SGXJail: Defeating Enclave Malware via Confinement

Samuel Weiser, Luca Mayr, Michael Schwarz, Daniel Gruss
Graz University of Technology

Abstract

Trusted execution environments, such as Intel SGX, allow executing enclaves shielded from the rest of the system. This fosters new application scenarios not only in cloud settings but also for securing various types of end-user applications. However, with these technologies new threats emerged. Due to the strong isolation guarantees of SGX, enclaves can effectively hide malicious payload from antivirus software. Were these scenarios already outlined years ago, we are evidencing functional attacks in the recent past. Unfortunately, no reasonable defense against enclave malware has been proposed.

In this work, we present the first practical defense mechanism protecting against various types of enclave misbehavior. By studying known and future attack vectors we identified the root cause for the enclave malware threat as a too permissive host interface for SGX enclaves, leading to a dangerous asymmetry between enclaves and applications. To overcome this asymmetry, we design SGXJail, an enclave compartmentalization mechanism making use of flexible memory access policies. SGXJail effectively defeats a wide range of enclave malware threats while at the same time being compatible with existing enclave infrastructure. Our proof-of-concept software implementation confirms the efficiency of SGXJail on commodity systems. We furthermore present slight extensions to the SGX specification, which allow for even more efficient enclave compartmentalization by leveraging Intel memory protection keys. Apart from defeating enclave malware, SGX-Jail enables new use cases beyond the original SGX threat model. We envision SGXJail not only for site isolation in modern browsers, *i.e.*, confining different browser tabs but also for third-party plugin or library management.

1 Introduction

Isolation is an essential element of modern computer systems. Traditionally, the operating system was responsible for isolating processes. With the emergence of various novel use cases, further isolation became necessary. For instance, executing untrusted JavaScript code demands isolation from the browser via sandboxing. Also, mutually untrusted services in the cloud, e.g., from different tenants, run in different containers or virtual machines. In any case, it is still necessary to trust system administrators, operating systems, and hypervisors.

Intel addressed this problem with SGX. Intel SGX can be used to isolate software modules via hardware protected enclaves from a compromised or malicious administrator, operating system, or hypervisor. The trust anchor in SGX is only the processor. Even if any other system part is manipulated or compromised, SGX maintains its security guarantees. This enables new use cases, such as trusted cloud computing, where tenants do not only distrust the other tenants, but also the cloud provider and its hardware and software infrastructure [3, 17, 50]. A similar distrust also exists when protecting copyrighted material [2] or cryptographic or security-critical secrets [30, 35, 42] on a compromised user PC or server.

While isolation techniques such as SGX can be an excellent tool for security, they can also be misused for hiding malicious activity inside an enclave. In the recent past, we have seen not only enclave malware exploiting side channels [54] but also enclave ransomware and shellcode [38], however, with the help of a colluding host application. Recent research showed that enclaves can effectively hijack and impersonate any benign host application [53], opening up enclaves for various types of userspace malware. This confirms what researchers already suspected years ago [13, 16, 48]. Having witnessed first proof-of-concept attacks [38, 53], we can expect that more sophisticated and real-world attacks will appear in the future. Hence, it is necessary to providently explore the defense space, before real-world attacks are discovered.

Unfortunately, little is known about how to address this emerging threat properly. While conventional programs can be scanned for misbehavior by anti-virus technology, SGX is a complete game changer when it comes to enclave analysis. On the one hand, SGX prevents runtime inspection of enclaves. On the other hand, SGX allows lazy loading of malicious enclave content at runtime. Thus, malware infection can be completely decoupled from enclave distribution and installation, which renders all static analysis techniques on the enclave void. In other words, SGX is a viable alternative to malware obfuscation and analysis evasion techniques. If Intel chose to allow certified anti-virus software to inspect enclaves, this would undermine essential security guarantees and is in fundamental conflict with the very goal SGX has [48]. Others proposed to detect enclave malware via their I/O behavior [13, 16], which is prone to both false positives and false negatives. Moreover, tracing and analyzing all enclave I/O behavior is believed infeasible in practice [38]. Others proposed to embed malware analysis code within the enclave itself, which raises several questions regarding its practicality [13]. Consequently,

> "[...] the release and adoption of SGX-protected enclaves is likely to require a completely new approach to protecting our machines from the very malware SGX was designed to prevent." [16]

So far, no practical defense against enclave malware exists.

In this work, we propose the first practical defense mechanism against enclave malware. To do so, we analyze enclave primitives and their resulting attack vectors and identify the root cause for the enclave malware threat as a too permissive feature set available to enclaves, forcing applications to trust any enclaves they host blindly. Consequently, a proper defense mechanism should give applications means to confine enclave operation to a clearly specified interface. To that end, we propose SGXJail, a lightweight yet effective measure to establish mutual distrust between enclaves and its host application. SGXJail does so by confining enclave operation to a clearly defined set of memory pages. This mitigates entire classes of runtime attacks (ROP, JOP, DOP, etc.) from the enclave to the host and enables reasoning about enclave misbehavior purely based on the legitimate communication interface. We instantiate SGXJail using process sandboxing and syscall filters and demonstrate its efficiency. Furthermore, we propose HSGXJail, a minimal hardware extension to the SGX specification making use of Intel memory protection keys to confine enclave execution, which is even more efficient. (H)SGXJail is opt-in, works on unmodified enclaves and can be easily integrated with the SGX software development kit[1].

With SGXJail, we expand possible SGX use cases beyond isolated execution. We envision modern software which is additionally hardened using SGXJail against potentially malicious or misbehaving third-party code. This is, for example, vital for all software enabling third-party plugins and add-ons, such as browsers, mail clients, or password managers.

Contributions. We summarize our contributions as follows.

1. We systematically break down the enclave malware threat and identify a number of enclave malware primitives.
2. We devise SGXJail, the first practical defense against enclave malware.
3. We implement and evaluate SGXJail in software.
4. We propose highly efficient HSGXJail via minimal hardware changes.

[1]The code is available at https://github.com/IAIK/sgxjail

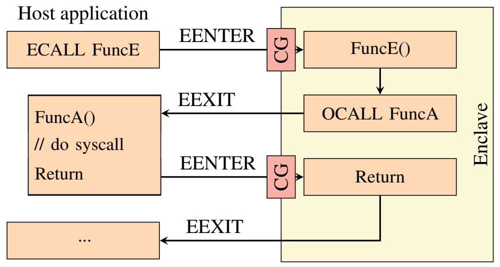

Figure 1: SGX enclaves are tightly integrated in a host application. The application can invoke the enclave via ECALLs while the enclave can perform OCALLs. Enclaves can only be entered via the `EENTER` instruction at certain call gates (CG) and can only be left via `EEXIT`.

The rest of the paper is organized as follows. Section 2 provides background information. Section 3 describes the threat model. Section 4 analyzes various enclave primitives and attack vectors. Section 5 presents (H)SGXJail. Section 6 discusses related work. We summarize our discussion of enclave malware in Section 7 and conclude in Section 8.

2 Background

In this section, we provide background on Intel SGX as well as runtime attacks.

2.1 Intel SGX

Intel Software Guard Extensions (SGX) are an instruction-set extension introduced with the Skylake microarchitecture [26]. SGX allows creating so-called *enclaves* running trusted code isolated from the remaining system.

