., in case of an unknown command line argument). We experimentally confirmed this attack by means of an elementary `echo` program, which unknowingly prints in-enclave secrets after overriding `argv[1]` in the *untrusted* runtime. With respect to mitigations, note that properly sanitizing string arguments can be non-trivial in itself, as explored in Section 5.2.

We also found that the special `enclave_ecall_thread_start()` trusted runtime function unconditionally redirects control flow, without performing any validation on the provided untrusted function pointer. We successfully exploited this to jump to arbitrary in-enclave locations, hence allowing code reuse attacks [41].

SGX-LKL. Our analysis of the open-source SGX-LKL `ecall` interface revealed the exact same vulnerability. That is, the trusted `__sgx_init_enclave()` libOS function passes the untrusted `argv` pointer directly to the shielded application without any prior sanitization. We experimentally confirmed that this vulnerability can be abused for information leakage, similar to the above exploit.

Further, the in-enclave signal handler `ecall` entry point does not check that the `siginfo` `struct` pointer provided by the untrusted runtime lies outside the enclave. This vulnerability can be abused in certain scenarios to leak in-enclave memory contents. For instance, we describe a full exploit for the `SIGILL` signal in Appendix A.4.

Sancus. To demonstrate that untrusted pointer dereference vulnerabilities are *not* limited to advanced virtual memory-based architectures, we also reviewed the trusted runtime and infrastructural enclaves of the low-end open-source Sancus [32] TEE for embedded TI MSP430 devices. As with the above runtimes, we focused our security audit on the enclave boundary code only.

A first critical vulnerability was found in a recent extension [31] to the Sancus compiler infrastructure, which implements a high-level authenticated message passing abstraction to develop distributed event-driven enclave programs. Much like Intel's `edger8r`, the Sancus compiler fully automatically generates `ecall` bridge code to transparently marshal, decrypt, and authenticate input buffers, which can be subsequently processed by the shielded application. We found that the compiler-generated bridge code does *not* sanitize untrusted pointer arguments (cf. Appendix A.5). This may be exploited to forcefully decrypt enclave secrets.

A second input pointer validation vulnerability was found in an infrastructural trusted loader enclave [14] that decrypts third-party application enclaves to preserve code confidentiality. We noticed that the trusted loader enclave code lacks any input pointer validation checks, allowing us to build an arbitrary write primitive in enclave memory. We successfully exploited this vulnerability in a PoC that launches a ROP-style [41] control flow hijacking attack by corrupting the loader enclave call stack.

5.2 Validating string arguments

In case the enclave interface is written in a low-level language like C, string arguments do not carry an explicit length and may not even have been properly null-terminated. Thus, shielding runtimes need to first determine the expected length and always include a null terminator when copying the string inside the enclave.

Attack vector #5 (strings): Runtimes should avoid computing untrusted string sizes, and always include a null byte at the expected end. ▷ At least one related instance repeated across two production SDKs.

TEE design. We show below how computing on unchecked string pointers may leak enclave secrets through side-channels, even if the `ecall` is eventually rejected. While side-channels are generally a known issue across TEE technologies [8, 21, 34, 46] and may even be observed by non-privileged adversaries, for example by measuring overall execution time [30] or attacker-induced cache evictions [26, 39], we show that TEE-specific design decisions can still largely affect the overall exploitability of subtle side-channel vulnerabilities. Particularly, we develop a highly practical attack that abuses several privileged adversary capabilities that have previously been proven notorious in the Intel SGX design, e.g., untrusted page tables [47, 54], interrupts [24, 45, 46], and storing interrupted CPU register contents in SSA memory frames [5, 44].

Intel SGX-SDK. We discovered that `edger8r`-generated code may be tricked into operating on unchecked in-enclave pointers when computing the size of a variable-length input buffer. While such illegal `ecall` attempts will always be properly rejected, we found that adversaries can exploit the unintended size computation as a deterministic oracle that reveals side-channel information about arbitrary in-enclave memory locations. This vulnerability is tracked via CVE-2018-3626 (Intel SA-00117), leading to enclave TCB recovery and changes in the EDL specification [18]. Prior to our disclosure, EDL allowed programmers to specify a custom `[sizefunc]` attribute that takes as an argument an *unchecked* pointer to an application-specific structure, and returns its size. Likewise, there is a dedicated `[string]` EDL attribute to specify null-terminated

```
1  static sgx_status_t SGX_CDECL sgx_my_ecall(void* pms)
2  {
3    CHECK_REF_POINTER(pms, sizeof(ms_my_ecall_t));
4    ms_my_ecall_t* ms = SGX_CAST(ms_my_ecall_t*, pms);
5    char* _tmp_s = ms->ms_s;
6
7  ★ size_t _len_s = _tmp_s ? strlen(_tmp_s) + 1 : 0;
8    char* _in_s = NULL;
9
10   CHECK_UNIQUE_POINTER(_tmp_s, _len_s);
11   __builtin_ia32_lfence(); // fence after pointer checks
12   ...
```

Listing 3: Proxy function generated by `edger8r` for the EDL specification: `public void my_ecall([in,string] char *s);`.

string arguments. Essentially, this special case comes down to `[sizefunc=strlen]`.

Consider the code skeleton generated by `edger8r` in Listing 3 for an `ecall` that expects a single string pointer argument. In order to verify that the complete string is outside the enclave, the trusted edge routine *first* computes the size of the argument buffer (through either `strlen()` or a dedicated `sizefunc` in general), and only *thereafter* checks whether the entire buffer falls outside of the enclave. It is intended that the edge code first determines the length in untrusted memory, but we made the crucial observation that the `strlen()` invocation at line 7 operates on an arbitrary unchecked pointer, potentially pointing into enclave memory. Any pointer poisoning attempts will subsequently be rejected at line 10, but the unintended computation may have already leaked information through various side-channels [24, 45]. In general, leakage occurs whenever there is secret-dependent control or data flow in the specified `sizefunc`. This is most obviously the case for the common `[string]` EDL attribute, since the amount of loop operations performed by `strlen()` reveals the number of non-zero bytes following the specified in-enclave pointer.

Our attack builds on top of the open-source SGX-Step [45] enclave interrupt framework to turn the subtle `strlen()` side-channel leakage into a fully *deterministic oracle* that reveals the exact position of all `0x00` bytes in enclave private memory (thereby for instance fully breaking the confidentiality of booleans or providing valuable information for cryptanalysis). Particularly, we use SGX-Step to reliably step the `strlen()` execution, one instruction at a time, leveraging the "accessed" bit in the page table entry of the targeted in-enclave memory location as a noise-free oracle that is deterministically set by the processor for every `strlen()` loop iteration [47]. We confirmed that our single-stepping oracle continues to work reliably even when the victim enclave was compiled to a single, extremely compact `rep movsb` instruction (x86 string operations can indeed be interrupted in between every loop iteration [17]).

