Pandora

Alder, Fritz; Daniel, Lesly-Ann; Oswald, David; Piessens, Frank; Van Bulck, Jo

Pandora: Principled Symbolic Validation of Intel SGX Enclave Runtimes

Fritz Alder[1], Lesly-Ann Daniel[1], David Oswald[2], Frank Piessens[1], and Jo Van Bulck[1]

[1]*DistriNet, KU Leuven, Belgium,* [2]*University of Birmingham, UK*

***Abstract*—The popularity of Intel SGX technology in recent years has given rise to a wide range of *shielding runtimes* to transparently safeguard secure enclave applications against a hostile operating system. Adequate validation of the crucial and numerous shielding runtimes is, however, a multi-faceted and fast-changing challenge, as new attack techniques against SGX enclaves are discovered regularly and commonly necessitate extensive software patches throughout the SGX ecosystem.**

This paper proposes *Pandora*, a practical, enclave-aware symbolic execution tool designed to address this challenge. In contrast to existing tools, Pandora's *truthful* and runtime-agnostic symbolic execution of the *exact* attested enclave binary for the first time allows to validate the critical enclave shielding runtime itself. Furthermore, Pandora provides principled foundations to deal with the moving-target nature of enclave software security by implementing accurate taint tracking of attacker inputs, a precise symbolic enclave memory model, and support for pluggable vulnerability detectors.

We extensively evaluate Pandora on 11 different SGX shielding runtimes with 4 detection plugins for a diverse set of vulnerability types. Our experiments show that Pandora can autonomously discover 200 new and 69 known vulnerable code locations. Notably, Pandora is the first tool that allows a wide-scale ecosystem investigation of recent pointer-alignment software mitigations in real-world SGX enclave runtimes.

1. Introduction

Recent years have seen the rise of trusted execution environments (TEEs) that provide strong, hardware-rooted protection of small software components, called *enclaves*, against hostile, possibly attacker-controlled system software. With the release of the Software Guard Extensions (SGX) [1], [2], included in selected Intel processors from 2015 onwards, TEE protection is readily available in today's mainstream computing platforms, and even more recent technology, like the Trust Domain Extensions (TDX) [3] for upcoming Intel server processors, continues to rely critically on SGX enclaves. Thus, the widespread availability of SGX has boosted ongoing interest in enclave applications and limitations from both industry and academia.

While SGX hardware enforces that enclave memory cannot be accessed from the outside, enclave software remains ultimately responsible to be bug-free and should properly sanitize registers and pointer arguments in the shared address space. This non-trivial requirement has given rise to a sizable ecosystem of SGX *shielding runtimes* that support diverse enclave applications. Modern SGX development paradigms nowadays include *(i)* custom `C`/`C++` software development kits (SDKs) [4], [5] that directly expose a secure function call abstraction; *(ii)* numerous SGX-tailored library operating systems (libOSs) [6]–[10] to support lift-and-shift protection of existing legacy applications; and *(iii)* enclaved memory-safe language runtimes [11]–[14].

The popularity of Intel SGX has, furthermore, triggered a long and ongoing line of attacks exploring limitations of this technology [15]. In this respect, a clear trend has been that, while some of the earlier SGX attacks [16]–[20] could still be mitigated fully transparently at the hardware level by means of CPU microcode patches, progressively more stringent demands have been placed on enclave software behavior to mitigate evermore specific vulnerabilities [20]–[27] when interacting with the untrusted environment. This has increasingly made secure enclave software development, and especially the sanitization responsibilities for the numerous SGX shielding runtimes, a moving target (cf. Section 2).

While software mitigations for transient-execution and side-channel attacks have been widely studied for Intel SGX, and presently various compiler-based solutions [24], [28]–[33] exist, the crucial aspect of validating the security of the enclave interface has received much less attention. Researchers have only recently started to explore more systematic analyses through fuzzing [34]–[36] or symbolic execution [37]–[39]. However, existing approaches fall short in that they focus on validating enclave application logic only, without considering vulnerabilities in the crucial shielding runtime, or even being compatible with diverse runtimes beyond Intel's SGX SDK. Furthermore, existing approaches focus mainly on detecting memory-safety issues, without considering more subtle types of shielding responsibilities, such as untrusted pointer alignments [25], [26] and CPU register sanitizations [21], [22], [27]. These approaches, hence, are not fitted for the diverse and fast-changing SGX software ecosystem, where a subtle sanitization oversight in a shielding runtime may be the equivalent of a zero-day rootkit vulnerability in a commodity OS kernel.

To address these challenges, the main objective of our work is the development of a principled, tool-supported approach to validate the security of enclave software binaries using symbolic execution. We propose Pandora, an extensible, enclave-aware symbolic execution tool that is built upon the popular angr framework and extends it with several novel technical contributions. Particularly, we accurately

implement missing, SGX-specific x86 semantics, conceive a proficient, enclave-aware symbolic memory model, and develop a generic enclave memory extractor. Thus, Pandora for the first time enables *truthful* and runtime-agnostic symbolic exploration of full enclave binaries, identical to the attested initial memory layout and including the crucial shielding runtime itself. Furthermore, to deal with the moving-target nature of secure enclave software development, we propose *pluggable* vulnerability detectors, extending the notion of angr breakpoints with SGX-specific memory-access and control-flow events that allow rapid scripting of powerful Pandora plugins.

Our extensive experimental evaluation on 11 different shielding runtimes from research and industry, with 4 plugins validating diverse sanitizations, highlights the delicacy and complexity of present SGX software responsibilities. We demonstrate the power of Pandora's truthful symbolic execution semantics by identifying several subtle vulnerabilities in commonly overlooked low-level enclave initialization and relocation code that cannot be analyzed with state-of-the-art enclave symbolic-execution tools. We, furthermore, are the first to construct an automated tool for wide-scale validation of intricate untrusted pointer-alignment software mitigations [26], [40] recently deployed throughout the SGX ecosystem in response to ÆPIC [25] attacks.

In the wider research landscape, we envision our open-source tool as a solid foundation to enable future science on validating the security of enclaved software, including low-level and fast-changing SGX software shielding runtimes.

Contributions. In summary, our contributions are:

- We propose Pandora, an extensible, enclave-aware symbolic execution framework for truthful and principled validation of SGX binaries.
- Responding to the heterogeneity of the emerging SGX software landscape, we propose a universal enclave memory extractor and corresponding angr loader.
- Responding to the volatile and elusive SGX software responsibilities, we propose pluggable detectors for diverse vulnerabilities, from validating CPU register cleansing over untrusted pointer sanitization and alignment constraints to control-flow transitions.
- In an extensive experimental evaluation on 11 different SGX runtimes, Pandora autonomously confirmed 69 known and 200 new vulnerable code locations.

Disclosure and Artifacts. We responsibly disclosed all findings to the respective vendors (tracked via 7 CVEs), providing them with comprehensive reports from our tool. We, furthermore, included recommendations for software mitigations and assisted in validating the applied fixes, which has uncovered remaining issues in at least one runtime.

In the spirit of open science, we provide a comprehensive open-source artifact[1] with self-contained HTML reports of all vulnerabilities from Table 2, multiple runtimes to test out Pandora, and documentation of how to reproduce our results. The artifact also includes the binaries of analyzed shielding runtime versions (where allowed by licensing) to provide a representative public data set of vulnerable enclaves that can serve as a baseline for future research.

1. Available at https://github.com/pandora-tee.

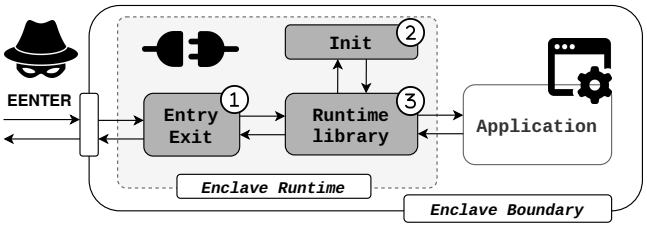

Figure 1. A shielding runtime transparently protects enclave applications by ① cleansing CPU registers upon entry or exit events; ② finalizing the initial memory layout, including any in-enclave relocations, upon first entry; and ③ sanitizing pointer arguments before handing control over to the application, which can call back via trusted standard library functions.