Enclaves are hosted by an ordinary application process. Although the enclave and the host application reside in the same virtual address space, the address range of the enclave is inaccessible to the host application. Only the enclave itself can access its memory while the hardware prevents any other access to enclave memory. However, the enclave can access the entire virtual address space of the host application, allowing to share data between the enclave and the host application. This asymmetry in access permissions fits the original threat model of SGX but gives rise to enclave malware.

The host application is responsible for loading the enclave into the current address space and providing an interface through which the enclave communicates with the outer world. The CPU measures the loading process to ensure the integrity of the loaded enclave. The enclave is only executed if the resulting measurement matches a developer-specified value.

Figure 1 shows the process of invoking an enclave. The enclave defines secure functions denoted as ECALLs, which the application can call with the `EENTER` instruction. Call gates (CG) restrict enclave invocation to valid entry points.

Enclaves can request OS services such as syscalls via so-called OCALLs. To leave the enclave, an enclave can issue the `EEXIT` instruction. Enclave developers need to specify the ECALL/OCALL interface via a so-called Enclave Definition Language (EDL). Each enclave is shipped with its own EDL file. An EDL file roughly contains function signatures of the enclave's ECALLs and OCALLs, augmented with additional security attributes (e.g., `in`, `out`). Intel provides developers with an SDK [25] that automatically generates glue code from the EDL file with appropriate parameter validation and buffer copying inside the enclave.

SGX assumes that all non-enclave code (*i.e.*, operating system and host application) is untrusted. SGX provides no means to protect applications from misbehaving enclaves. Instead, an enclave can access all application memory and divert control flow to arbitrary application code via `EEXIT`.

2.2 Runtime Attacks

While it is typically not possible to directly inject or modify code at runtime, e.g., via a buffer overflow, an attacker can often manipulate control data and thus change the control flow of an application. By overwriting a code pointer, an attacker can divert the control flow to existing code snippets, resulting in so-called control-flow hijacking attacks. One of the most generic and powerful attacks is return-oriented programming (ROP) [55] which overwrites return addresses to create arbitrary attack payloads. Similar attacks exist for overwriting function pointers [5, 9, 10, 18, 32, 51] or signal handlers [6].

Some widely deployed techniques against code-reuse attacks are address-space layout randomization (ASLR) [45], stack canaries [14, 46] and shadow stacks [12]. While stronger control-flow integrity (CFI) [1, 31] can eradicate control-flow attacks, they leave data-only attacks [8, 28] unaddressed.

3 Threat Model

In this section, we first outline various application scenarios of SGX and argue why the original SGX threat model does not properly address enclave misbehavior. We then present our extended SGX threat model addressing enclave malware.

Scenario A. In the near future, SGX technology will likely permeate consumer systems and create diverse and many-faceted trust relations. Multiple independent software vendors (ISV) can use SGX for mutually protecting their proprietary library code (e.g., multimedia codecs, classification algorithms) or sensitive customer data (e.g., user passwords, encryption keys or bitcoin wallets) inside third-party enclaves. Applications can embed such third-party enclaves to leverage their functionality.

In this scenario, an attacker develops innocent-looking enclave malware (e.g., disguised as browser plugins) and distributes it as a third-party enclave via existing software stores or repositories. A user installs those third-party enclaves alongside other applications. The attacker defers installation of malicious payload to runtime via a generic loader [48]. Hence, neither the maintainers of software repositories nor the user can detect this malware before it is actually triggered.

This might not only be invasive malware like ransomware, bots, or rootkits. A malicious enclave can also stealthily collect data about the user and host system without the user knowing, and with plausible deniability for the developer. An enclave developer can then monetize this data, e.g., by selling it to advertising agencies.

Scenario B. As more software is moved into enclaves, chances increase for exploitable vulnerabilities within enclave code. Enclaves are equipped with increasingly complex software, such as fully-fledged TLS stacks [24]. Thus, it is just a matter of time for bugs in the trusted code of enclaves, enabling well-known memory corruption attacks [56] inside enclaves. In fact, it has already been shown that enclaves are prone to such attacks [33, 52, 60]. This can be used to infiltrate trusted enclaves with a malicious payload.

A Holistic Threat Model. The original threat model of Intel SGX considers all non-enclave code as untrusted, including application code hosting enclaves (cf. Section 2). This model might be well-suited from an enclave's perspective. However, it does not fit more advanced application scenarios outlined before, leaving applications completely unprotected against misbehaving third-party enclaves. This creates a dangerous asymmetry, as also outlined by Schwarz et al. [53].

In this work, we introduce a more holistic threat model which does not violate the original threat model of SGX but augments it to explicitly address misbehaving enclaves. We consider a commodity system running software from various independent software vendors. On the one hand, third-party library vendors protect their secret data (e.g., cryptographic keys or intellectual property) inside enclaves. On the other hand, application developers include third-party enclaves in their applications for implementing various tasks. However, they want some form of assurance that third-party enclaves are well-behaving, for the reasons outlined before. While from an enclave vendor's perspective SGX provides strong protection against other enclaves as well as compromised systems, application developers have no means to assure themselves of proper behavior of (third-party) enclaves they use.

From a user's perspective, the computer (including the operating system and certain applications) are trusted. A mechanism is needed to protect applications (and, subsequently the computer) from potential enclave misbehavior, even if such enclaves are fully controlled by a dedicated attacker (e.g., enclave malware). In particular, an application needs protection against any inspection or alteration of its state (memory, CPU registers) by enclaves, apart from what it is exposing to the enclave via the ECALL/OCALL interface. SGXJail does not prevent API attacks, exploiting too permissive OCALLs or badly designed interfaces (e.g., avoiding Iago attacks and confused deputy attacks), which is a separate, yet important

Table 1: Enclave primitives leading to various attack vectors on the host application.

Requirement / Attack	Arbitrary Read	Arbitrary Write	Arbitrary EEXIT
Information disclosure	✓	✗	✗
Control-flow attacks	(✓)	✓	(✓)
Data-only attacks	(✓)	✓	✗

line of research, as we discuss later. Also, this work does not focus on microarchitectural side channels, although SGXJail prevents certain classes of side-channel attacks. Finally, the CPU hardware is considered trusted.

4 Analyzing the Enclave Malware Threat

In this section, we analyze enclave primitives leading to different attack vectors violating memory safety of the application. This helps us design a proper defense mechanism and solve the enclave malware threat at the level of memory safety, leaving only high-level API attacks as a resort for the attacker, as discussed at the end of this section.

4.1 Enclave Primitives

Intel SGX entrusts enclaves with powerful primitives leading to different attacks violating memory safety, as depicted in Table 1. We outline these primitives in the following.

Arbitrary read. An enclave can read arbitrary memory of the host application. This is intended for exchanging data between enclave and host. Furthermore, an enclave can use hardware transactions to suppress exceptions stemming from reading inaccessible memory [53], giving a powerful fault-resistant arbitrary read primitive.

Arbitrary write. An enclave can write arbitrary writable host memory, which is intended for data exchange. Furthermore, it can use hardware transactions to suppress exceptions while writing inaccessible or non-writable memory [53], yielding a fault-resistant arbitrary write primitive.

Arbitrary EEXIT. An enclave can choose the precise code location in the application where execution shall continue after leaving enclave execution via the `EEXIT` instruction. Moreover, the enclave has control over many CPU registers immediately after an `EEXIT`, in particular the stack pointer, which gives enclaves the possibility to configure the application's CPU state before resuming application execution.