We developed a practical end-to-end AES-NI key extraction PoC in an application enclave built with a vulnerable version of `edger8r`. Our victim enclave provides a single, multi-threaded `ecall` entry point that encrypts the first 16 bytes of a given string using side-channel resistant AES-NI instructions with a secret in-enclave key. Since AES-NI operates exclusively on CPU registers (e.g., `xmm0`) and due to the limited nature of the `strlen()` side-channel, we cannot perform key extraction by directly targeting the AES state or key in memory. Instead, our attack uses repeated encryption `ecalls`, assuming varying (but not necessarily known) plaintext and known

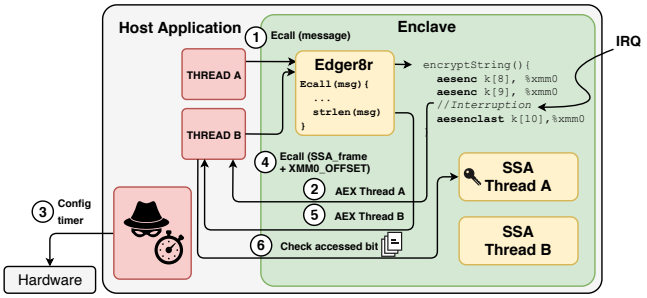

Figure 4: Overview of the key extraction attack exploiting `strlen()` side-channel leakage in Intel SGX-SDK.

ciphertext. We further abuse that the Intel SGX architecture enables a privileged adversary to precisely interrupt a victim enclave at a chosen instruction-level granularity [45], thereby forcing the processor to write the register state to a fixed SSA location in enclave memory (this includes the `xmm` registers that are part of the `XSAVE` region of the SSA frame). Figure 4 depicts the high-level phases of the attack flow, using two threads A and B:

(a) Invoke the encryption `ecall` from thread A ① and interrupt the enclave ② before the final round of the AES (*i.e.*, before the `aesenclast` instruction). To keep the PoC simple, we achieve this requirement by inserting an access to a dummy page at the appropriate point, and catching accesses to this page in a signal handler on the untrusted side. Note that in a real-world attack, the single-stepping feature of SGX-Step could be used to execute the victim enclave exactly up to this point, without relying on a more coarse-grained page fault for interruption.

(b) While the `ecall` in thread A is interrupted, prepare the timer used by SGX-Step ③ and launch a second thread B ④ to probe the position of the first zero byte (if any) in the intermediate AES state. Concretely, this involves a second `ecall` to the same entry point, but this time supplying an illegal in-enclave target address pointing to the fixed memory location containing the `xmm0` register in the SSA frame of the interrupted thread A. Each time when a timer interrupt arrives ⑤, we monitor and clear ⑥ the "accessed" bit of the targeted SSA page table entry.

(c) After the `strlen()` probing has finished, the obtained leakage is stored alongside the corresponding ciphertext, and thread A is resumed by restoring read/write access to the dummy page.

(d) Repeat from step (a) with a different plaintext until the full key has been recovered (see Algorithm 1).

Experimentally, we determined that this attack succeeds with 881 AES invocations on average (over 1000 runs with random keys, minimum: 306, maximum: 3346), given a deterministic, noise-free `strlen()` oracle. Note that this attack could also be adapted to work with noisy measurements, using the so-called zero-value model known from hardware side-channel attacks [12]. Besides, the attack would also be applicable when targeting the first round of the AES in a known-plaintext scenario.

Properly closing this side-channel requires profound changes in the way `edger8r` works. Notably, the bridge code includes an `lfence` instruction at line 11 to rule out advanced Spectre-v1 misspeculation attacks that might still speculatively compute on unchecked

Algorithm 1 `strlen()` oracle AES key recovery where $S(\cdot)$ denotes the AES SBox and $SR(p)$ the position of byte p after AES ShiftRows.

while not full key K recovered **do**
  $(P, C, L) \leftarrow$ random plaintext, associated ciphertext, strlen oracle
  if $L < 16$ **then**
    $K[SR(L)] \leftarrow C[SR(L)] \oplus S(0)$
  end if
end while

pointers before they are architecturally rejected. However, our attack is immune to such countermeasures because we directly observe side effects of normal, non-speculative execution. Further, early rejecting the `ecall` when detecting that the start pointer falls inside the enclave does not suffice in general. In such a case, adversaries might still pass pointers below the enclave base address, and observe secret-dependent behavior based on the first bytes of the enclave. Intel implemented our recommended mitigation strategy by dropping support for the superfluous `[sizefunc]` EDL attribute entirely, and further abstaining from computing untrusted buffer sizes inside the enclave. Instead, alleged buffer sizes are computed outside the enclave, and passed as an *untrusted* argument, such that the `CHECK_UNIQUE_POINTER` test can take place immediately. For the `strlen()` case, the untrusted memory can simply be copied inside, and an extra null byte inserted at the alleged end. This solution conveniently moves all secret-dependent control flow from the enclave into the untrusted application context.

Microsoft Open Enclave SDK. After Intel had properly patched the `strlen()` side-channel vulnerability in the SGX-SDK, OE appears to have tried to adopt our proposed mitigation strategy of passing an *untrusted* alleged string length into the enclave. However, after reviewing the generated code, we found that `oeedger8r` fails to include a `0x00` terminator byte after copying the untrusted string inside enclave memory (cf. Appendix A.6). This critical oversight can be exploited to trick the shielded enclave application into operating on non-null-terminated strings. The trusted user function will incorrectly assume that the string is properly terminated and may perform out-of-bounds memory read/writes, hence turning a mitigation for a subtle and functionally correct side-channel issue into a more dangerous source of enclave memory corruption. This OE vulnerability is tracked via CVE-2019-0876 and specific to enclaves that expect EDL string arguments, and output or manipulate them in-place (e.g., `strcpy()`).

We experimentally demonstrated this vulnerability by means of a minimal PoC application enclave which overwrites all non-alphanumeric chars in a string with `0x20`, until the null terminator is encountered. If this enclave operates on an unterminated string, the length field of the subsequent heap frame is corrupted, which subsequently can be further leveraged in more complex exploits.