2. Background and Related Work

Enclave Shielding. Due to its strong attacker model, enclave software faces several additional security challenges compared to traditional user-space software. In current practice, these additional challenges are primarily handled by a shielding runtime that transparently intervenes on interactions with the untrusted environment, as shown in Fig. 1.

Intel SGX enclaves are embedded as a contiguous virtual address region within an untrusted, surrounding host application. As in-enclave software is allowed to freely dereference outside memory locations, the host application can efficiently communicate through the enclave's application programming interface (API) by passing pointers to arguments and return values in the shared virtual address space. However, this also opens the door to an especially powerful class of confused-deputy attacks, necessitating that the enclave shielding runtime adequately sanitizes any attacker-provided API pointers prior to dereference. Despite this requirement being well-known and the availability of automated methods—such as the `edger8r` tool to automatically generate interface sanitization code from developer annotations in the Intel SGX-SDK [4] and Open Enclave [5], or the Rust type system leveraged in EDP [11]—a continuous stream of vulnerabilities [21], [34], [37]–[39] has proven SGX pointer sanitization vulnerabilities to be particularly elusive and widespread in practice. As an example, Listing 1 illustrates how adequately sanitizing an elementary pointer-to-pointer argument can be non-trivial in practice.

Moreover, in response to the recently disclosed ÆPIC [25] and related memory-mapped I/O (MMIO) [26] stale data vulnerabilities in Intel processors, enclave software requirements for sanitizing untrusted pointer arguments have been considerably complicated. That is, not only does enclave software nowadays need to ensure that attacker-provided pointers properly fall entirely outside the protected enclave range, but any subsequent pointer dereferences also need to proceed at a certain alignment and size or need to be preceded and followed by fragile x86 instruction sequences

```
void encl_get_from_addr(struct user_arg *op) {
    assert(is_outside_enclave(op, sizeof(*op)));
    // Copy op->addr to avoid TOCTOU attacks
    volatile char* ptr = (char*) op->addr;
    assert(is_outside_enclave(ptr, 1));
    g_state = *ptr; }
```

Listing 1. Example of API sanitization: the highlighted lines enforce that all attacker-controlled pointers lie outside the enclave prior to dereference.

to cleanse microarchitectural buffers and stall the CPU pipeline. These successive refinements of software responsibilities hence necessitated extensive and ongoing changes throughout the heterogeneous SGX software ecosystem.

A parallel moving-target evolution can be observed at the level of the application binary interface (ABI). An initial comprehensive study [21] has shown that secure initialization was widely overlooked for certain crucial CPU configuration flags, such as the x86 direction flag that may introduce memory-safety violations in otherwise secure code. Similar issues have since been shown for stack-pointer initialization in SGX enclave exception handlers [23] and for x87 and SSE floating-point configuration registers [22]. The latter was most recently refined once again in an Intel advisory [27] with additional SSE sanitizations to protect against certain operand-dependent floating-point instruction timing channels in otherwise constant-time code. A recent overview study [41] has documented how these ABI vulnerability disclosures necessitated several rounds of widespread patches throughout popular SGX shielding runtimes.

Symbolic Execution. Symbolic execution [42] statically interprets a program using *symbolic* inputs (*i.e.*, mathematical terms) and collects *constraints* (*i.e.*, mathematical formulas over these terms) encoding programs paths. These constraints can be solved with an SMT solver to generate concrete inputs exercising the path or check security assertions. Its ability to systematically explore program paths and generate concrete inputs has made symbolic execution a tool of choice for intensive testing [43] and vulnerability analysis [44]. More recently, researchers have also started to apply symbolic execution to the specific context of Intel SGX enclaves [37]–[39]. We provide an extensive comparison of Pandora to these existing tools in Section 3.1. Some works [20], [45] have, furthermore, focused on detecting microarchitectural side-channel vulnerabilities in enclave applications using symbolic execution, but their goal is orthogonal to our scope of validating shielding responsibilities.

Fuzzing. A well-known, complementary approach to static analysis via symbolic execution is dynamic concrete execution via fuzz testing. An orthogonal and concurrent line of work [34], [35], [46], [47] has started to explore such fuzzing for Intel SGX enclave applications. Compared to symbolic execution, fuzzing can more easily scale to complex code bases by quickly generating test cases and may find bugs with fewer false positives. However, unlike symbolic execution, fuzzing requires carefully crafted test cases to investigate convoluted execution paths. Hence, in line with existing surveys [48], [49], we regard fuzzing-based approaches as complementary to symbolic validation.

3. Problem Statement and Overview

The combination of a varied and evolving Intel SGX runtime ecosystem with the frequent discovery of new attack techniques that necessitate additional software sanitizations makes the problem of principled enclave software validation particularly challenging and, indeed, largely unexplored for the fundamental shielding runtimes themselves. Therefore, we set the following goals:

G1 *Truthful symbolic exploration.* Enclave-aware symbolic execution should closely mimic the real SGX hardware. Particularly, to not miss vulnerabilities in the runtime itself, the symbolic exploration should *(a)* start from the very first entry instruction without skipping initialization procedures or stubbing runtime library functions; and *(b)* operate on the *exact* initial memory contents, as remotely attested via MRENCLAVE [50], while accurately detecting and symbolizing any subsequent accesses to untrusted or unmeasured memory.

G2 *Runtime-agnostic.* Validation should not be limited to enclaves developed with any specific single shielding runtime. The heterogeneous SGX ecosystem with ill-documented and varying enclave binary formats calls for a lightweight conversion approach to a unified format capturing the exact enclave memory layout.

G3 *Extensible validation policies.* The system should support prompt reactions to evolving sanitization responsibilities by adding new or modified vulnerability detection plugins. This calls for an approach that decouples validation *policies* from enclave-aware symbolic execution *mechanisms*, such that plugins can solely focus on elegantly expressing the required software security invariants to be validated for explored paths.

In addition to these three research goals, we define the following secondary design challenge:

D1 *Accessibility.* The tool should be open-source and easy to use, including on closed-source binary targets. Reports should be easily interpretable by human analysts.

3.1. Research Gap

Initially, SGX software vulnerability research was mainly guided through manual code review [21]–[23], [41], whereas automated enclave analysis through symbolic execution has only more recently started to be explored [37]–[39]. Table 1 compares Pandora to these existing tools. In summary, existing approaches are mostly focused on application bug detection instead of principled validation of the absence of shielding runtime vulnerabilities. This means that they are inherently insufficient for truthful symbolic exploration (*G1*), as the focus is on analyzing enclave application logic only, while (largely) skipping the underlying shielding runtime and operating on inaccurate initial memory contents. Moreover, existing tools are ill-fitted for the diverse

TABLE 1. COMPARISON OF SYMBOLIC-EXECUTION TOOLS FOR SGX.

		Runtime			Binary	Dump	Reentry	Plugins				
Tool	App	SDK	Entry	Init				Ptr	ABI	ÆPIC	Jmp	Open
TEEREX [37]	●	Intel	○	○	●	○	◐	◐	○	○	◐	○
Guardian [38]	●	Intel	●	○	◐	○	○	◐	◐	○	◐	●
COIN [39]	●	Intel	○	○	○	○	○	◐	○	○	○	●
Pandora	●	*any*	●	●	●	●	●	●	●	●	●	●

Features can be fully (●), partially (◐), or not (○) supported. Columns 4–7 denote whether the tool executes the runtime *entry* and *init*ialization phases; can handle *binaries* without additional specification; and uses the exact memory layout (*dump*).

SGX ecosystem (*G2*), as they all make runtime-specific assumptions that strictly limit them to enclaves developed with Intel's SGX SDK only. Finally, existing tools focus mainly on a narrow set of classical memory-safety issues for pointers without principally supporting more intricate shielding responsibilities (*G3*), such as recently rolled out pointer-alignment ÆPIC mitigations [25], [26].

TEERex. TEEREX [37] is a closed-source prototype to detect memory corruption vulnerabilities in enclave applications developed with the Intel SGX SDK. Similarly to our work, TEEREX is based on angr [48], a popular symbolic execution tool for binary code, and performs taint tracking of untrusted attacker arguments and memory accesses outside the enclave using unconstrained symbolic values.