4.2 Attack Vectors

Given the above primitives, a malicious enclave can mount a broad range of attacks violating memory safety of the host application. In the following, we cluster them into information disclosure, control-flow attacks as well as data-only attacks and give representative examples of these attacks. A detailed overview of attacks violating memory safety was presented by Szekeres et al. [56].

4.2.1 Information disclosure

A malicious enclave can use the arbitrary read primitive to exfiltrate sensitive user data like cryptographic keys or passwords from the host application. Even if the application contains no such user secrets, an enclave can disclose other sensitive information, e.g., as used in various runtime protection mechanisms. For example, an enclave can derandomize application protection schemes like ASLR [45], stack canaries [14], code randomization [44] or randomization-based control-flow integrity schemes [31, 39]. The enclave can furthermore disclose the host application's codebase and, subsequently, generate targeted exploitation payload like ROP chains on the fly. Thus, information disclosure is a powerful tool often used for subsequent exploitation.

4.2.2 Control-flow attacks

A malicious enclave can deliberately tamper with the application's control flow in several ways. For example, it can directly corrupt code pointers, use rogue EEXITs and bypass various mitigation mechanisms.

Code pointer corruption. An enclave can manipulate an arbitrary code pointer of the host using the write primitive. This can be, e.g., return addresses on the stack or virtual function pointers on the heap. As soon as the application fetches a corrupted code pointer, execution is diverted to an attacker-chosen address. By carefully crafting a so-called ROP chain (cf. Section 2) and diverting execution to it, the attacker can gain arbitrary code execution with the privileges of the application, allowing to execute arbitrary syscalls in lieu of the application. To prepare a ROP chain, the enclave scans the host application for ROP gadgets using the arbitrary read primitive and writes the corresponding addresses on a fake stack using the arbitrary write primitive [53].

An enclave is by no means restricted to ROP attacks only. Similar to ROP, it can craft jump-oriented programming (JOP) attacks, loop-oriented programming (LOP) attacks, or call-oriented programming (COP) by overwriting indirect function pointers [5, 9, 10, 18, 32]. COOP attacks are also possible by overwriting virtual function pointers in C++ applications [51] or SROP attacks [6], faking a signal handler.

Rogue EEXIT. A malicious enclave can also mount control-flow attacks without corrupting a single code pointer. By using the arbitrary EEXIT primitive, the enclave can directly corrupt the CPU state. For example, it can manipulate the stack-pointer register to point to an attacker-crafted ROP chain. By doing an `EEXIT` instruction towards an arbitrary `ret` instruction of the host, the enclave can immediately trigger the ROP chain, leading to the same implications as for ROP.

Bypassing Defenses. Several defense mechanisms seek to protect the application's control flow. Stack canaries [14] protect against linear buffer overflows overwriting return addresses on the stack. ASLR [45] hides code addresses via randomization, while others randomize code itself [44], both making the generation of ROP gadgets hard. More elaborate mechanisms enforce control-flow integrity (CFI), arguably at different granularity. CPI [31] hides code pointers in a shadow stack[2] while CCFI [39] encrypts code pointers. As these mechanisms rely on randomization, they can be easily broken by the enclave via information disclosure. If CFI metadata is involved, it can be easily corrupted using the write primitive. Stronger hardware-enforced CFI schemes like CET [27] are still unavailable on modern x86 CPUs, and it is unclear to what extent they consider rogue EEXIT attacks.

4.2.3 Data-only attacks

Apart from control-flow attacks, enclaves can corrupt application data other than code pointers or CFI metadata. For example, they can corrupt loop counters, function arguments or syscall arguments [8, 28] using the arbitrary read/write primitives. Typically, data-only attacks are much more restricted than control-flow attacks. For example, they can only reuse code reachable in the normal control flow. Yet, data-only attacks are agnostic to CFI protection schemes and can even achieve Turing-complete computation in many cases by chaining together valid execution paths [28].

4.3 API attacks

The previous attack vectors all violate memory safety of the application by reading, writing and executing application memory in an illegitimate way. It is clear that defeating these attacks is paramount to protecting an application from misbehaving enclaves. Only with such protections in place, it makes sense to reason about the application's security on the API level. Obviously, SGXJail does not defend against too permissive OCALLs, e.g., giving an enclave the ability to access arbitrary files. Yet, we need to ask to what extent an enclave can attack its host application purely via the ECALL/OCALL interface, that is, without relying on the above SGX attack primitives. For example, an enclave can seek to attack the application by crafting invalid API calls or returning malformed data. For a successful attack, either the API itself needs to be flawed, or the underlying implementation misses important validation steps (e.g., confused deputy attacks [22] and Iago attacks [11]). Since such API-based attacks are highly application specific, they cannot be addressed by a generic defense mechanism anticipated in this work. We discuss proper mitigation strategies in Section 7. Also, we do not address misuse of computational power (e.g., for cryptocurrency mining).

[2]This corresponds to the weaker randomization scheme since the stronger segment-based isolation is unavailable for 64-bit execution mode.

5 SGXJail

In this section, we present SGXJail, a novel mechanism to protect host applications from untrusted (third-party) enclaves. SGXJail defeats entire classes of attacks by prohibiting enclave primitives outlined in Section 4 at the discretion of the host application. SGXJail can be implemented purely in user space and relies on process isolation and syscall filters, similar to other sandboxing techniques like Docker [40]. We evaluate SGXJail under different workloads to demonstrate its efficiency. Finally, we show how SGXJail can also be implemented via minimal changes to the SGX specifications and corresponding hardware, which we call HSGXJail.

5.1 SGXJail via Software Confinement

SGXJail defeats enclave malware by breaking all three enclave primitives described in Section 4.1. SGXJail does so by confining enclave operation to a strict set of memory pages.

Figure 2 illustrates the basic idea of SGXJail. To break the arbitrary read and write primitives, we rely on the operating system's ability to isolate processes.[3] Namely, we run potentially malicious or misbehaving enclaves in a separate *sandbox process* which does not have access to the host application's memory. To still allow benign ECALL/OCALL interaction, we establish shared memory between the sandbox process and the host application to implement a form of inter-process communication.

Even with the above process isolation in place, a malicious enclave can perform an attack on the control flow of the sandbox process to issue arbitrary syscalls on behalf of the sandbox process. Such an attack can either be a rogue EEXIT attack, or a code-reuse attack (e.g., ROP) through manipulating the stack [53]. Breaking the primitives that allow an attacker to change the control flow is not trivial. EEXIT can jump to any executable page, and the target address cannot be restricted. Similarly, if the enclave rewrites the saved return address on the stack, the sandbox process cannot detect this modification. A possible–but rather expensive solution–is to mark all executable pages of the sandbox process (except for trampoline code) as non-executable before entering the enclave. When leaving the enclave, the sandbox immediately traps to the kernel, which can then assess the legitimacy of the address at which the sandbox process should resume and remap the pages as executable. However, this requires frequent and expensive page remapping by updating a majority of the page tables of a process. Instead of trying to prevent an attack from hijacking the control flow in the sandbox process, we confine the damage of such a hijacked control flow. In particular, we restrict the syscall interface of the sandbox process by using seccomp syscall filters [36] to whitelist only abso-

[3]Conforming with Intel's and our extended SGX threat model, software-based side-channel attacks circumventing such isolation, e.g., Meltdown [37] or Rowhammer [29], are out of scope.