5.3 Validating variable-sized buffers

Multi-byte input buffers are commonly specified by passing a pointer to the start of the buffer and an associated size. In order to properly validate such buffers, the trusted runtime should first compute the end pointer by adding the alleged size argument, and thereafter assert that the complete input buffer address range falls outside the enclave. However, since the buffer size is an adversary-controlled

parameter, care should be taken to prevent the pointer addition from overflowing and silently wrapping around the address space.

Attack vector #6 (integer overflow): Runtimes should use safe arithmetics when computing addresses in a buffer with untrusted size. ▷ Relatively well-understood in production SDKs, not in research code.

TEE design. We found that the address-related vulnerabilities in this section are significantly more exploitable in TEE designs that provide increased attacker control over the shared memory and enclave memory layouts. For instance, some integer overflow vulnerabilities require the adversary to control the enclave base address in a shared address space, as is the case for the Intel SGX [8] and Sancus [32] designs, but not for ARM TrustZone [34] or Keystone [21]. Further, we found that logical errors may arise when checking variable sized buffers in a shared address space. As detailed below, the exploitability of such logic bugs depends heavily on the ability of the adversary to trigger certain edge cases (e.g., passing a pointer that lies just before the enclave base address), which might also be considerably easier in single-address space TEE designs like Intel SGX or Sancus.

Fortanix Rust-EDP. In contrast to the other runtimes described in this paper, Fortanix's EDP [11] leverages the type system of the safe Rust language to disallow inadvertent untrusted pointer dereferences apart from the dedicated `UserSafe` type, which transparently sanitizes any pointers passed into the enclave. Rust-EDP's shielding system has been explicitly designed to avoid known enclave boundary attacks and implements libOS-like functionality through a deliberately very narrow `ocall` interface that is kept invisible to the application programmer. However, our analysis shows that the promising approach of enforcing pointer sanitization through the use of a type system may evidently still suffer from security issues if the implementation in the type itself is incorrect.

We manually scrutinized the implementation of the confined `UserSafe` type (part of the Rust compiler's SGX-EDP target [11]) and found a potentially exploitable integer overflow vulnerability in the pointer validation logic. Listing 4 shows the relevant `is_user_range()` function, which checks whether an untrusted memory range specified by a pointer and length falls completely outside the enclave. Concretely, we observed that the 64-bit integer addition to compute the `end` pointer at line 4 may overflow. Note that Rust can automatically detect integer overflows, but these runtime checks are only enabled in debug mode, meaning that in production builds (e.g., `rustc -C debug-assertions=off`), integer overflows do not cause an error by default [25].

We confirmed (after isolating the validation function in a dummy Rust test program) that said function can be made to early-out and return `true` at line 5 even when passing an illegal in-enclave pointer if the enclave base is near the top of the address space. Note that Intel SGX leaves the enclave base address under explicit attacker control [8], so this requirement may be satisfied by real-world attackers. For example, the untrusted runtime can return a specially-crafted pointer from the `alloc()` usercall, potentially leading to in-enclave memory disclosure or corruption, depending on how the pointer is further used within the enclave. After our disclosure, the EDP trusted runtime now explicitly asserts that untrusted sizes returned by `alloc()` do not overflow.

```
1 /// `true` if the specified memory range is in userspace.
2 pub fn is_user_range(p: *const u8, len: usize) -> bool {
3   let start = p as u64;
4 ★ let end = start + (len as u64);
5   end <= image_base() || start >= image_base() + (unsafe {
        ENCLAVE_SIZE } as u64) // unsafe ok: link-time constant
6 }
```

Listing 4: Pointer validation in the Rust-EDP `UserSafe` type.

Google Asylo. Apart from the aforementioned `[user_check]` issues, the entry points in Asylo's trusted runtime take care to validate all second-level input buffers. However, our code review also revealed a subtle logic mistake in the input validation logic itself. That is, we observed that many of the trusted runtime functions (cf. Appendix A.3 for a relevant sample) rely on the `TrustedPrimitives::IsTrustedExtent(input,input_size)` library function returning `true` to reject the `ecall` attempt when detecting that an untrusted input buffer is *completely* contained within enclave memory.

While this function itself translates to the corresponding `sgx_is_within_enclave()` primitive from the SGX-SDK, which is indeed correct and free from integer overflow vulnerabilities, the logic mistake occurs when considering malicious input buffers that only partly *overlap* with untrusted and enclave memory. For instance, `IsTrustedExtent()` will properly return `false` and the `ecall` will still be allowed when passing a lengthy adversarial input buffer that starts one byte before the enclave base address but continues into the enclave memory range. Evidently, this may subsequently lead to trusted enclave memory corruption or disclosure. Hence, the trusted runtime should instead make use of the proper `sgx_is_outside_enclave()` SGX-SDK primitive.

Attack vector #7 (outside ≠ ¬inside): In a shared address space, input buffers should not fall *partially* inside the trusted memory region. ▷ Generally understood in production SDKs, not always in research code.

Graphene-SGX. We discovered a critical integer overflow vulnerability in the widely used pointer range validation function that often computes on untrusted attacker-provided sizes (similar to the Rust-EDP issue described above). We further found that Graphene-SGX suffers from the same subtle logic mistake that we spotted in the Asylo code base: at the time of our review, there was no `sgx_is_outside_enclave()` primitive, and all instances of the intended "abort if not completely outside" were erroneously checked for "abort if completely inside enclave" (cf. Listing 5 for a relevant sample). A related type of pointer validation vulnerabilities arises when the libOS allocates variable-sized output buffers in untrusted memory outside the enclave to be able to exchange data for `ocall` arguments and return values. For performance reasons, Graphene-SGX allocates such shared memory buffers directly on the untrusted host stack. While the untrusted host stack pointer is indeed validated to lie outside of enclave memory upon enclave entry, we observed that the trusted libOS does *not* properly check whether the untrusted stack does not overflow into enclave memory after allocating a new shared memory buffer in the widely used `OCALLOC` macro. Depending on the specific `ocall` implementation, the enclave will subsequently copy data to/from the inappropriately allocated buffer, leading to information disclosure and/or memory corruption.

Keystone. While Keystone [21] is still a research prototype and lacked essential functionality when we reviewed its code, we discovered and reported a potential integer overflow vulnerability (cf. Appendix A.7) in the trusted security monitor's `detect_region_overlap()` function, which is used during the creation of an enclave. However, this overflow was not directly exploitable due to certain restrictions on region sizes in the Keystone codebase.