In contrast to Pandora, however, TEEREX does not support truthful symbolic exploration (vs. *G1a*), as it entirely skips analysis of the whole trusted runtime and directly performs symbolic execution of enclave application entry points, called `ecalls`. Moreover, TEEREX is inherently runtime-specific (vs. *G2*), as it relies on Intel SGX SDK-specifics to identify addresses of `ecall` functions, to hook specific pointer validation functions, and to set up an approximate, non-truthful initial memory layout (vs. *G1b*). Concerning vulnerability detection (*G3*), TEEREX only reports unconstrained and NULL-pointer dereferences and cannot detect more subtle pointer issues, or ABI and ÆPIC issues. Particularly, by hooking the crucial validation functions (e.g., `is_outside_enclave` in Listing 1), TEEREX may miss logical partial validation errors [21] that will be caught by Pandora's precise enclave-aware memory model (cf. Section 7). TEEREX is not openly available (vs. *D1*).

Guardian. Guardian [38] is similarly based on angr and can partially check API and ABI shielding policies. Regarding truthful exploration (*G1a*), Guardian is the only prior work that starts at the enclave entry point within the trusted runtime, but it nevertheless skips the complex enclave initialization phase, which may still contain critical vulnerabilities (cf. Section 7). Furthermore, similar to TEEREX, Guardian is constrained to binaries developed with specific versions of the Intel SGX SDK (vs. *G2*) and only constructs an approximate, non-truthful initial memory layout (vs. *G1b*). As to vulnerability detection (*G3*), Guardian validates a principled, yet fundamentally incomplete *orderliness* policy, where the developer is required to manually annotate execution phases (vs. *D1*). Guardian validates that, after the entry phase, an

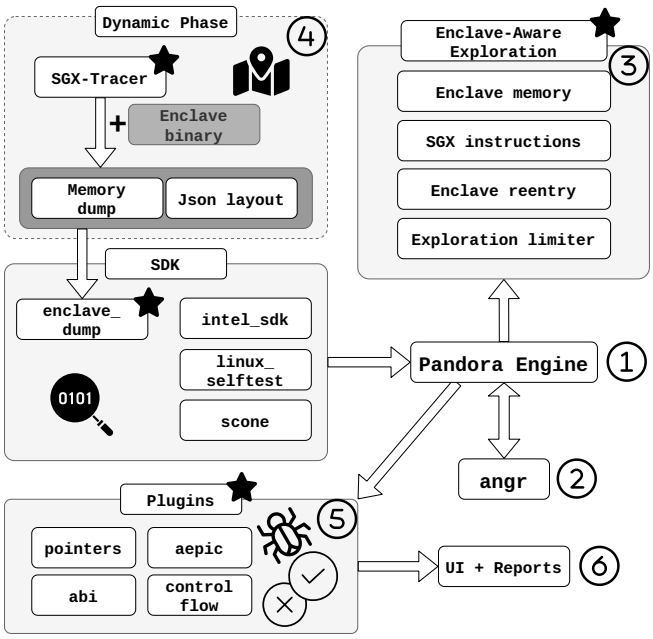

Figure 2. Overview of the Pandora architecture.

(incomplete) blocklist of ABI configuration registers has been cleared, and that untrusted memory outside the enclave is only accessible during execution of the shielding runtime, but not during the application phase. This simplified permission state-machine model may be overly conservative for applications and, more problematically, remains inherently insufficient to detect critical vulnerabilities (e.g., CVE-2018-3626 [21]) in the shielding runtime itself, as the latter is allowed unrestricted access to the full address space.

COIN. COIN [39] uses concolic execution to find memory-safety vulnerabilities in enclave applications. COIN specifically targets applications developed on top of the Intel SGX SDK (vs. *G2*) and requires the enclave source code (vs. *D1*) for extracting the parameters of `ecalls` in order to set up an approximate, non-truthful initial state (vs. *G1b*). Regarding vulnerability detection (*G3*), COIN is largely orthogonal to our work by focusing on traditional memory-safety application vulnerabilities instead of nuanced, enclave-specific shielding issues and skipping analysis of the runtime itself (vs. *G1a*).

Finally, upon finalization of our paper, a concurrent study called SymGX [51] was published, focusing on detecting cross-boundary pointer vulnerabilities in the source code of Intel SGX applications.

3.2. Solution Overview

Figure 2 depicts a high-level overview of the Pandora software architecture, which we implemented in 5,934 lines of extensible Python code (as measured by `sloccount`). At Pandora's core, the engine component ① augments the

underlying symbolic execution library angr ② [52] with accurate SGX semantics and drives the enclave-aware truthful symbolic exploration ③ (*G1*), described in Section 4. The engine is primed with the exact initial enclave image via a novel, runtime-agnostic dynamic memory extraction phase ④ (*G2*) detailed in Section 5. As such, Pandora is the first symbolic-execution tool that can find vulnerabilities before the application code, *i.e.*, in the runtime entry procedures and in the low-level enclave initialization phase.

While symbolically executing a binary, the Pandora engine triggers vulnerability-detection plugins ⑤ (*G3*), described in Section 6, that are based on subscribable events exposed by the SGX-aware exploration. After a completed run, Pandora formats the findings of each plugin into convenient and interactive HTML reports ⑥ (*D1*), shown in Appendix A, including severity levels, descriptions, disassembly, register dumps, and full basic-block backtraces to enable human analysts to investigate the reported issues.

4. Enclave-Aware Symbolic Execution (*G1*)

4.1. Modeling x86 Instruction Semantics

The underlying VEX representation used by angr does not have a symbolic model for many x86 instructions that commonly occur in enclave binaries. Most prominently, the ENCLU user leaf instructions [53] are used inside the enclave to perform architectural tasks, such as creating a local attestation report (EREPORT), generating cryptographic keys (EGETKEY), or exiting the enclave (EEXIT). While prior work faced similar angr limitations and either did not execute [37] or merely hooked and skipped [38] over these instructions, Pandora truthfully emulates used enclave instructions as closely as possible. For example, in EREPORT, we copy the relevant SGX enclave control structure (SECS) fields provided by the enclave loader, including the processor extended features request mask, into the generated report structure. When specific fields are not available and no sane defaults can be provided, values are symbolized to ensure that all possible paths are explored.

Furthermore, in response to advanced ABI attacks [22], [27], instructions like XSAVE and XRSTOR or their variants are commonly used to save and restore extended x86 register on enclave context switches. In contrast to prior work [37], [38], Pandora carefully emulates their behavior as closely as possible. Where necessary, we add dedicated shadow registers to keep track of special x86 registers, such as MXCSR, which are not normally part of angr's execution model. As shown in Section 7, this precise register view enables Pandora plugins to accurately uncover subtle oversights, e.g., attacker-controlled registers when switching to enclave functions or insecure MXCSR configuration values.

4.2. Taint Tracking of Attacker Inputs

In order to accurately deal with attacker-controlled inputs, Pandora comes with a capable symbolic taint-tracking mechanism. Specifically, initial register contents on enclave entry, as well as memory reads from outside the enclave or from uninitialized unmeasured pages inside the enclave (cf. Section 4.3), are transparently replaced with unconstrained symbolic values. Thus, the symbolic execution initially makes no assumptions about attacker-provided inputs, until specific constraints are added by any subsequent sanitizations performed by the enclave code. Pandora, furthermore, uses angr's annotation system to mark attacker-controlled symbolic values with an *attacker-taint*, which is conservatively propagated during symbolic execution and can be conveniently queried by plugins. For instance, plugins can check that values are properly sanitized (e.g., Section 6.1) or react differently based on whether a value is attacker-tainted or not (e.g., Section 6.2).

Pandora's taint tracking mechanism only tracks explicit data flows. Any implicit flows that result from attacker-controlled control flow are ignored (e.g., Pandora does not propagate the taint from `x` to `y` in `if(x == 1){ y = 1 }`). While tracking only explicit flows may, in principle, lead to false negatives, it brings a large increase in practicality [54] and is common in taint-tracking-based vulnerability detection [55].