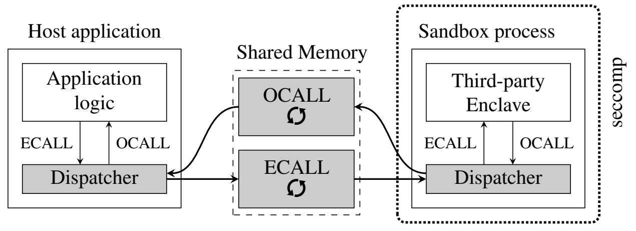

Figure 2: With SGXJail, the enclave is isolated within a separate sandbox process and can communicate with the host application only via shared memory. Also, the enclave is confined using seccomp filters.

lutely necessary syscalls. Even if a malicious enclave gains arbitrary code execution inside the sandbox process, it can no longer perform malicious actions. In contrast to sandboxing techniques like Docker isolating the entire system (e.g., via cgroups), we only need to restrict a single user process for which syscall filters are the appropriate choice.

Life Cycle. A complete SGXJail life cycle works as follows. First, SGXJail creates a new process, the *sandbox process*. The third-party enclave is then loaded within this sandbox process. Moreover, SGXJail creates a shared memory between host application and sandbox process and installs dispatchers for routing all ECALLs and OCALLs through this shared memory. Afterwards, SGXJail activates seccomp filters to restrict the syscalls of the sandbox process to an absolute minimum. Only syscalls required for the communication between application and sandbox process, as well as syscalls required to terminate the sandbox process, are whitelisted. After the initialization, the application can issue ECALLs and receive OCALLs, as follows. The application dispatcher automatically encapsulates ECALLs into messages and transfers them via the shared memory to the sandbox process. ECALL function arguments are copied from the host application to the shared memory. The sandbox process dispatcher listens for incoming messages, decapsulates arriving messages and performs the actual ECALL towards the enclave. Results are returned back to the application, again via message passing over shared memory. The application dispatcher finally copies ECALL results from the shared memory to application memory and hands over to the application. In the same way, OCALLs are routed from the sandbox process through the shared memory to the application host and vice versa. Upon termination of the application, the sandbox process is simply destroyed. Multiple enclaves are isolated via separate sandbox processes with individual shared memory segments.

Compatibility. SGXJail is a transparent enclave confinement mechanism. It does not require any changes to third-party enclaves themselves, *i.e.*, it is binary-compatible with existing enclaves and their existing cryptographic signatures. Also, no enclave source code needs to be available. Instead, SGXJail is tightly integrated within the SGX SDK [25]. All glue code for dispatching and redirecting ECALLs and OCALLs via shared memory is automatically generated from an enclave's EDL file [25] which needs to be shipped together alongside each pre-compiled third-party enclave. Also, code for instantiating the sandbox process, the shared memory and activating seccomp filters is provided by SGXJail. For SGXJail, only the untrusted application code has to be recompiled under the SGXJail toolchain.

The installation of seccomp filters is independent of the enclave itself. Since enclaves are not entitled to issue syscalls, the selection of proper syscall filters solely depends on SGX-Jail and does not affect compatibility with enclaves.

SGXJail enforces benign enclave communication to follow the ECALL/OCALL interface specified in the enclave's EDL file. An enclave implementing other communication methods (e.g., by directly accessing host memory) breaks as soon as SGXJail is active. This is intentional, as enclave developers are strongly encouraged to clearly define the enclave's API via ECALLs and OCALLs. In particular, SGXJail breaks unsafe usage of ECALLs and OCALLs where enclave and host application exchange and dereference raw, unchecked pointers rather than buffered data. For example, if one marks an ECALL function parameter with the so-called `user_check` attribute within the EDL file [25], the SDK passes this function parameter without further checking and copying into the enclave. A quick code inspection revealed usage of `user_check` in some Intel architectural enclaves and remote attestation code, all for performance reasons. They could be updated to avoid `user_check` at the cost of slight performance loss. To yet support `user_check`, one would need to manually share (*i.e.*, map) host memory with the sandbox process to which an enclave shall have unrestricted access. Also, host application pointers passed to the enclave need to be translated to the sandbox process due to ASLR. SGXJail could provide simple helper functions for sharing host memory and translating pointers.

5.2 Implementation Details

For generating dispatcher code, we extend the `edger8r` tool [25] accordingly. The sandbox dispatchers are generated in the files `Enclave_us.c|h`, while the application dispatchers are located in `Enclave_u.c|h`. An enclave always copies arguments to enclave memory before processing it. Similarly, our dispatcher code copies arguments to application memory before invoking an OCALL. This prevents TOCTOU vulnerabilities such as double-fetch bugs [58] by design.

ECALLs and OCALLs are routed between application and sandbox process via two distinct shared memory regions, one for each direction. The dispatchers synchronize ECALL/OCALL interaction via shared semaphores. This has the advantage that processes (application and sandbox) are consuming no CPU time while waiting for the other communication partner. For receiving OCALLs, the application installs a separate listener thread that only gets active upon incoming OCALLs.

Selection of appropriate syscall filters is crucial for the security of SGXJail, as a malicious enclave can directly exploit a lax configuration (e.g., via rogue EEXIT attacks). It is favorable to restrict both the number of syscalls as well as their complexity to reduce the attack surface given by the whitelisted syscalls. This also has an impact on the type of inter-process communication between sandbox and application process. By choosing shared memory as communication channel, we do not require any syscall for the actual communication, and only one syscall (`futex`) for synchronization. In summary, we configure seccomp [36] to only allow the syscalls `futex` necessary for semaphores as well as `exit_group` for terminating the sandbox process. Thus, the shared memory approach results in only one whitelisted syscall in addition to the required `exit_group` syscall. Unless the implementation of these two syscalls is buggy, they cannot cause a security violation when issued by a malicious enclave.

The SGX SDK passes OCALL function arguments from the enclave to the application via the application's stack. The enclave knows the application's stack location via the stack pointer (`RSP` register), which is preserved by the `EENTER` instruction. Hence, it can allocate a stack frame on the host stack via a function called `sgx_ocalloc` and store any outgoing OCALL arguments there. One can leverage this mechanism for reducing SGXJail overhead, as follows. Currently, when doing an OCALL, our sandbox dispatchers copy OCALL arguments from the sandbox to the shared memory. By modifying `RSP` immediately before an `EENTER` to point to the shared memory, one can instruct the enclave to write OCALL arguments directly to the shared memory instead of the sandbox application's stack. When the enclave `EEXIT`s, one can simply restore the original sandbox stack (namely, `RSP`).

In our current implementation, the size of the shared memory is hard-coded to three pages for each direction. For ECALL/OCALL arguments exceeding the shared memory, one can dynamically resize the shared memory on demand. Although multithreaded enclaves are currently not supported by our prototype implementation, support can be easily added. This is done by installing separate semaphores and shared buffers for all enclave threads, which are enumerated in a public enclave XML configuration file. Also, support for nested calls (OCALLs issuing ECALLs) can be added by adapting the synchronization mechanism appropriately.