Sancus. We found both logical errors and integer overflow vulnerabilities in the `sancus_is_outside_sm()` function provided by the trusted runtime. Particularly, the current implementation does not properly detect an untrusted buffer that spans the entire enclave address range, or a carefully crafted length specifier that triggers an integer overflow to wrap around the 16-bit address space.

5.4 Pointer-to-pointer validation pitfalls

While the previous sections have focussed on the *spatial* aspect of untrusted pointer dereferencing, we also found more subtle vulnerabilities related to the *temporal* aspect. That is, whenever a pointer points to an untrusted address or size (as it is often the case, for instance, in marshalling structs), the runtime should take care to first copy the second-level pointer value to a trusted location in enclave memory before applying the sanitization logic. If this is not the case, adversaries may overwrite the second-level pointer in untrusted memory after the validation has succeeded but before the pointer is dereferenced in the enclave code. This class of vulnerabilities is also referred to as "double fetch" bugs in operating system kernels [37, 53].

Attack vector #8 (double fetch): Untrusted pointer values should be copied inside the enclave *before* validation to avoid time-of-check time-of-use. ▷ Relatively well-understood (once pointer sanitization is applied).

TEE design. Double fetch bugs typically rely on a very narrow vulnerability time window and hence can be notoriously hard to exploit in traditional user-to-kernel contexts. However, recent research demonstrated how some TEE design decisions may considerably simplify exploitation of synchronization bugs in enclaves. AsyncShock [50] exploits that Intel SGX adversaries may provoke page faults in the enclaved execution, and SGX-Step [45] similarly abuses that privileged SGX adversaries may abuse system timers to very precisely interrupt a victim enclave after *every* single instruction. Finally, Schwarz et al. [37] use a cache side-channel to expose double fetch bugs in both Intel SGX and ARM TrustZone TEEs.

Graphene-SGX. Scrutinizing Graphene-SGX's `ocall` interface, we found several instances of exploitable double fetch vulnerabilities. Listing 5 provides a relevant code snippet that attempts to sanitize the result of the `sock_accept` system call. First, at line 1, a buffer `ms` is allocated in *untrusted* memory outside the enclave. The `struct` buffer pointed to by `ms` contains another pointer `ms->ms_addr` that will be initialized by the untrusted runtime to point to the socket address returned by the system call. As `ms->ms_addr` is an untrusted pointer, the libOS shielding system attempts to properly validate that it lies outside the enclave at line 5 (modulo the logic bug described in Section 5.3) before dereferencing `ms->ms_addr` a second time when copying the socket address buffer inside at line 11. However, since the parent `ms` `struct` was allocated in untrusted memory and has never been copied inside, SGX adversaries can interrupt

```
1  OCALLOC(ms, ms_ocall_sock_accept_t *, sizeof(*ms));
2  ...
3  retval = SGX_OCALL(OCALL_SOCK_ACCEPT, ms);
4  if (retval >= 0) {
5★   if (len && (sgx_is_within_enclave(ms->ms_addr, len)
6                || ms->ms_addrlen > len)) {
7      OCALL_EXIT();
8      return -PAL_ERROR_DENIED;
9    }
10   ...
11★  COPY_FROM_USER(addr, ms->ms_addr, ms->ms_addrlen);
```

Listing 5: Double fetch vulnerability in Graphene-SGX.

the enclave in between lines 5 and 11 and trivially overwrite the `ms_addr` field with an arbitrary in-enclave address, potentially leading to trusted memory disclosure.

5.5 Validating ocall return values

Apart from validating `ecall` arguments, the enclave trusted runtime should also take care to properly scrutinize `ocall` return values when passing pointers or sizes back into the enclave.

Attack vector #9 (Iago): Pointers or sizes returned through `ocalls` should be scrutinized [4]. ▷ Understood, but still prevalent in research libOSs that shield system calls; one instance in a production SDK.

TEE design. We found that the complexity of the shielding system may largely affect this attack surface. That is, SDK-based approaches typically do not feature a large built-in `ocall` interface, whereas libOSs should safeguard against Iago attacks [4] by scrutinizing return values from the complex system call interface before passing them on to the shielded application.

Microsoft Open Enclave SDK. OE's trusted runtime includes a `oe_get_report()` function which is used to provide attestation functionality to the enclaved binary. Internally, this function performs the same `ocall` twice; the first time specifying the output buffer as a null pointer in order to obtain the required quote size. Based on this size, a buffer is allocated on the enclave heap, and subsequently filled through a second `ocall` invocation. We found, however, that the untrusted runtime can return different sizes for the two `ocall` invocations (tracked via CVE-2019-1369). Particularly, the in-enclave buffer is allocated based on the size obtained from the first `ocall`, whereas the size returned by the second `ocall` is passed on to the caller of `oe_get_report()`. Hence, returning an unexpectedly large size in the second `ocall` invocation may cause the enclave application to read or write out of bounds. We experimentally confirmed that OE's remote attestation example enclave can leak up to 10 kB of trusted heap memory (this upper bound is due to an internal limit), possibly at multiple heap locations depending on other memory allocations.

LibOS-based runtimes. We discovered several exploitable instances of Iago attacks [4] in Graphene-SGX's `ocall` interface. For example, an untrusted system call return value `len` is later used to copy `len` bytes from untrusted memory into a fixed-size buffer inside the enclave, leading to arbitrary write-past the in-enclave buffer. To demonstrate this vulnerability, we developed a PoC where the `readdir()` system call in the untrusted runtime returns an unexpected length, causing an out-of-bounds write in the enclave.

Similarly, in SGX-LKL's `ocall` interface, we found several instances of Iago vulnerabilities where for example the untrusted pointers returned by `mmap()` are not checked to lie outside of enclave memory, or the untrusted length returned by `write()` is passed unsanitized back to the shielded application. To demonstrate how this can be successfully exploited, we developed an elementary victim application featuring a common programming idiom where `write()` is used to output a buffer piecewise, each time advancing a pointer with the number of bytes successfully written (*i.e.*, the system call's return value). We modified the untrusted runtime to unexpectedly increment the return value of the `write()` system call, causing the shielded application binary to output secret enclave memory beyond the buffer bounds. Finally, we also confirmed and reported the existence of similar issues in Google Asylo.

Keystone. Similar to the above SGX runtimes, Keystone provides system call wrappers to simplify porting of existing code to an enclave. While Keystone documentation indicates that the developers are aware of potential issues, the codebase currently lacks mitigations against Iago attacks. Hence, we developed an exploit using the `write()` system call, similar to the SGX-LKL PoC.