4.3. Enclave-Aware Memory Model

Pandora features a fully enclave-aware memory model that truthfully simulates the enclave address space in a more accurate and expressive way than prior work, while also including reasonable performance optimizations. Particularly, we are the first to realize a precise, runtime-agnostic enclave memory model that properly recognizes attacker-controlled symbolic addresses and sizes and that takes into account novel attack surface from unmeasured SGX enclave pages.

4.3.1. Address-Space Partitioning. At its core, we implemented our enclave-aware memory model as an angr `MemoryMixin` extension that performs rigorous checks on every memory access. Particularly, we use angr's constraint solver to unambiguously decide for every accessed buffer with a possibly symbolic address and size whether it is restricted to *(i)* lie fully inside the enclave; *(ii)* lie fully outside the enclave; or *(iii)* partially touch the protected enclave range. Accesses to memory inside or outside the enclave will be handled differently, as outline below. Pandora plugins can, furthermore, subscribe to these respective events to check and report specific vulnerabilities (cf. Section 6).

Note that the above accurate classification is non-trivial to implement, and prior work side-stepped these intricacies by either hooking runtime-specific pointer-validation functions [37] or ignoring the (possibly symbolic and attacker-controlled) size of memory reads [38]. Our fully symbolic memory model, on the other hand, allows to meticulously detect subtle oversights or logical errors in the crucial validation functions themselves. For instance, Section 7 discusses a particularly intricate finding where overflow protection logic was silently optimized away by the compiler.

4.3.2. Untrusted Memory Accesses. For accesses falling outside the protected enclave range, we model the strongest type of adversary that utilizes tools such as SGX-Step [56] to perform instruction-granular time-of-check to time-of-use attacks. For example, an enclave checking an external pointer that resides in untrusted memory, before accessing this pointer again at a later time (as in Listing 1) may realistically receive two different values. Pandora truthfully simulates this by ignoring untrusted memory writes and fully symbolizing all untrusted memory reads with a fresh attacker-tainted symbolic value on *every access*.

4.3.3. Enclave Memory Accesses. In close accordance with the SGX specification [53], we distinguish two types of memory inside the enclave: measured and unmeasured pages. Measured enclave pages are attested as part of the MRENCLAVE enclave identity and are, hence, always demonstrably initialized to the exact value provided by the enclave loader. Unmeasured enclave pages, on the other hand, are protected from enclave creation time onwards, but their initial content is *not* attested as part of the MRENCLAVE enclave identity. These unmeasured enclave pages have many uses in enclaves, for example to reserve heap memory or to load additional code or data during execution that did not exist at enclave creation time yet. As the initial value of these pages is not part of the enclave identity, and thus under attacker control, enclave software must always securely overwrite these pages before first use. However, to the best of our knowledge, to date no sanitizer exists to validate this critical security property. To enable this with Pandora, we ensure that any read from unmeasured enclave memory initially returns an attacker-tainted symbolic value. Only when unmeasured bytes are securely initialized, we create an angr memory backing and the newly written secure values will be taken into account for future reads.

Pandora, furthermore, implements two types of safe performance optimizations. First, we remove measured and initialized unmeasured enclave memory that consists of all-zero bytes from the angr backend. Any reads from such regions will statically return zero bytes until they are overwritten with non-zero data. Second, only for source and destination buffers that are constrained to fall entirely inside the enclave, we optionally hook common memory-management functions (`memcpy` and `memset`) and x86 `rep` string operations with custom `SimProcedures` that eliminate loop overhead, while still taking care to trigger any relevant angr mixins and breakpoints.

4.4. Enclave Entry and Reentry

During enclave lifetime, `EEXIT` and `EENTER` instructions can switch execution to and from the untrusted environment. Prior work [37]–[39] relied on parsing runtime-specific and fragile data structures to find out the supported `ecalls` in order to skip the crucial runtime entry and/or initialization phases entirely and immediately start executing at the respective application `ecall` function.

4.4.1. Enclave Entry. To truthfully execute entry into the enclave, we parse the actual thread control structure (TCS) from enclave memory to retrieve the entry point location and fill registers with the exact same values that they would receive from the architecture, such as the TCS address and `FS` and `GS` base addresses. All other registers are filled with unconstrained, attacker-tainted symbolic values to initiate Pandora's taint-tracking mechanism (cf. Section 4.2).

4.4.2. Enclave Exit. Pandora allows to truthfully build up enclave state by emulating a new `EENTER` with the same accumulated memory view after a symbolic path reached the `EEXIT` instruction. Hence, the enclave entry code in the runtime itself will perform any necessary checks and autonomously decide whether the entry request is an `ecall` or an `ocall` return and dispatch this request accordingly. The strength of this approach is that subtle attack vectors, like dereferencing a function pointer before in-enclave relocation (cf. Section 7) or returning from an `ocall` where no prior `ocall` was executed [21], can in principle be detected.

4.5. Path Exploration and State Reduction

Pandora's unique focus on truthful symbolic exploration of the *entire* enclave binary, including low-level shielding runtime code, comes with the potential cost of state explosion. To reduce memory consumption for individual explorations, Pandora optionally supports depth-first exploration in addition to breadth-first exploration.

With regard to reentry, every path that reached `EEXIT` would have to be reentered in a naive approach, because the enclave may have accumulated relevant global state. However, we observed that many paths result in a clean failure that is reported to the untrusted world with the request to restart the enclave with correct parameters. To avoid redundantly exploring all these semantically equivalent traces, we implement a novel *state uniqueness reduction* before reentering enclave exploration. That is, two symbolic `EEXIT` states are different from each other only if they have made different changes to the internal memory of the enclave. For example, two enclave traces that both result in no changes to the enclave except setting a specific bit indicating that the enclave failed, are equivalent and reentering both would be redundant. With this uniqueness criterion, we thus remove all non-unique enclave traces before preparing them for reentry, *i.e.*, before Pandora executes on them again. Note that this approach is a safe over-approximation, e.g., states may still be semantically equivalent even though they differ in some de-allocated stack variables. However, we found that our state uniqueness reduction is sufficient to greatly reduce the state space without risking that unique states may be lost. The impact of this optimization ultimately depends on the runtime, *i.e.*, on the number of individual paths that lead to enclave exits. Specifically, for the runtimes investigated in this work, this state reduction has an efficacy between 14% (EnclaveOS, 13 of 93 exit states pruned) and 60% (Occlum, 1694 of 2811 exit states pruned).

5. Runtime-Agnostic Enclave Loading (*G2*)

Truthful symbolic execution naturally starts with an accurate representation of the initial enclave memory layout (*G1b*). Unfortunately, however, in contrast to well-established standards like the executable and linkable format (ELF) for Linux binaries, there exists no standardized format to distribute SGX binaries. Hence, over the last years, all SGX shielding runtimes have adopted their own custom formats to describe the additional information needed to correctly load the enclave, e.g., often by encoding opaque blobs into additional ELF metadata sections [4], [5]. This is especially problematic as Intel SGX requires a particularly involved, multi-stage loading process [1], [53].

First, the untrusted system software constructs the initial enclave memory layout, containing regions for code and data, and also including several unique enclave-specific data structures. The two most prominent data structures are the SECS structure describing, among others, the enclave load address and size, as well as the TCSs, describing the enclave entry point and thread-local data storage. Furthermore, as SGX enclaves are commonly compiled as position-independent code and loaded as dynamic libraries, the MRENCLAVE identity must be independent of the load address. Hence, the enclave cannot rely on the untrusted loader to perform any remaining ELF relocations (e.g., for dynamic function-pointer tables). Thus, as a second loading step, enclave shielding runtimes generally include in-enclave code to perform any necessary ELF relocations upon the first enclave entry, *i.e.*, *after* the enclave has already been created and loaded into memory.

Static Analysis. Notably, all prior works [37]–[39] on SGX-aware symbolic execution entirely side-step the aforementioned intricacies by restricting themselves to one particular runtime, specifically the Intel SGX SDK, and by loading the enclave largely as a normal ELF file. Particularly, existing approaches only take care to create approximate space for stack and heap and either skip to the application directly [37], or they manually patch fragile and version-specific global data structures to falsely mark the symbolic enclave as initialized and skip over the costly, low-level runtime initialization and relocation phases [38]. Thus, prior works simulate an inaccurate enclave memory layout (vs. *G1b*) and are, moreover, only compatible with one specific version of one specific runtime (vs. *G2*).