An interesting question arises whether SGXJail should be integrated with the SGX SDK in a way that does not demand recompilation of the application. Thus, system administrators can globally enforce SGXJail by just installing corresponding shared libraries. Since the enclave's EDL file is public anyway and will be distributed alongside third-party enclaves, the generation of dispatcher code is straight forward. Moreover, one would need to hook the enclave API of the unmodified application binary and inject dispatcher code, which can be done by preloading SGX SDK libraries (in particular, `sgx_urts.so`).

Table 2: ECALL and OCALL latency in CPU cycles of SGXJail compared to the unprotected Vanilla version. The standard deviation is shown in braces.

Latency	ECALL	OCALL
Vanilla	15 624 ($\pm$ 301)	13 438 ($\pm$ 1046)
SGXJail	22 094 ($\pm$ 814)	19 515 ($\pm$ 1360)

5.3 Evaluation

SGXJail does not affect runtime performance of host applications or enclaves in isolation. That is, as long as no interaction between enclave and application takes place, they can run without performance loss. The only performance overhead occurs when doing ECALLs and OCALLs due to the message passing via shared memory and the necessary synchronization between application and sandbox process. To evaluate this effect, we first present microbenchmarks for bare metal ECALL and OCALL latency, which are followed by macrobenchmarks on more representative workloads.

Test Setup. All evaluations are done on a commodity notebook featuring an Intel i5-6200U CPU, a Samsung SM951 SSD and running Ubuntu 16.04 Desktop and SGX SDK version 2.4. For the benchmarks, we disabled the screen as well as network interfaces to reduce noise from screen redrawing or external interrupts. Also, we fixed the CPU frequency to its maximum (2.3 GHz) and pinned the benchmark to a single core. The benchmarks include a warm-up phase.

Microbenchmarks. To measure the ECALL latency, we implemented a simple ECALL and measured its execution time from within the host application. That is, the ECALL latency includes EENTER, EEXIT, all glue code for the enclave and the host, as well as context switching and synchronization between application and sandbox for SGXJail. To measure the OCALL latency, we, in addition, perform one simple OCALL from within the ECALL and subtract the ECALL latency. We repeated the measurement 500 times. The resulting latencies are shown in Table 2. The raw ECALL latency increases from $15.6 \cdot 10^3$ cycles to $22.1 \cdot 10^3$ cycles while the OCALL latency increases from $13.4 \cdot 10^3$ cycles to $19.5 \cdot 10^3$ cycles. Hence, the absolute latency remains small. Since many practical usage scenarios of SGX involve somewhat complex computations inside the enclave, the actual runtime overhead is much lower than the pure ECALL/OCALL overhead.

Macrobenchmarks. Quantifying performance of enclaves is highly application specific. Unfortunately, enclaves are not widely deployed yet and standardized benchmarking suites are unavailable to the best of our knowledge. A common approach is to port existing programs to an enclave [61]. While this sounds appealing, it tends to introduce many unnecessary OCALLs to the standard library which well-designed enclaves would not perform, e.g., the `getpid` syscall in openVPN [61].

Instead, we quantify the performance of SGXJail as follows. First, we benchmark a synthetic workload under different OCALL frequencies. The results of this benchmark are generic and can be applied to any enclave for which the OCALL frequency can be determined. Second, we benchmark storage of sensitive enclave data to disk via the Intel protected filesystem (PFS). The PFS is integrated within the SGX SDK and is likely to be used by a vast number of enclaves.

For our first benchmark, we observe that an enclave typically issues OCALLs to perform syscalls, e.g., writing to files. Our benchmarked OCALL performs a `close` syscall on an invalid file descriptor. Such a fast syscall gives an upper bound on the performance overhead since longer syscalls decrease the influence of the OCALL overhead. We repeated each measurement 100 times. The OCALL-to-enclave ratio (w.r.t. their runtime) as well as the overhead of SGXJail compared to unprotected Vanilla applications is given in Figure 3, whereas the simple standard deviation is shown as the area under the curves. We execute a fixed baseline workload inside the enclave, which corresponds to 2201.44 ($\pm$ 25.67) $\cdot 10^6$ cycles, or 0.96 ($\pm$ 0.011) s on our 2.3 GHz CPU. As this workload runs within the enclave, we quantify it as enclave seconds, or Esec. While we keep the enclave workload constant, we issue OCALLs at different frequencies and measure the additional OCALL work. This is shown as ratio on the left axis of Figure 3 and allows us to decouple the OCALL overhead from the OCALL frequency, which we quantify as OCALLs/Esec.

One can see that the overhead of SGXJail is virtually nonexistent for low-frequency OCALLs, meaning that pure enclave execution is not impeded by SGXJail at all. Even for 10 000 OCALLs/Esec the overhead is below 3% and for a large number of 50 000 OCALLs/Esec the overhead is only around 11%. To put these numbers into perspective, Netflix observed a maximum of 50 000 OCALLs/s across their systems [20]. For even higher OCALL frequencies the OCALL workload starts to exceed the enclave workload in the vanilla version already. With SGXJail, enclaves can issue up to 113 000 OCALLs/Esec before OCALL processing exceeds actual enclave computations (ratio=1). For unprotected apps this point is reached for 164 000 OCALLs/Esec. Such situations should be dealt with in practice by redesigning the enclave API and reducing or removing unnecessary OCALLs. Yet, SGXJail only introduces around 20% overhead even in this extreme case.

Our first benchmark measures the raw OCALL performance. However, this does not reflect the performance of copying OCALL arguments between enclave and application. To evaluate the maximum overhead of a real-world scenario, we benchmark an enclave which only accesses files via the Intel protected file system (PFS) library. PFS is shipped with the SGX SDK and is intended for sealing sensitive enclave data on the host file system for persisting state across reboots. To resemble a worst-case scenario of PFS, we implement and benchmark a single ECALL which opens a new file (`sgx_fopen_auto_key`), writes a fixed-size buffer (`sgx_fwrite`), and immediately closes the file again

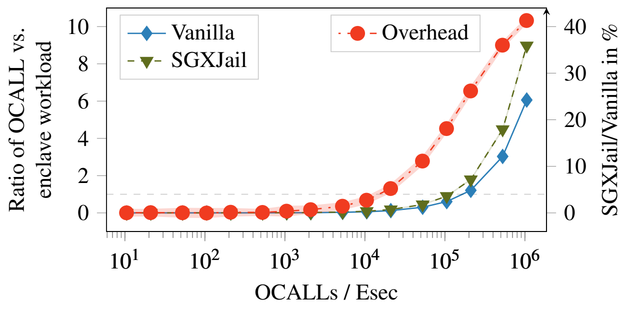

Figure 3: Benchmark on unprotected (Vanilla) and hardened (SGXJail) applications, plotted over different numbers of OCALLs per enclave second (Esec).

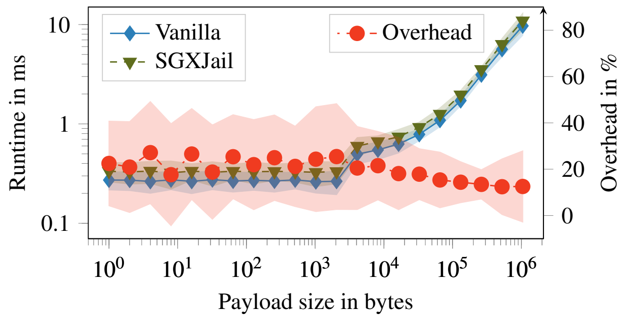

Figure 4: PFS runtime of SGXJail compared to unprotected Vanilla enclaves for different payload sizes.