5.6 Scrubbing uninitialized structure padding

Apart from pointers and size arguments, enclaves may also pass composite `struct` types to the untrusted world. While, as with all output buffers, we assume that enclave applications do not intentionally disclose secrets through the program-visible state (*i.e.*, the `struct`'s individual members), prior research on operating system kernel [7] and SGX enclave [23] interfaces has shown that padding bytes silently added by the compiler may still unintentionally leak uninitialized secret memory.

Attack vector #10 (uninitialized padding): Scrubbing program-visible state may not suffice for `struct` outputs [23]. ▷ Especially relevant for production SDKs that expose the enclave interface to the programmer.

TEE design. This subtle attack vector cannot be easily mitigated by sanitizing program-visible API state. Possible mitigations include securely initializing the entire output `struct` using `memset()` and/or doing a member-wise deep-copy, or declaring the output `struct` as "packed" so the compiler does not unknowingly introduce padding. However, both solutions require application-specific knowledge about the exact `struct` types being passed. As an important insight, we therefore found that this attack vector can only be transparently shielded when the enclave interface is predefined and fixed. That is, the fixed `ocall` interface in libOS-based runtimes can indeed be manually scrutinized for this type of vulnerabilities. However, this is not the case for SDK-based runtimes that offer a *generic* enclave interface defined by the programmer, and hence (opposed to their shielding responsibility) ultimately outsource the responsibility of scrubbing uninitialized `struct` padding to the application developer.

SDK-based runtimes. Lee et al. [23] first demonstrated how uninitialized `struct` padding may pose a subtle information leakage source in the `edger8r`-generated code of the Intel SGX-SDK. Building on their findings, we generalized this attack vector to also demonstrate its applicability to `oeedger8r`-generated code in Microsoft's Open Enclave SDK, as well as in the Sancus TEE. Similarly, we confirmed that padding leakage can also occur in Keystone, e.g., through the padding of `calc_message_t` in the demo enclave.

LibOS-based runtimes. We reviewed the `ocall` interfaces in the libOS-based runtimes we studied (Graphene-SGX, LKL, Rust-EDP). Rust-EDP appears to be free of such issues, and Graphene-SGX explicitly enforces `struct` packing through a compiler `#pragma`. However, SGX-LKL contains at least two instances of an `ocall` using a `struct` with potentially vulnerable padding bytes (`sigaction` and `siginfo_t`). In Google Asylo, most `structs` passed through an `ocall` are explicitly declared as packed, however, we found one instance of a padded `struct BridgeSignalHandler` used in the syscall interface.

6 DISCUSSION AND GENERAL MITIGATIONS

The most intuitive solution to defend against our attacks is to incorporate additional checks in the enclave code to properly sanitize ABI state and API arguments/return values. When properly implemented, such checks suffice to block all of the attacks described in this work, and they have indeed been adopted by the various projects we analyzed. However, leaving the decision of whether (and how) to correctly implement numerous interface validation checks to enclave developers, who are likely unaware of this class of vulnerabilities, may be problematic. Moreover, even when developers think about inserting the necessary checks, our analysis has revealed several recurring pitfalls, including subtle logical bugs, side-channels, double fetches, and integer overflows. This highlights the need for more principled approaches to rule out this class of vulnerabilities at large, as well as defense-in-depth code hardening measures that may raise the bar for successful exploitation.

Code hardening. Interface sanitization vulnerabilities are closely related to a wider class of memory safety issues [3, 22], and their exploitation may hence be partially hindered by established techniques such as heap obfuscation (cf. Section 4.1). Furthermore, SGX-Shield [40] aims to obstruct memory corruption attacks by randomizing the memory layout of enclaved binaries shielded by the Intel SGX-SDK. However, prior research [3] has shown that SGX-Shield does *not* randomize the trusted runtime, meaning that the code we studied would still feature a deterministic and static memory layout, and may offer numerous gadgets for mounting code reuse attacks. Further, as the trusted runtime also forms an integral part of SGX-Shield's loader [40], any memory safety or side-channel vulnerabilities in the trusted runtime itself may also be used to disrupt the preliminary randomization stage. While randomizing the memory layout of the trusted runtime would indeed be desirable, this constitutes a non-trivial task [3, 40] given its low-level nature, including hand-written assembly code and static memory addresses expected by SGX's `eenter` and `eresume` instructions. In this respect, we want to emphasize that some of the attacks we presented are free from non-static address dependencies, and hence remain inherently immune to software randomization schemes. For example, the SGX-SDK `strlen()` oracle in Fig. 4 depends solely on the *fixed* address of the victim's SSA frame, which is deterministically dictated by the SGX hardware and immutable from software.

As a perpendicular code hardening avenue, we recommend to implement more aggressive responses when detecting pointer violations in the trusted runtime. That is, most of the runtimes we

studied merely reject the `ecall` attempt when detecting pointer poisoning. In the SGX-SDK `strlen()` oracle attack of Section 5.2, we for example abused this to repeatedly call a victim enclave, each time passing an illegal pointer and making side-channel observations before the `ecall` is eventually rejected. To rule out such repeated attacks, and reflecting that in-enclave pointers represent clear adversarial or buggy behavior, we recommend to immediately destroy secrets and/or initiate an infinite loop upon detecting the *first* pointer poisoning attempt in the trusted runtime.

Hardware-assisted solutions. As a more principled approach to rule out the confused deputy attacks described in this paper, solutions could leverage finer-grained memory protection features in the processor. In particular, tagged memory [51] or capability architectures [52] appear to be a promising approach to inherently separate the memory domains of untrusted and trusted code. On a capability machine [52], pointers are represented at run-time as unforgeable objects carrying associated permissions and length fields. The machine ensures that untrusted code can never create a valid capability that points inside enclave-private memory and pass it as an argument to an `ecall`, thereby eradicating an entire class of pointer dereference vulnerabilities architecturally.

As an example of an alternative tagged memory design, the recently proposed Timber-V [51] architecture provides lightweight and strong enclaved execution on embedded RISC-V platforms. Timber-V processors offer enhanced MPU isolation by keeping track of a 2-bit tag for every memory word, allowing individual memory locations to be associated with one out of 4 possible security domains. The CPU further restricts tag updates, and offers checked memory load/store operations, which take an *expected* tag as an argument and trap whenever the actual memory location being dereferenced does not match the expected tag. Hence, any pointer poisoning attempts by untrusted code outside the enclave would be immediately caught by the hardware.