We argue that, with ample code review or reverse-engineering efforts, it is in principle possible to devise an approach that accurately mimics the runtime-specific loading process to construct a truthful initial memory layout, satisfying *G1b*. Indeed, Appendix B describes such optional support we added to Pandora to load enclave binaries from selected runtimes based on static analysis of a given enclave binary. We found, however, that such a purely static-analysis approach is highly labor-intensive and inherently fragile, requiring to implement a custom loader for every studied enclave runtime, possibly even with changes across runtime versions. This would evidently limit the scope and not satisfy our vision of runtime-agnostic analysis for the sprawling SGX ecosystem that has become heterogeneous both in runtime capabilities as well as in programming languages available to the enclave developer.

Dynamic Enclave Memory Extraction. To overcome the labor-intensity and inherent fragility of the above pure static analysis approach with runtime-specific loaders, Pandora supports a more powerful approach that requires a short-lived dynamic execution phase to load the binary-under-test once. Specifically, we developed a minimal standalone program, called SGX-TRACER, to passively observe the loading process of an enclave binary on actual Intel SGX hardware.[2] SGX-TRACER consists of about 400 lines of C code and uses the `ptrace` Linux system call to attach to the untrusted enclave host process and intercept all calls to the (in-kernel or out-of-tree) Intel SGX driver. SGX-TRACER can thus fully transparently *(i)* detect enclave creation via `ECREATE` and record crucial enclave SECS metadata, including load address and size; *(ii)* record the exact memory contents of all pages that are subsequently added via `EADD`; and *(iii)* track additional metadata and permissions for these pages, as well as locate special pages like TCSs, before the enclave identity is finalized via `EINIT`. This allows SGX-TRACER to accurately extract the *exact* initial enclave memory (*G1b*), as attested by MRENCLAVE, for *any* SGX process (*G2*).

The output by SGX-TRACER is stored as a binary dump and accompanying JSON file and can subsequently be used on non-SGX hardware by Pandora. Particularly, we developed a minimal angr loader to reconstruct a truthful symbolic memory view, including permissions of each page and whether the page is measured or unmeasured (cf. Section 4.3). This inherently runtime-agnostic loader makes Pandora compatible with *any* enclave dump extracted via SGX-TRACER, regardless of runtime-specific loading details.

One downside of utilizing an enclave memory dump for symbolic execution is that this process loses all debug symbols, including function names. Pandora can run without any of these symbols, but upon finding a potential vulnerability, the generated reports may be less understandable for human analysts (vs. *D1*). Hence, we implemented a custom symbol handler that can augment a plain memory dump extracted by SGX-TRACER with symbol information from the original ELF file, if optionally provided via a Pandora command-line option (together with a static offset).

6. Pluggable Vulnerability Detection (*G3*)

During symbolic exploration, angr triggers a set of breakpoints that can be hooked to investigate the symbolic state. Exemplary angr breakpoints are memory or register accesses and function calls. Pandora extends the legacy angr events with a set of eight new *enclave-specific breakpoints* (cf. Appendix C). Specifically, Pandora exposes breakpoints before and after enclave entry and exit, as well as breakpoints before and after symbolic memory reads and writes

2. Real SGX hardware may not even be a strict requirement, as SGX-TRACER could, in principle, also spoof the existence of the SGX driver.

that are restricted to resolve fully inside, fully outside, or partially overlapping with the enclave memory range.

Pandora's enclave-aware breakpoints form the basis for our notion of pluggable vulnerability detection (*G3*). Specifically, specialized plugins can subscribe to relevant enclave events, as well as legacy angr breakpoints, to accurately validate certain software invariants during symbolic exploration. We created 4 plugins for a diverse set of enclave shielding runtime responsibilities at the levels of ABI register cleansing, API-level pointer arguments, ÆPIC-style pointer alignment considerations, and attacker-controlled control flows. Plugins can, furthermore, make use of Pandora's built-in reporting interface (*D1*) to conveniently summarize any findings in human-readable HTML reports that are automatically annotated with all relevant information, e.g., a severity score and description of the issue and how to reach the vulnerable state (cf. Appendix A).

6.1. ABI-Level CPU Register Sanitization

Enclaves share the CPU register set with their untrusted surrounding host process. An important responsibility of the shielding runtime is, therefore, to securely initialize any low-level configurations registers on enclave entry. Due to the intricacies of these low-level register manipulations, those sanitizations have to be carefully implemented in a fragile, hand-written assembly stub before a jump into higher-level languages can be securely made, compliant with ABI expectations [57], [58] by the compiler.

While the general concept of ABI-level sanitization is relatively well-understood across SGX shielding runtimes, an ongoing line of manually discovered vulnerabilities [21]–[23], [27], [41] has underlined the intricacies and challenges for secure register initialization in the complex x86 instruction set. Prior work on automated enclave software vulnerability detection has either fully ignored CPU register sanitization by focusing on API validation only [34], [37], [39], or resorted to a simplistic and incomplete blocklist approach that merely checks whether selected CPU registers have certain concrete safe values [38]. On the other hand, Pandora's `ABISan` plugin proposes a more principled approach based on taint tracking, which can autonomously discover insufficient register initialization or cleansing.

6.1.1. Attacker-Tainted Configuration Registers. The `ABISan` plugin hooks all angr register read events and relies on Pandora's taint-tracking mechanism (cf. Section 4.2) to detect when unsanitized CPU configuration registers are read. To avoid evident false positives, `ABISan` only requires a concise allowlist for the x86 data registers, *i.e.*, the 16 general-purpose registers, 16 vector registers, and floating-point unit (FPU) register stack, which do *not* contain control or status bits and, hence, are allowed to be tainted with attacker inputs. Any other attacker-tainted register reads will be automatically reported as critical policy violations.

Our systematic taint-tracking approach has two main strengths compared to simply checking that an incomplete subset of registers has been initialized to certain values [38]. First, `ABISan` can *autonomously* track all relevant occurrences where the attacker has influence over the result of a computation through control registers.[3] This may, in principle, even include yet unknown ABI attack avenues. For instance, we experimentally validated that `ABISan` can fully autonomously discover attacker-tainted reads from individual bits in the RFLAGS [21] register, e.g., the crucial direction flag for x86 `REP` string instructions, as well as a particularly subtle oversight for floating-point operations that required several rounds of patches in Rust-EDP and OpenEnclave to make sure that not only the x87 FPU control word is initialized, but also the internal x87 tag word [22]. Second, `ABISan` also enables tracking advanced attack vectors where the enclave would inadvertently restore tainted control registers prior to using them in a computation.

6.1.2. Enclave Entry Sanitization. Our `ABISan` plugin inspects the complete register state when reaching the first `CALL` instruction inside the enclave. Indeed, the first function call inside the enclave revealed to be a surprisingly effective heuristic for the switch from assembly sanitization code to the higher-level, compiler-generated API entry point: across the 11 investigated runtimes, only a single runtime performed a `CALL` from inside assembly code before jumping to `C` code, which we accommodated in our heuristic. Upon reaching the API entry point, `ABISan` warns for every control and data register that has not been entirely cleared of attacker-tainted data.

Thanks to Pandora's powerful taint-tracking mechanism and enclave-aware execution model, we were able to express the entire `ABISan` policy in only 142 lines of Python code. It is important to note that the flexible nature of our plugins allows for quickly reacting to the ever-changing landscape of recommendations to ABI sanitization responsibilities for Intel SGX. For example, initial research [22] first investigated issues with incomplete sanitization of floating-point control registers and recommended setting the MXCSR register to the ABI-specified value of `0x1F80` on enclave entry. More recently, however, Intel [27] further nuanced secure MXCSR initialization by recommending the value `0x1FBF`, which additionally sets all floating-point exception status flags, to protect against subtle, one-cycle timing differences dependent on (possibly secret) floating-point operand values. We were able to swiftly incorporate this latest recommendation into `ABISan`'s validation policy. This demonstrates that our plugin system can react flexibly and promptly to such updated recommendations, which, as we will show in Section 7, require changes that propagate slowly throughout the Intel SGX software ecosystem.