(`sgx_fclose`). We repeat the measurements 200 times. After each run, we delete the file and synchronize the file system to reliably capture the overhead of PFS. Figure 4 shows the PFS performance for different payload sizes up to 1MB. The runtime includes enclave as well as OCALL computation. The simple standard deviation is shown as area around the curves.

The maximum overhead for protecting PFS with SGXJail is roughly around 20%. There is almost constant runtime up to 2 kB payloads for SGXJail and the unprotected vanilla enclave with a sudden increase at 4 kB payloads. The reason is that the PFS library caches smaller chunks of data and defers actual file writing to closing the file with `sgx_fclose` with 8 OCALLs in total. When exceeding the internal buffer of 3072 bytes, the PFS library flushes data to the file system using 7 more OCALLs, resulting in the sudden increase of the absolute runtimes for SGXJail and the vanilla enclave.

For larger payloads (4 kB and more), the overall overhead does not increase but falls below 20%. This suggests that argument copying itself is not the bottleneck of PFS. We verified this by manually removing argument copying in the sandbox for the actual file write OCALL. Using 1 MB payloads, the overhead dropped by roughly 3%. Rather than argument copying, the runtime overhead of SGXJail is dominated by the OCALL overhead since the PFS implementation chops

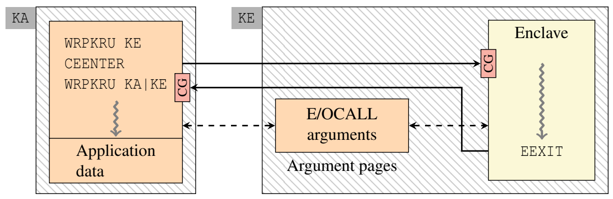

Figure 5: HSGXJail confines the enclave to pages marked with memory protection key `KE`. Thus, the application can protect its pages via a disjoint memory protection key `KA`. ECALL/OCALL interaction is constrained to non-enclave `KE` pages (dashed lines). Moreover, `EEXIT` can only target a single exit point, namely the instruction following a `CEENTER` (a new confined `EENTER` instruction).

larger payloads into a sequence of smaller OCALLs. In fact, for 1 MB payloads we observed 313 OCALLs in total.

We have shown that SGXJail does not impede pure enclave computation (0% overhead). For real-world workloads up to 10 000 OCALLs/Esec, the overhead is below 3% (cf. Figure 3). Even for uncommonly high OCALL frequencies (100 000 OCALLs/Esec), the overhead of SGXJail is still below 20%, whereas plain writing of protected files with high OCALL interaction comes at only 20% overhead. To further improve performance, SGXJail could use HotCalls for faster enclave communication [61]. Alternatively, we propose a lightweight hardware extension (HSGXJail) which provides SGXJail isolation at virtually no overhead.

Memory overhead. SGXJail requires one additional process for the sandbox. As for site isolation in browsers [47], this incurs only a slight (constant) increase in used memory for the sandbox and the shared memory used for communication.

5.4 HSGXJail via Hardware Confinement

In this section, we propose a more efficient defense mechanism via a minimal change to the SGX specification with respect to Intel memory protection keys (MPK), *i.e.*, disallowing one MPK instruction in SGX.

To prevent an enclave from accessing host application memory, we propose a stricter page access policy. To that end, HSGXJail introduces two extensions: first, data confinement and second, control confinement. First, memory regions that are not supposed to be used by the enclave shall be inaccessible to the enclave. Data confinement limits memory pages an enclave can read or write, thus breaking the arbitrary read and write primitives. Second, `EEXIT` shall be only allowed on well-defined exit points. Control confinement prevents the enclave from misusing `EEXIT` to jump to arbitrary host application code, thus breaking the arbitrary `EEXIT` primitive.

Data Confinement with Intel Memory Protection Keys. The central issue of enclave malware is an asymmetry in the memory access policy, granting enclaves unrestricted access to host-application memory. Data confinement uses a recent protection mechanism called memory protection keys (MPK) [26] to partition virtual memory into enclave-accessible memory and protected application memory. If the enclave attempts to access protected application memory, the CPU raises a page fault. To prevent the enclave from reconfiguring MPK, HSGXJail disallows certain MPK instructions in enclave execution mode. Similar to SGXJail (cf. Section 5.1), we use this mechanism to confine enclave execution to a narrow ECALL/OCALL interface, as shown in Figure 5.

Memory protection keys work as follows: they augment page-based read, write and execute permissions with additional access policies. Each application page can be assigned one particular memory protection key. This protection key is stored directly in the corresponding page table entry (PTE). By assigning different protection keys to different pages, MPK allows to partition virtual memory pages into 16 disjoint protection domains. The PKRU CPU register controls which access policy is applied to those protection domains. For each of the 16 protection keys, PKRU allows to selectively disable write and read access for the current execution thread. The PKRU register can be updated via the unprivileged `WRPKRU` instruction, enabling frequent switching of protection domains within the application. Since each CPU thread maintains its own local PKRU register, MPK supports multithreading.

For HSGXJail, we partition the application into protection key `KA` comprising all application pages and `KE`, covering enclave memory as well as argument pages, as shown in Figure 5. Immediately before entering an enclave, the application configures PKRU to confine memory accesses to the enclave only (`WRPKRU KE`). During enclave operation, the enclave can only access argument pages for ECALL/OCALL arguments. After leaving the enclave, the application re-enables full access to the application itself (`KA`) as well as the argument pages (`KE`) via `WRPKRU KA|KE`.

To prevent the enclave from manipulating MPK by reconfiguring the PKRU register, HSGXJail demands a slight modification to the SGX specification. Whenever HSGXJail is active, the `WRPKRU` instruction is disallowed for the enclave and raises an invalid opcode exception instead. This change should be easily adaptable via a microcode update to the CPU.

HSGXJail poses no limit on the number of applications using third-party enclaves, however, the number of enclaves within a single application is restricted. Since MPK supports up to 16 different protection domains, HSGXJail can natively secure applications utilizing up to 15 distinct enclaves. Note that one protection domain is needed for the application itself. To support more enclaves per application, one can follow various approaches: First, in many cases enclaves provide simple functionality, e.g., ECALLs without OCALLs, or OCALLs for issuing syscalls but not towards other enclaves. In these cases, enclaves are never called in an interleaved way and thus, are never concurrently active. Hence, the application can safely share the same argument pages and also the same protection key among those enclaves. This increases the number of supported enclaves by the degree of enclaves which are not interleaved with other enclaves. Second, memory protection keys can be dynamically updated and scheduled among different enclaves. While this supports an arbitrary large number of enclaves per application, it incurs additional performance penalty in updating protection keys in the PTEs.

Control Confinement. Whenever leaving enclave execution (via ECALLs and OCALLs), the enclave jumps into the host application via an `EEXIT` instruction. However, since the enclave can freely choose the jump target of `EEXIT`, a variety of code-reuse attacks become possible (cf. Section 4).

Data confinement already limits an enclave's read and write access by means of MPK. While MPK protects data accesses, it does not prevent fetching code from other protection domains. This design choice is intentional to enable application code to update protection domains without accidentally removing access to its own code. Hence, data confinement does nothing to protect an application from rogue `EEXIT`s.