The untrusted pointer dereference issues we identified in this work bear some similarities with how privileged OS kernel code needs to properly sanitize user space pointers in e.g., system call arguments. As a defense-in-depth mechanism, recent x86 processors support Supervisor Mode Access Protection (SMAP) features to explicitly disallow unintended user space pointer dereferences in kernel mode [17]. We encourage further research to investigate porting such CPU features to enclave mode.

Safe programming languages. The combination of TEEs and safe programming languages, such as Rust, has been proposed as a promising research direction to safeguard enclave program semantics, but still requires additional interface sanitizations [48]. The approach of Fortanix's Rust-EDP [11] shows how the compiler's type system can be automatically leveraged to limit the burden of pointer sanitization concerns from a cross-cutting concern throughout the enclave code base to the correct implementation of a single untrusted pointer type. However, it is important to note that safe languages by themselves are not a silver bullet solution to our attacks. That is, the trusted runtime code remains responsible to bootstrap memory safety guarantees by (*i*) establishing expected ABI calling conventions in the low-level entry assembly code, and (*ii*) providing a correct implementation of sanitization in the untrusted pointer type. In this respect, the subtle integer overflow vulnerability in Fortantix's EDP, presented in Section 5.3, demonstrates that developing both the trusted runtime libraries and the enclave in safe Rust may still not suffice to fully eradicate pointer sanitization vulnerabilities.

Finally, as an alternative to Intel's `edger8r` tool, the use of separation logic has been proposed to automatically generate secure wrappers for SGX enclaves [49]. This approach aims to provide the advantages of safe languages, and even formal verification guarantees, but still relies on explicit developer annotations.

7 CONCLUSIONS AND FUTURE WORK

Our work highlights that the shielding responsibilities in today's TEE runtimes are not sufficiently understood, and that various security issues exist in the respective trusted computing bases. We showed that this attack surface is large and often overlooked: we have identified 35 interface sanitization vulnerabilities in 8 open-source TEE runtimes, including production-quality SDKs written by security-savvy developer teams. Our analysis further reveals that the entry points into this attack surface are more pervasive than merely argument pointers: we contributed a classification of 10 recurring vulnerability classes spanning the ABI and API tiers.

In the defensive landscape, our work emphasizes the need to research more principled interface sanitization strategies to safeguard the unique TEE shielding responsibilities. We particularly encourage the development of static analysis tools, and fuzzing-based vulnerability discovery and exploitation techniques to further explore this attack surface.

ACKNOWLEDGMENTS

We thank Jethro Beekman (Fortanix), Job Noorman (KU Leuven), and Johannes Götzfried for insightful discussions, and the anonymous reviewers for constructive feedback that helped improving the paper. We further would like to thank the maintainers of the open-source projects we studied for their contributions to the community and for promptly responding and working on mitigations.

This research is partially funded by the Research Fund KU Leuven, and by the Agency for Innovation and Entrepreneurship (Flanders). Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO). This research is partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/R008000/1 and by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM). Abdulla Aldoseri is supported by a stipend from the University of Bahrain.

REFERENCES

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 689–703.

[2] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 267–283.

[3] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *Proceedings of the 27th USENIX Security Symposium*. 1213–1227.

[4] S. Checkoway and H. Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 253–264.

[5] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[6] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 5:1–5:5.

[7] J. Corbet. 2010. Structure holes and information leaks. online, accessed 2019-08-06: https://lwn.net/Articles/417989/. (December 2010).

[8] V. Costan and S. Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[9] J. Edge. 2008. CVE-2008-1367 Kernel doesn't clear DF for signal handlers. https://bugzilla.redhat.com/show_bug.cgi?id=437312. (March 2008).

[10] A. Fog. 2018. Calling conventions for different C++ compilers and operating systems. http://www.agner.org/optimize/calling_conventions.pdf. (April 2018).

[11] Fortanix. 2019. Fortanix Enclave Development Platform – Rust EDP. online, accessed 2019-08-30: https://edp.fortanix.com/. (2019).

[12] J. D. Golić and C. Tymen. 2003. Multiplicative Masking and Power Analysis of AES. In *Cryptographic Hardware and Embedded Systems (CHES)*. 198–212.

[13] Google. 2019. Asylo: An open and flexible framework for enclave applications. online, accessed 2019-08-06: https://asylo.dev/. (2019).

[14] J. Götzfried, T. Müller, R. De Clercq, P. Maene, F. Freiling, and I. Verbauwhede. 2015. Soteria: Offline software protection within low-cost embedded devices. In *Annual Computer Security Applications Conference (ACSAC)*. 241–250.

[15] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *International Symposium on Engineering Secure Software and Systems (ESSoS '18)*. Springer, 44–60.

[16] N. Hardy. 1988. The Confused Deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review* 22, 4 (1988), 36–38.

[17] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 325384 (2016).

[18] Intel. 2018. *Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Edger8r Generated Code Side Channel Exploits*. Revision 1.0.

[19] Intel. 2019. Intel Software Guard Extensions – Get Started with the SDK. online, accessed 2019-05-10: https://software.intel.com/en-us/sgx/sdk. (2019).

[20] E. Mohammadian Koruyeh, K. N Khasawneh, C. Song, and N. Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[21] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović. 2019. Keystone: A Framework for Architecting TEEs. *arXiv preprint arXiv:1907.10119* (2019).

[22] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. Byunghoon Kang. 2017. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *Proceedings of the 26th USENIX Security Symposium*. 523–539.

[23] S. Lee and T. Kim. 2017. Leaking Uninitialized Secure Enclave Memory via Structure Padding. *arXiv preprint arXiv:1710.09061* (2017).

[24] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium*. 557–574.

[25] G. Lehel and N. Matsakis. 2017. rust-lang RFC: Integer overflows in Rust. online, accessed 2019-05-10: https://github.com/rust-lang/rfcs/blob/9ef0c35/text/0560-integer-overflow.md. (2017).

[26] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. 2016. Armageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium*. 549–564.

[27] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. Ryn Choe, C. Kruegel, and G. Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *NDSS 2017*.

[28] Pieter Maene, Johannes Götzfried, Ruan De Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. 2017. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Trans. Comput.* PP, 99 (2017).

[29] Microsoft. 2019. Open Enclave SDK. online, accessed 2019-05-10: https://openenclave.io/sdk/. (2019).

[30] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar. 2019. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming* 47, 4 (2019), 538–570.