6.2. Untrusted Pointer Value Sanitization

We implemented a capable `PTRSan` plugin in 120 lines of Python code that proposes three expressive security in-

3. The only limitation here is that we are restricted to the subset of x86 behavior that is emulated by angr. For instance, angr does not consider the alignment-check flag in RFLAGS and largely ignores floating-point precision configuration bits in the underlying VEX symbolic-execution engine.

variants to catch the pervasive issues of confused-deputy attacks via untrusted pointer arguments in the shared address space. Note that, in contrast to prior work [37]–[39], PTRSan is entirely independent of the runtime-specific sanitization function, solely relying on Pandora's built-in taint tracking and enclave-aware memory model. Hence, as demonstrated in Section 7, PTRSan for the first time allows to find subtle logical errors in the sanitization logic itself.

6.2.1. Address Inside or Outside Enclave. Any symbolic memory access that crosses the enclave boundary, even partially, violates the trusted-untrusted memory division. This case arises when, according to the constraint solver, a symbolic address and size pair can have concrete values that fall both inside and outside the enclave's protected address range. PTRSan, hence, always reports such cases as a critical issue of a pointer that has not been sufficiently constrained by the enclave software.

6.2.2. Tainted In-Enclave Address. Attacker-tainted accesses that are constrained to resolve entirely in untrusted memory are clearly benign behavior of the enclave. On the other hand, attacker-tainted accesses that are constrained to always lie *entirely* in trusted enclave memory may still be benign behavior, e.g., an attacker-controlled, yet constrained index into an in-enclave array data structure. Hence, we only report a warning in these cases and mark them as potential issues that may warrant manual and application-specific further inspection. To simplify such further analysis, PTRSan reports the size and maximum address range of the tainted memory access. This criterion to warn for tainted in-enclave memory accesses thus ensures that no clear violation of secure memory accesses can occur, at the potential burden of occasional false-positive warnings. These false positive are non-straightforward to eliminate generically, but we discuss possible enhancements and heuristics in Section 8.

6.2.3. Untainted Outside-Enclave Address. Untainted accesses that are constrained to always resolve entirely in enclave memory are clearly benign behavior of the enclave. However, if untrusted memory is ever accessed with an address that is *not* tainted by the attacker, PTRSan sees this as a critical issue hinting at unexpected behavior, e.g., an uninitialized or NULL pointer dereference.

6.3. Untrusted Pointer Alignment Sanitization

The recently disclosed ÆPIC [25] and MMIO stale data leakage [26] attacks on Intel SGX platforms have shown that enclave secrets may propagate from microarchitectural fill buffers into architectural, software-visible registers when dereferencing unaligned pointers to MMIO devices. While CPU microcode updates have since been released to transparently cleanse fill buffers upon enclave exits on affected processors, additional software mitigations are still necessary to prevent confused-deputy exploitation of these issues during enclave execution [26], [40]. That is, even when the enclave shielding runtime has properly checked that untrusted, attacker-tainted pointer arguments fall entirely outside the enclave memory range, as can be validated by PTRSan, SGX enclaves have no way of knowing whether these untrusted memory locations refer to vulnerable MMIO regions. Indeed, privileged adversaries can trivially map untrusted memory pages to arbitrary MMIO devices, including the x86 APIC configuration registers [56]. As such, dereferencing untrusted pointers during enclave execution may unintentionally expose secret stale data, and Intel explicitly advises that SGX shielding runtimes should additionally constrain untrusted pointer dereferences to certain safe combinations of alignments and lengths [26], [40]. Note that this holds both for outside-enclave reads and writes, through the shared buffers data read (SBDR) and device register partial write (DRPW) processor vulnerabilities, respectively.

In response to these dynamic challenges, we developed a specialized ÆPICSan plugin, which investigates the alignment of each symbolic memory access that may resolve outside the enclave. Specifically, in accordance with Intel's intricate software security guidance [26], [40], we validate that every untrusted read or write access resolving outside the enclave is minimally 8-byte aligned, *i.e.*, has the lower three address bits cleared. We, furthermore, ensure that untrusted read accesses have a size that is always *maximally* eight bytes at a time, whereas untrusted writes should be in chunks of *multiples* of eight bytes at a time [26]. Finally, when detecting unaligned untrusted writes, ÆPICSan parses the disassembly of the current basic block to filter out safe cases where the vulnerable write is preceded by the VERW instruction to cleanse leaky microarchitectural buffers and directly followed by an LFENCE; MFENCE instruction pair to avoid inadvertent transient refills, as per Intel's software security guidance [26].

Our complete ÆPICSan validator requires only 103 lines of Python code, where the majority of code concerns parsing the disassembly. This clearly shows the strength of exposing Pandora's enclave-aware memory model (cf. Section 4.3) to individual plugins that may have partially overlapping functionality, e.g., PTRSan vs. ÆPICSan.

The recent SBDR/DRPW disclosures required extensive manual software mitigations, frequently encompassing several rounds of commits and pull requests, throughout the SGX runtime ecosystem. We are the first to provide any form of toolchain support for automatically detecting and validating SGX pointer-alignment considerations, and we are the first to perform a wide-scale investigation of such issues remaining in real-world enclaves (cf. Section 7).

6.4. Control-Flow Hijacking Validation

Lastly, Pandora includes a CFSan plugin, implemented in 110 lines of Python code, that validates enclave control-flow events. This plugin reports insecure jump targets according to the location of the target and whether the target is attacker-tainted.

First, similar to prior work [37], [38], we report a critical security issue when the attacker can arbitrarily control a jump target inside the enclave. Furthermore, similar to the

false-positive heuristic for CFSan, we only report a warning when attacker-tainted jump targets are constrained to always fall entirely inside the enclave.

In addition to this first criterion, partially covered by prior work, CFSan also includes novel rules to detect any enclave jumps to attacker-controlled memory contents. Specifically, we found that several shielding runtimes feature unmeasured and executable memory pages, so as to dynamically load (encrypted) code at runtime. As explained in Section 4.3, this type of enclave memory is not part of the attested MRENCLAVE measurement and is, as such, initially attacker-controlled until it is first initialized by enclave software. Thus, any enclave jumps to unmeasured memory that has not yet been initialized are reported as a critical security issue. While, apart from validation on our own test enclaves, we have not encountered such instances in our evaluation on real-world enclave binaries, we are the first to formulate and write a sanitizer for this nuanced class of novel unmeasured enclave vulnerabilities.

Finally, note that, in line with our goal of truthful symbolic execution, the Pandora base engine already intercepts any jumps to outside the enclave memory range or to non-executable pages inside the enclave, regardless of CFSan. We simply abort the symbolic execution paths for these cases, as both of these events would result in a runtime exception on real SGX hardware and would, hence, not be an exploitable vulnerability besides denial-of-service.

7. Evaluation

We evaluated the efficacy of Pandora and its vulnerability detection plugins in two distinct ways. First, we developed a concise unit-test validation framework, loosely based on the existing Linux selftest enclave [59], to precisely diagnose (known) vulnerabilities in small benchmark enclaves compiled with increasing levels of mitigations. Second, we performed a comprehensive ecosystem analysis on 11 relevant, real-world SGX shielding runtimes, uncovering over 200 newly found vulnerable code locations, tracked via 7 common vulnerabilities and exposure (CVE) identifiers. Additionally, further demonstrating the versatility of Pandora, we made our symbolic-execution tool autonomously reproduce over 69 previously known vulnerable code locations from the literature in older versions of the investigated runtimes.

Table 2 provides an overview of all reported and reproduced issues, whereas a more detailed breakdown is included in Table 4 in Appendix D. Notably, among all the listed vulnerabilities, only one could potentially have been uncovered with existing state-of-the-art SGX symbolic execution tools (cf. Table 4) — due to either lack of support for the required runtime, low-level initialization or entry code, or the specific vulnerability type.

TABLE 2. EVIDENCE OF PANDORA FINDING AND REPRODUCING VULNERABILITIES BOTH IN PRODUCTION AND RESEARCH RUNTIMES.