To break the arbitrary `EEXIT` primitive, HSGXJail restricts `EEXIT` to a single valid *exit point*. In particular, `EEXIT` can only target the instruction immediately following a so-called `CEENTER` instruction. This *exit point* is similar to the enclave *entry points* used to protect an enclave from malicious applications, both of which are shown as call gates (CG) in Figure 5.

Control confinement can be easily implemented via small changes to SGX. We propose to extend the semantics of `EENTER` via a novel confined `CEENTER` instruction. From the enclave's perspective, `CEENTER` behaves exactly as `EENTER`. `EENTER` already stores the exit point (*i.e.*, the address of the instruction immediately following `EENTER`) in register `RCX`. However, SGX leaves it up to the enclave to store this exit point and later on pass it to `EEXIT`. In contrast, our `CEENTER` instruction additionally stores the exit point in a protected, thread-local CPU register called `OEXIT` which is inaccessible to the enclave. To make use of this exit point, we propose to adapt the semantics of the `EEXIT` instruction, as follows: Instead of jumping to a target provided by the enclave via register `RBX`, our `EEXIT` ignores `RBX` and instead directly jumps to the address stored in the protected `OEXIT` register. Both, `CEENTER` and `EEXIT` can be implemented in CPU microcode.

Compatibility. To be fully compatible with existing enclave software, we activate HSGXJail only on demand. If the application issues a normal `EENTER` instruction, HSGXJail is inactive and SGX behaves as usual. When entering the enclave via our new confined `CEENTER` instruction, HSGXJail is active until `EEXIT`. Moreover, HSGXJail's slim design is fully compatible with advanced SGX features such as multithreading, dynamic memory management and virtualization [26]. Availability of HSGXJail can be indicated via a model-specific register.

Software Considerations. HSGXJail protects applications from existing, unmodified third-party enclaves. HSGXJail can be integrated entirely within the SGX SDK [25], thus being fully transparent to existing application code. This allows to use HSGXJail by recompiling applications, without the need to rewrite any application code.

To use HSGXJail, the SDK needs the following slight adaptations. First, the SDK replaces `EENTER` with `CEENTER` in the untrusted `urts` library. The `urts` library already uses a single exit point, which is the address immediately following `EENTER`. The corresponding trusted `trts` library belonging to the enclave performs `EEXIT` only towards this single exit point. Since our modified `EEXIT` instruction enforces the same exit point, it does not change the behavior of benign enclaves. No changes to the `trts` library are required. Benign enclaves compiled under the original `trts` library work out of the box.

For data confinement, the SGX SDK needs to establish enclave-accessible argument pages reflecting the ECALL/OCALL interface and configure memory protection keys accordingly. By default, all application code runs with protection key *zero*. Thus, the SDK assigns protection keys starting with *one* to all enclave pages as well as the corresponding argument pages. Similar to the software-only variant, SGXJail, the SDK can do this once when loading a new enclave.

When doing an ECALL, the SDK additionally copies all input arguments from application memory to an enclave-accessible argument page. In the same way, the SGX copies back any output arguments from the argument page to application memory at the end of an ECALL. The same applies to OCALLs. While argument copying causes some overhead, it is deemed necessary to generically prevent TOCTOU attacks and guarantee the security of the application. For the same reason, the enclave copies untrusted application arguments to enclave memory before operating on it.

Before entering the enclave, the SDK saves all necessary CPU registers in application memory, clears sensitive content from the registers and configures the application's stack pointer `RSP` to point to one of the argument pages. Configuring `RSP` in that way causes the enclave to read and write any OCALL arguments directly from/to the argument page, which is enclave-accessible, without additional copying overhead. After leaving the enclave, the SDK restores the application's CPU registers, including the stack pointer.

Performance Estimates. The only functionally necessary change for HSGXJail is disallowing the `WRPKRU` instruction on `CEENTER`, which can be easily implemented in the CPU. The microcode changes we propose to `CEENTER` and `EEXIT` for control confinement are minimal and only comprise register operations rather than memory accesses, resulting in negligible performance overhead. Second, data confinement via MPK requires no change and shows the same performance as for MPK without HSGXJail. Hence, it is reasonable to expect a negligible overhead of HSGXJail in every aspect, far lower than the overhead of the software-based SGXJail variant.

6 Related Work

Defense by Detection. Researchers proposed to detect enclave malware by monitoring their I/O behavior [13, 16]. However, this is believed to be infeasible in practice [38]. Others proposed analyzing enclave code before actually running it [13], which is not feasible for generic loaders. Generic loaders can remotely fetch arbitrary malicious code at runtime. Refusing such generic loaders would annihilate all use cases for protecting intellectual property. Instead, Costan et al. [13] proposed to force generic loader enclaves to embed malware analysis code within the enclave. However, it is unclear how effective this technique is in detecting malicious code. It also raises the question who decides which analysis code to embed and to ensure the analysis code does not leak enclave secrets. Also, analysis code cannot be easily updated, and enclaves without analysis code cannot be executed without risk.

Defense by Prevention. While applying control-flow integrity (CFI) to the host application sounds appealing, it does not close all attack vectors outlined in Section 4. Although hardware-assisted CFI can prevent some control-flow attacks [27], they are not yet available and might miss rogue EEXIT attacks. Software CFI schemes like [31, 39] can simply be bypassed by leaking secrets and corrupting CFI metadata via the arbitrary read and write primitives. Moreover, no CFI scheme can prevent data-only attacks.

Readactor [15], Heisenbyte [57], and NEAR [62] severely limit the arbitrary read primitive necessary for many attacks by forcing page faults when trying to access sensitive code. However, they have significantly larger overhead than SGXJail, and blind ROP attacks might still be possible [4]. Ryoan [23] executes malicious enclaves inside a software sandbox using software fault isolation (SFI). However, Ryoan demands recompilation of the enclave with SFI, which cannot be applied in our setting. Also, Ryoan severely restricts the enclave life cycle to a single stateless invocation, which is incompatible to generic third-party enclaves.

7 Discussion

Since the very first blog post in 2013 [48], the enclave malware threat has been discussed at a high level but was mostly disregarded by the research community. With recent attacks showing powerful and practical enclave malware, research on proper defense mechanisms becomes pressing.

In this work, we identified three enclave primitives, namely arbitrary memory reads, writes and `EEXIT`s, which lie at the heart of the enclave malware threat by exposing an application to a variety of runtime attacks originating from misbehaving enclaves. Although these primitives help support different SGX programming models, they not only give rise to enclave malware but they are unnecessary in practice, as enclaves ought to strictly comply with the defined ECALL/OCALL interface. In particular, the enclave runtime services offered by the SGX SDK demand precise EDL specification of the data exchanged, and bypassing this specification is considered bad practice. Moreover, the SDK uses only a single enclave exit point, from which all ECALLs and OCALLs are dispatched.

Based on these observations, we proposed (H)SGXJail to confine enclave primitives to the narrow interface specified by the EDL. This applies the principle of least privileges [49] also to enclaves and closes a entire class of runtime attacks, including information disclosure, control-flow attacks, as well as data-only attacks. Even more, by automatically copying ECALL/OCALL arguments from and to application memory, (H)SGXJail prevents double-fetch bugs [58] by design.