[31] J. Noorman, J. Tobias Mühlberg, and F. Piessens. 2017. Authentic execution of distributed event-driven applications with a small TCB. In *STM*. 55–71.

[32] J. Noorman, J. Van Bulck, J. Tobias Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. 2017. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)* 20, 3 (2017), 7:1–7:33.

[33] OP-TEE. 2019. Security Advisories. online, accessed 2019-08-29: https://www.op-tee.org/security-advisories. (2019).

[34] S. Pinto and N. Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 130.

[35] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *arXiv preprint arXiv:1908.11143* (2019).

[36] Graphene Project. 2019. Graphene: a Library OS for Unmodified Applications. online, accessed 2019-08-30: https://grapheneproject.io/. (2019).

[37] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. 2018. Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features. In *Asia CCS 2018*. 587–600.

[38] M. Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical enclave malware with Intel SGX. In *DIMVA*. 177–196.

[39] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. 2017. Malware guard extension: using SGX to conceal cache attacks. In *DIMVA*. 3–24.

[40] J. Seo, B. Lee, S. Min Kim, M.W Shih, I. Shin, D. Han, and T. Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs.. In *NDSS 2017*.

[41] H. Shacham et al. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).. In *ACM CCS 2007*. 552–561.

[42] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *NDSS 2017*.

[43] C.C Tsai, D. E Porter, and M. Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC*.

[44] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*.

[45] J. Van Bulck, F. Piessens, and R. Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX*. 4:1–4:6.

[46] J. Van Bulck, F. Piessens, and R. Strackx. 2018. Nemesis: Studying microarchitectural timing Leaks in rudimentary CPU interrupt logic. In *ACM CCS 2018*.

[47] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*. 1041–1056.

[48] N. van Ginkel, R. Strackx, T. Mühlberg, and F. Piessens. 2016. Towards safe enclaves. In *Hot Issues in Security Principles and Trust (HotSpot)*. 1–16.

[49] N. van Ginkel, R. Strackx, and F. Piessens. 2017. Automatically generating secure wrappers for SGX enclaves from separation logic specifications. In *Asian Symposium on Programming Languages and Systems*. 105–123.

[50] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*. 440–457.

[51] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *NDSS 2019*.

[52] J. Woodruff, R. NM Watson, D. Chisnall, S. W Moore, J. Anderson, B. Davis, B. Laurie, P. G Neumann, R. Norton, and M. Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA 2014*. 457–468.

[53] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. 2018. Precise and scalable detection of double-fetch bugs in OS kernels. In *IEEE Symposium on Security and Privacy*. 661–678.

[54] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*. 640–656.

A VULNERABLE CODE SAMPLES

A.1 OE legacy ecall dispatcher

The (legacy) `ecall` interface `_handle_call_enclave()` does not validate that `arg_in.args` points outside the enclave. While this pointer is subsequently checked by the `oeedger8r`-generated entry code, an error code is still written to the in-enclave memory location on failure (cf. Listing 2). After our report, the legacy `handle_call_enclave()` dispatcher has been removed completely.

```
1 static oe_result_t _handle_call_enclave(uint64_t arg_in) {
2   oe_call_enclave_args_t args, *args_ptr;
3   ...
4   if (!oe_is_outside_enclave((void*)arg_in,
5                              sizeof(oe_call_enclave_args_t)))
6     OE_RAISE(OE_INVALID_PARAMETER);
7   args_ptr = (oe_call_enclave_args_t*) arg_in;
8   args = *args_ptr;
9   ...
10 ★ func(args.args);
```

```
11   ...
```

Listing 6: https://github.com/Microsoft/OpenEnclave/blob/93ac313a/enclave/core/sgx/calls.c#L216

A.2 OE built-in attestation ecall

Evidently, a check that validates that `arg_in` points outside the enclave was overlooked. We thus can overwrite in-enclave memory through the write to `host_arg->result`. Note that the target buffer has to have a certain size to avoid segfaults in the function `_oe_get_local_report()` that is called within `_handle_get_sgx_report()` (this is because the parameter `oe_get_sgx_report_args_t` is a large struct). Because of that, `_oe_get_local_report()` will very likely fail with the return value `OE_INVALID_PARAMETER` (`0x3`) and overwrite the first four bytes of the memory at `host_arg` with `0x03000000`.

```
1  oe_result_t _handle_get_sgx_report(uint64_t arg_in) {
2    oe_result_t result = OE_UNEXPECTED;
3    oe_get_sgx_report_args_t* host_arg =
4        (oe_get_sgx_report_args_t*)arg_in;
5    oe_get_sgx_report_args_t enc_arg;
6    size_t report_buffer_size = sizeof(sgx_report_t);
7
8    if (host_arg == NULL)
9      OE_RAISE(OE_INVALID_PARAMETER);
10
11   // Validate and copy args to prevent TOCTOU issues.
12 ★ enc_arg = *host_arg;
13
14   OE_CHECK(_oe_get_local_report(NULL, 0,
15     (enc_arg.opt_params_size != 0) ? enc_arg.opt_params : NULL,
16     enc_arg.opt_params_size, (uint8_t*)&enc_arg.sgx_report,
17     &report_buffer_size));
18
19 ★ *host_arg = enc_arg;
20   result = OE_OK;
21 done:
22   if (host_arg)
23 ★   host_arg->result = result;
24   return result;
25 }
```

Listing 7: https://github.com/microsoft/OpenEnclave/blob/93ac313a/enclave/core/sgx/report.c#L388

A.3 Asylo ecall entry point

Asylo's trusted `ecall` dispatcher is declared in Intel SGX-SDK EDL specification as follows: `public int ecall_dispatch_trusted_call(uint64_t selector, [user_check] void *buffer)`. However, in the code below, it becomes apparent that the `[user_check]` argument buffer is never properly validated before being unmarshalled. This issue can most easily be mitigated by properly declaring the argument buffer using `edger8r`'s `[in]` pointer attribute instead of the problematic `[user_check]` attribute. Further, the validation logic at line 16 contains a logic mistake which incorrectly assumes that outside == ¬inside (cf. Section 5.3).

```
1  int ecall_dispatch_trusted_call(uint64_t selector, void *buffer) {
2    return asylo::primitives::asylo_enclave_call(selector, buffer);
3  }
4
5  int asylo_enclave_call(uint64_t selector, void *buffer) {
6    SgxParams *const sgx_params = reinterpret_cast<SgxParams *>(
       buffer);
7
8  ★ const void *input = sgx_params->input;
9  ★ size_t input_size = sgx_params->input_size;
10 ★ sgx_params->input = nullptr;
11 ★ sgx_params->input_size = 0;
12   void *output = nullptr;
13   size_t output_size = 0;
14
15   if (input) {
16 ★   if (TrustedPrimitives::IsTrustedExtent(input, input_size)) {
17       PrimitiveStatus status{error::GoogleError::INVALID_ARGUMENT,
          "input should lie within untrusted memory."};
18       return status.error_code();
19     }
```