Runtime	Version	Prod	Src	Plugin	Instances	CVE
Newly found vulnerabilities in shielding runtimes (total 200 instances)						
EnclaveOS	3.28	✓	✗†	ABISan	1	
EnclaveOS	3.28	✓	✗†	PTRSan	15	CVE-2023-38022
EnclaveOS	3.28	✓	✗†	#EPICSan	33	CVE-2023-38021
EnclaveOS	3.28	✓	✗†	CFSan	2	
GoTEE	b35f	✗	✓	PTRSan	31	
GoTEE	b35f	✗	✓	#EPICSan	18	
GoTEE	b35f	✗	✓	CFSan	1	
Gramine	1.4	✓	✓	ABISan	1	
Intel SDK	2.15.1	✓	✓	PTRSan	2	CVE-2022-26509
Intel SDK	2.19	✓	✓	#EPICSan	22	
↪ Occlum	0.29.4	✓	✓	#EPICSan	11	
Linux selftest	5.18	✗	✓	ABISan	1	
↪ DCAP	1.16	✓	✓	ABISan	1	
↪ Inclavare	0.6.2	✗	✓	ABISan	1	
Linux selftest	5.18	✗	✓	PTRSan	5	
↪ DCAP	1.16	✓	✓	PTRSan	17	
↪ Inclavare	0.6.2	✗	✓	PTRSan	2	
Linux selftest	5.18	✗	✓	CFSan	1	
↪ Inclavare	0.6.2	✗	✓	CFSan	1	
Open Enclave	0.19.0	✓	✓	ABISan	2	CVE-2023-37479
Rust EDP	1.71	✓	✓	ABISan	1	
SCONE	5.7 / 5.8	✓	✗	ABISan	2 / 1	CVE-2022-46487
SCONE	5.7 / 5.8	✓	✗	PTRSan	10 / 3	CVE-2022-46486
SCONE	5.7 / 5.8	✓	✗	#EPICSan	11 / 3	CVE-2023-38023
SCONE	5.8	✓	✗	CFSan	1	
Reproduced vulnerabilities in older versions (total 69 instances)						
GoTEE	b35f	✗	✓	ABISan	1	
Gramine	1.2	✓	✓	#EPICSan	10	
Intel SDK	2.1.1	✓	✓	ABISan	1	CVE-2019-14565
Intel SDK	2.13.3	✓	✓	#EPICSan	28	
Open Enclave	0.4.1	✓	✓	ABISan	1	CVE-2019-1370
Open Enclave	0.4.1	✓	✓	PTRSan	13	CVE-2019-0876
Open Enclave	0.4.1	✓	✓	#EPICSan	13	
Rust EDP	1.63	✓	✓	#EPICSan	2	

Legend: † Source code was made privately available; ↪ Based on above runtime.

7.1. Selftest Validation Framework

The Linux kernel natively includes drivers for Intel SGX since the 5.11 release [59]. As part of this effort, Linux also contains a bare-metal selftest enclave that provides a minimal example to test the loading and execution of an enclave binary without relying on any particular SGX shielding runtime. This Linux selftest enclave consists of hand-crafted assembly routines for entry and exit, plus an ecall dispatcher that calls C functions. While this selftest enclave is not intended to be a production runtime, Linux developers have noted that its code may be copied and provides a "great starting point if you want to do things from scratch" [60]. Indeed, we found that at least two real-world SGX projects directly built on the Linux selftest enclave to date: Alibaba Inclavare Containers [61] uses it as a skeleton example of best-practice enclave runtime integration and Intel's Data Center Attestation Primitives (DCAP) [62] for Windows more critically uses it as the base for a custom launch enclave that gets access to an SGX platform-specific cryptographic key to decide which application enclaves can be ran on the system. We report Pandora's findings on these bare-metal enclaves in the next section.

We developed a unit-test framework based on the Linux selftest enclave. This test suite contains individually crafted enclave binaries featuring multiple levels of ABI register cleansing and input pointer(-to-pointer) sanitizations. These enclaves, thus, provide a controlled test environment to craft

arbitrarily complex and challenging scenarios to validate the efficacy of our plugins and Pandora's enclave-aware symbolic memory model. Furthermore, they allow to prototype conceivable vulnerabilities that have not (yet) been encountered "in the wild", e.g., jumps to unmeasured and uninitialized pages (cf. Section 6.4).

7.2. SGX Runtime Ecosystem Analysis

Runtime Selection. To explore the vulnerability landscape for real-world enclave software, we evaluated Pandora on a diverse set of 8 production-quality and 3 research-grade Intel SGX shielding runtimes. Note that, as discussed in Section 3, we opted to focus on validating the vital enclave shielding runtime itself, including indispensable, low-level initialization and entry code, rather than the more accessible challenge of validating higher-level application logic as explored in complementary prior work [34], [37], [38]. While the latter typically only affects a single (research) application that makes incorrect use of shielding abstractions, e.g., unchecked `user_check` pointers [4], [5], production-quality shielding runtimes are supposed to be thoroughly vetted and any vulnerabilities found would affect universally all applications developed on top.

Our runtime selection includes diverse enclave programming paradigms, including 2 SDKs (Intel SGX SDK [4] and Microsoft Open Enclave [5]), 4 libOSs (EnclaveOS [63], SCONE [64], Occlum [10], and Gramine [65]), 2 secured language runtimes (Rust-EDP [11] and Go-TEE [12]), and 3 bare-metal enclaves (Linux selftest [59], Inclavare [61], and DCAP [62]). We included the bare-metal enclaves, as well as the academic Go-TEE research prototype runtime, to complement the insights from the more mature production ecosystem. Furthermore, while the majority of SGX shielding runtimes are developed as open-source software, our selection also includes two proprietary runtimes: EnclaveOS, with source code privately provided by the vendor, and SCONE, with only binaries available.

Due to the intricacies involved in building old runtime versions with often complex dependencies, we opted to limit our choice of known vulnerabilities to a representative sample across major runtimes. We see a systematic overview of the vulnerability landscape of past runtimes as an interesting and feasible direction for future work and believe that Pandora could aid in such a survey. In the following, after describing our experimental setup, we highlight the most interesting findings of each plugin.

Experimental Setup. We extracted exact enclave dumps via SGX-TRACER and ran Pandora on all runtimes with a time budget of 12 hours and a memory budget of 256 GB, whichever occurred first. Cloud instances with such memory budget are commercially available beginning at 4 $ per hour, making this limit feasible for occasional extensive validation with Pandora, e.g., as part of continuous integration (CI) for releases (as at least one vendor privately expressed interest in). Each runtime was explored twice: once with a default breadth-first exploration strategy and once with a depth-first strategy that eagerly followed the longest paths.

We note that in our experiments, the 256 GB memory limit was only hit twice, namely for the Intel SGX SDK 2.19 when using breadth-first search after approximately 8 hours, and for GoTEE as the enclave memory dump is exceedingly large at 64 GB. In all other cases, the memory consumption varied between 24.6 GB and 196.7 GB for breadth-first search and from 4.9 GB to 154.7 GB for depth-first search.

In some rare cases, our Pandora prototype crashed before reaching these limits due to remaining unsupported x86 instructions or due to crashes in the underlying angr and z3 solver. For EnclaveOS specifically, we manually guided Pandora to skip two functions that either contain still unsupported AES-NI instructions, or execute a waiting loop that expects a second thread to fill data before continuing. For the DCAP bare-metal launch enclave, we similarly instrumented Pandora to skip two functions with unsupported AES-NI instructions.

7.2.1. ABI Sanitization Issues. Following a recent overview study [41], Pandora promptly confirmed known ABI issues in older Intel SGX SDK and Open Enclave binaries, which have since been evidently mitigated (cf. Table 2). Nonetheless, Pandora found that the proprietary SCONE runtime still lacked any sanitization code for x87 and SSE floating-point configuration registers. We experimentally demonstrated that this lack of ABI sanitization, can be exploited in practice via a proof-of-concept exploit that successfully introduces rounding errors in an elementary "sconified" floating-point application. Following our responsible disclosure, tracked under CVE-2022-46487, these issues have been patched in the latest SCONE release 5.8.0.