Furthermore, SGXJail paves the way for reasoning about application security based on application code only (*i.e.*, without trusting any enclave code), and the ECALL/OCALL interface in particular. While SGXJail defeats a entire class of runtime attacks, it cannot solve the problem of too permissive host interfaces, e.g., a syscall proxy [38] which allows executing arbitrary syscalls. Further research on designing and validating ECALL/OCALL interfaces is needed to avoid API-level attacks via too permissive OCALLs or confused deputy [22] and Iago attacks [11]. In general, one has to consider enclave-to-host communication not as asymmetric (cf. the kernel's syscall interface) but as part of a mutually distrusted API where both communication parties distrust each other. Mutual distrust is an integral part of designing secure web APIs. Since enclave malware raises similar threats as web applications, we also see some overlap in defense strategies [43]. In special, input validation or sanitization [43, Section V5] can help prevent Iago-style attacks while verification of the logical execution flow [43, Section V11] can prevent confused deputy attacks.

Closing Side Channels. Several side-channel attacks mounted against benign SGX enclaves have been shown [7, 19, 34, 41, 59, 63]. Moreover, malicious enclaves themselves can mount side-channel attacks [21, 53, 54]. Although not the primary focus of this work, SGXJail prevents a variety of side-channel attacks that rely on accessing host application memory, e.g., Flush+Reload on shared host libraries used by the host application from within enclaves, Prime+Probe using host application arrays [54], Rowhammer attacks from within enclaves [21] as well as TSX-based address probing [53].

8 Conclusion

While designed to increase the security of a computing system, secure enclave technology such as Intel SGX might also be misused for shielding malware inside enclaves. However, research on potential enclave malware is still in its beginnings, and practical defense mechanisms are virtually non-existent.

In this work, we identified the root cause of enclave malware as an insufficient enclave-to-host isolation and proposed (H)SGXJail as a generic defense against a wide range of enclave malware threats. (H)SGXJail enforces mutual isolation between host applications and enclaves, thus protecting applications from potentially misbehaving or malicious third-party enclaves. SGXJail is an efficient and transparent software defense, running third-party enclaves in an isolated sandbox. Our proof-of-concept implementation shows zero overhead for pure enclave computation and less than 3% for realistic workloads. SGXJail is tightly integrated within the SGX SDK and can be used out of the box. Furthermore, we propose SGXJail directly in hardware. Our HSGXJail mechanism provides enclave confinement by means of Intel MPK with slim extensions to the SGX specification at virtually no cost. We believe HSGXJail should be immediately rolled out via a microcode update to SGX-enabled CPUs to proactively enable our SGX malware defense. However, support for MPK is still rare. Although some server CPUs support MPK [64], it is unclear when x86-based desktop CPUs catch up.

Apart from defending against enclave malware, (H)SGXJail opens up new use cases for Intel SGX and similar isolation technologies. For example, we envision that (H)SGXJail can be used as lightweight and secure sandboxing mechanism for browser site isolation or plugin management, where third-party code has proven to be both, potentially malicious and potentially security critical.

Acknowledgements

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS*, 2005.

[2] Erick Bauman and Zhiqiang Lin. A case for protecting computer games with SGX. In *Workshop on System Software for Trusted Execution*, 2016.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems*, 2015.

[4] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *S&P*, 2014.

[5] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS*, 2011.

[6] Erik Bosman and Herbert Bos. Framing signals - A return to portable shellcode. In *S&P*, 2014.

[7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[8] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[9] Nicholas Carlini and David A. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security*, 2014.

[10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *CCS*, 2010.

[11] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *ASPLOS*, 2013.

[12] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, 2001.

[13] Victor Costan and Srinivas Devadas. Intel SGX explained. 2016.

[14] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, 1998.

[15] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *S&P*, 2015.

[16] Shaun Davenport and Richard Ford. SGX: the good, the bad and the downright ugly, January 2014. URL: https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly.

[17] Anders T Gjerdrum, Robert Pettersen, Håvard D Johansen, and Dag Johansen. Performance of trusted computing in cloud infrastructures with intel sgx. In *International Conference on Cloud Computing and Services Science*, 2017.

[18] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *S&P*, 2014.

[19] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *EuroSec*, 2017.

[20] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel isolation: From an academic idea to an efficient patch for every computer. *USENIX ;login*, 2018.

[21] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[22] Norman Hardy. The confused deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–38, 1988.

[23] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Usenix OSDI*, 2016.

[24] Intel. Intel Software Guard Extensions SSL. URL: `https://github.com/intel/intel-sgx-ssl`.

[25] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, May 2016. Rev 1.5.

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

[27] Intel. Control-flow Enforcement Technology Preview, June 2017. Revision 2.0.

[28] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *CCS*, 2018.

[29] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.

[30] Klaudia Krawiecka, Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N Asokan. Protecting web passwords from rogue servers using trusted execution environments. *arXiv:1709.01261*, 2017.

[31] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *OSDI*, 2014.

[32] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. Loop-oriented programming: a new code reuse attack to bypass modern defenses. In *IEEE Trustcom/BigDataSE/ISPA*, 2015.

[33] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, 2017.

[34] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security Symposium*, 2017.

[35] Xueping Liang, Sachin Shetty, Deepak Tosh, Charles Kamhoua, Kevin Kwiat, and Laurent Njilla. Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In *International Symposium on Cluster, Cloud and Grid Computing*, 2017.

[36] Linux kernel. SECure COMPuting with filters, 2017. URL: `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[38] Marion Marschalek. The Wolf In SGX Clothing. *Bluehat IL*, January 2018.

[39] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: cryptographically enforced control flow integrity. In *CCS*, 2015.

[40] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014.

[41] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *CHES*, 2017.

[42] Nicolas Bacca. Soft launching ledger SGX enclave, 2017. URL: `https://www.ledger.fr/2017/05/22/soft-launching-ledger-sgx-enclave/`.

[43] OWASP. OWASP application security verification standard 4.0, 2019.

[44] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *S&P*, 2012.

[45] PaX Team. Address space layout randomization (ASLR), 2003. URL: http://pax.grsecurity.net/docs/aslr.txt.

[46] PaX Team. Rap: Rip rop. *Hackers to Hackers Conference*, 2015.

[47] Charlie Reis. Mitigating spectre with site isolation in chrome, 2018. URL: https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html.

[48] Joanna Rutkowska. Thoughts on Intel's upcoming Software Guard Extensions (Part 2), 2013. URL: http://theinvisiblethings.blogspot.com/2013/09/.

[49] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[50] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *S&P*, 2015.

[51] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *S&P*, 2015.

[52] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. *AsiaCCS*, 2018.

[53] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with Intel SGX. In *DIMVA*, 2019.

[54] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[55] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[56] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *S&P*, 2013.

[57] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *CCS*, 2015.

[58] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. A survey of the double-fetch vulnerabilities. *Concurrency and Computation: Practice and Experience*, 30(6), 2018.

[59] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, 2017.

[60] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in Intel SGX enclaves. In *ESORICS*, 2016.

[61] Ofir Weisse, Valeria Bertacco, and Todd M. Austin. Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves. In *ISCA*, 2017.

[62] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *AsiaCCS*, 2016.

[63] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, May 2015.

[64] Mingwei Zhang. XOM-Switch, 2019. URL: https://github.com/intel/xom-switch.