Listing 8: https://github.com/google/asylo/blob/e4810bdbac/asylo/platform/primitives/sgx/trusted_sgx.cc#L98

A.4 SGX-LKL SIGILL signal handler exploit

SGX-LKL intercepts the `SIGILL` (undefined instruction) to handle instructions like `rdtsc` inside the enclave. In this case, the host executes `rdtsc` and the result is passed back into the enclave through the enclave's signal handler interface. In case of `SIGILL`, an adversary can change the untrusted `siginfo` argument to point into the enclave, which will then yield the memory contents at that location as the 64-bit result of `rdtsc`, as shown by our PoC. This specific vulnerability can only be exploited if the target in-enclave memory starts with `0x04000000` (*i.e.*, `siginfo->signum==SIGILL`). In addition, the `rdtsc` result needs to be outputted back to the untrusted side (e.g., our PoC simply prints it to the terminal). Note that adversaries can also use the in-enclave signal handler's execution itself as a side-channel. Depending on the contents of the memory pointed to by `siginfo->signum` different code paths are taken, so established side-channel approaches may reconstruct the secret-dependent control through differences in timing [30], page tables [47, 54], or other microarchitectural elements [24, 46].

```
1  void __enclave_signal_handler(gprsgx_t *regs,
2                                enclave_signal_info_t *siginfo) {
3    ...
4    int ret;
5  ★ switch (siginfo->signum) {
6    case SIGSEGV:
7  ★     ret = handle_sigsegv(regs, siginfo->arg);
8        break;
9    case SIGILL:
10 ★     ret = handle_sigill(regs, siginfo->arg);
11       break;
12   default:
13       ret = -1;
14   }
15   ...
```

Listing 9: https://github.com/lsds/sgx-lkl/blob/664eb25a/src/sgx/enclave_signal.c#L17

A.5 Sancus authentic execution stub

Passing a ciphertext pointer argument that points inside the enclave may unintentionally decrypt enclave memory, potentially leading to information disclosure. Interestingly, we observed that untrusted array index arguments were properly sanitized to safeguard against well-understood buffer overflow vulnerabilities.

```
1  void SM_ENTRY __sm_handle_input(uint16_t conn_id,
2                                  const void* payload, size_t len)
3  {
4    if (conn_id >= SM_NUM_INPUTS) return;
5
6    size_t data_len = len - AD_SIZE - SANCUS_TAG_SIZE;
7  ★ uint8_t* cipher = (uint8_t*)payload + AD_SIZE;
8  ★ uint8_t* tag = cipher + data_len;
9
10   uint8_t* input_buffer = alloca(data_len);
```

```
11
12 ★ if (sancus_unwrap_with_key(__sm_io_keys[conn_id],
13                              payload, AD_SIZE, cipher,
14                              data_len, tag, input_buffer))
15   {
16    __sm_input_callbacks[conn_id](input_buffer, data_len);
17   }
18 }
```

Listing 10: https://github.com/sancus-pma/sancus-compiler/blob/5d5cbff/src/stubs/sm_input.c#L7

A.6 OE string ecall edge wrapper

As part of OE's "deep copy" marshalling scheme, the `_handle_call_enclave_function()` from the trusted runtime properly copies the entire marshalled input buffer into the enclave (including the string argument and alleged length which are put into the serialized `input_buffer` by the untrusted runtime). The `oeedger8r` bridge then takes care to redirect all pointers to the marshalled input buffer. However, when doing so the auto-generated `oeedger8r` entry code below does *not* explicitly null-terminate the untrusted string argument. Hence, the trusted user function will incorrectly assume that the string is properly terminated and may perform out-of-bounds memory read/writes beyond the end of the string.

```
1 void ecall_my_ecall(uint8_t* input_buf,
2     size_t input_buf_size, uint8_t* output_buf,
3     size_t output_buf_size, size_t* output_bytes_written)
4 {
5   oe_result_t _result = OE_FAILURE;
6   /* NOTE: output buf code removed for sake of space */
7   my_ecall_args_t* pargs_in =(my_ecall_args_t*) input_buf;
8   size_t input_buf_offset = 0;
9
10  /* Make sure buffers lie within the enclave */
11  OE_ADD_SIZE(input_buf_offset, sizeof(*pargs_in));
12  if (!input_buf || !oe_is_within_enclave(input_buf,
        input_buf_size))
13     goto done;
14
15  /* OE_SET_IN_POINTER(s, s_len * sizeof(char)) */
16  if (pargs_in->s) {
17 ★   *(uint8_t**)&pargs_in->s = input_buf + input_buf_offset;
18     OE_ADD_SIZE(input_buf_offset, (size_t)(s_len*sizeof(char)));
19     if (input_buf_offset > input_buf_size) {
20         _result = OE_BUFFER_TOO_SMALL;
21         goto done;
22     }
23  }
24  oe_lfence();              /* lfence after checks */
25 ★ my_ecall(pargs_in->s);    /* Call user function */
26  ...
27 }
```

Listing 11: Proxy function generated by `oeedger8r` for the EDL specification: `public void my_ecall([in,string] char *s);`.

A.7 Keystone integer overflow

We discovered a potential vulnerability that originates from an integer overflow in the `detect_region_overlap()` function which is used during the process of creating an enclave. Evidently, there is no check to guarantee that the integer additions do not overflow. Suppose that `epm_base=0x82800000` and `epm_size=100000`. If one passes `addr=0x1` and `size=0xffffffffffffffff`, there is an overlap between both regions. However, when these values are put into the above condition, this evaluates to "no overlap" (zero). The above issue was not exploitable at the time of discovery: various constraints imposed on the size prevented the exploitation of this issue, but it might have been problematic in the future if the overlap check was used in different parts of the code.

```
1 static int detect_region_overlap(uintptr_t addr, uintptr_t size)
2 {
3   ...
4 ★ region_overlap |= ((uintptr_t) epm_base < addr + size)
5         && ((uintptr_t) epm_base + epm_size > addr);
6   ...
```

Listing 12: https://github.com/keystone-enclave/riscv-pk/blob/e24d47c/sm/pmp.c#L71