Additionally, `ABISan` found that the academic GoTEE runtime, as well as the Linux selftest, Inclavare, and DCAP bare-metal enclaves, universally lack ABI entry sanitizations for RFLAGS and floating-point configuration registers. Interestingly, Inclavare and DCAP took care to cleanse extended processor state on enclave exit, but not on entry. Highlighting the strength of `ABISan`'s taint policy, the plugin autonomously discovered attacker-tainted reads from the x86 direction flag for compiler-emitted `REP` string instructions that could be fatally corrupted in the DCAP launch enclave, and notably found that GoTEE even lacks secure stack pointer initialization, which could be exploited to obtain full code execution in this runtime (cf. as also reported by both `CFSan` and `PTRSan`). The issues in DCAP are mitigated in version 1.19 and onward.

Our systematic analysis, furthermore, identified an interesting case of *regression* in Open Enclave, which was assigned CVE-2023-37479 by Microsoft and mitigated in release 0.19.3. Particularly, in response to prior research [21], commit `efe7504` in Open Enclave included a patch to properly sanitize the x86 alignment-check flag. However, `ABISan` discovered that in current versions of Open Enclave, the alignment-check flag was no longer properly sanitized after the initial enclave sanitization routines have completed. Upon further investigation, we were able to conclude

that Open Enclave accidentally reintroduced the once-fixed vulnerability with commit `16efbd6` in 2021, in a patch set to mitigate another attack [23] that places more stringent demands on stack-pointer initialization for exception handlers. This instance of unintended regression thus provides a clear illustration of the complexity of shielding responsibilities and the potential value of including an automated tool like Pandora in CI pipelines to test against known vulnerabilities before releasing new software versions.

A final and particularly widespread line of ABI sanitization issues follows from Intel's recent MXCSR configuration-dependent timing (MCDT) software guidance [27]. Particularly, Intel recommends that shielding runtimes set all floating-point exception status flags in the MXCSR register for the lifetime of the enclave to avoid subtle, operand-dependent timing differences in otherwise constant-time code on affected processors. Notably, this refined guidance did not result from an academic publication or security advisory and may have been easily missed by runtime developers. Indeed, `ABISan` detected that only the Intel SGX SDK and the dependent Occlum runtime properly set MXCSR according to the new recommendation, and *all* other runtimes did not. Following our disclosure, this has since been patched in Open Enclave (0.19.3), Rust-EDP (1.71.0), and EnclaveOS (3.30), and will be patched in the upcoming SCONE 5.9.0 release.

7.2.2. Pointer Sanitization Issues. The strength of the `PTRSan` plugin is to rigorously investigate issues with pointer dereferences across many enclave runtimes.

In the SCONE production runtime, `PTRSan` uncovered 10 unique critical issues: 8 entirely unconstrained, attacker-tainted pointer dereferences and 2 untainted outside-enclave reads. Although the source code was not available, Pandora was able to generate precise basic-block backtraces annotated with ELF symbols, aiding in our investigation and even the development of proof-of-concept exploits. We reported each issue, tracked as a bundle under CVE-2022-46486, to the SCONE developers who confirmed our findings and included patches in the latest release 5.8.0. However, `PTRSan`'s subsequent analysis on SCONE 5.8.0 revealed two more remaining vulnerabilities: an entirely unconstrained attacker-tainted pointer and an untainted outside enclave read, to be mitigated in the upcoming 5.9.0 release.

In EnclaveOS, `PTRSan` was able to detect a particularly subtle instance of an untrusted pointer dereference as part of a string length calculation, which is logically correct but can be abused as a capable side-channel oracle to precisely locate all null bytes in enclave memory [21]. Fortanix gave a *high* severity rating for this finding, tracked under CVE-2023-38022, and mitigated it in version 3.29. As a second notable finding in EnclaveOS, Pandora autonomously detected that overflow protections were missing in the untrusted pointer validation logic of the enclave binary. Upon closer examination, we found that existing source-level overflow checks were silently optimized away by the compiler. Specifically, the source code utilized `void*` pointer arithmetic, which, unfortunately, is undefined behavior in `C`, leading to the compiler removing this check completely. Pandora correctly reported that, with this check missing, the attacker can cause untrusted pointers to wrap the address space via an unsigned integer overflow. This issue highlights the strength of Pandora's binary-level validation and accurate symbolic constraint solving of not only untrusted pointer values but also their sizes, and is also mitigated in version 3.29.

Furthermore, as part of this research, `PTRSan` additionally confirmed an untrusted pointer dereference in the protected code loader of the Intel SGX SDK version 2.15.1, tracked via CVE-2022-26509 and patched in later versions. This issue underlines the importance of validating low-level runtime initialization code, as this pointer check was missing before any in-enclave relocations, including global variables containing the enclave base address and size needed in the validation function itself, had been performed.

In the GoTEE research runtime, `PTRSan` discovered numerous (31) unconstrained pointer dereferences, highlighting that even safe languages are not immune to oversights in pointer validation for SGX's unique attacker model. Furthermore, all bare-metal enclaves were found especially vulnerable without any pointer sanitization measures (as reported both by `PTRSan` and `ÆPICSan`). For the DCAP launch enclave, Pandora reported 17 unique critical issues, of which 11 were unconstrained, attacker-controlled reads and 6 were unconstrained writes to arbitrary in-enclave locations (mitigated in version 1.19). Likewise, the Inclavare enclave contains several vulnerable invocations of `memcpy` with unconstrained source and destination parameters, and the Linux selftest enclave contains 5 entirely unconstrained, attacker-tainted pointer dereference locations that can be trivially exploited to leak or corrupt arbitrary in-enclave memory locations.

Finally, for the known-vulnerable version 0.4.1 of Microsoft Open Enclave, Pandora correctly identified CVE-2019-0876 [21], which highlights the power of multiple reentries, as the vulnerability can only be triggered after the enclave has been initialized. In addition, `PTRSan` also reported a (presumably unknown) issue in this old runtime version, indicating a lack of pointer sanitization in the `oe_initialize_cpuid()` function.

7.2.3. *ÆPIC* Sanitization Issues. Pandora is the first tool to support automated analysis and validation of ÆPIC-style untrusted pointer alignment vulnerabilities in SGX enclaves. We, thus, employed our novel `ÆPICSan` plugin to perform a large-scale, automated analysis to assess the completeness of Intel's particularly complex and error-prone software mitigation guidelines [26], [40] in real-world enclave shielding runtimes. As result of this systematic analysis, Pandora found that SBDR and DRPW mitigations were missing entirely in GoTEE (18 unique instances), SCONE (11 instances; tracked via CVE-2022-46487 and partially mitigated in version 5.8.0), and EnclaveOS (33 instances; tracked via CVE-2023-38021 and mitigated in release 3.32). Furthermore, when analyzing the latest SCONE 5.8.0 release, Pandora found that the in-enclave `memcpy` function was not properly patched to

exclude SBDR issues which will be fixed in 5.9.0. Existing mitigations in Gramine, Rust-EDP, and Open Enclave were found sufficient, but ÆPICSan autonomously discovered a missing SBDR sanitization in the enclave initialization phase of the latest version of the Intel SGX SDK (also inherited by the derived Occlum runtime), highlighting that adequately restricting untrusted pointer alignments is challenging even for mature runtime developers.

As expected, we additionally confirmed that ÆPICSan can automatically reproduce ample SBDR and DRPW issues in older versions of Gramine, Rust-EDP, Open Enclave, and the Intel SGX SDK without mitigations.

7.2.4. Control Flow Issues. The CFSan plugin found a delicate issue in EnclaveOS where the global offset table (GOT) is incorrectly accessed before relocation of the enclave has completed. The GOT is used to jump to functions in position-independent code and has to be securely initialized, *i.e.*, relocated, before it can be used inside enclaves. The issue found by Pandora, and confirmed and fixed by Fortanix in version 3.31, concerns an unusual trace where an error occurs during initialization, which results in the code calling a debug logging function. A similarly evasive GOT relocation issue in an early-error path was reported by CFSan for SCONE 5.8.0, to be patched in 5.9.0.

Furthermore, CFSan found that Inclavare's bare-metal enclave assembly entry stub incorrectly uses a signed JGE x86 jump instruction, instead of a proper unsigned JAE condition to sanitize the attacker-provided index into the ecall function-pointer table. Critically, this subtle oversight ultimately allows *arbitrary* control-flow hijacking by passing a large negative index into the ecall table and loading the function pointer from untrusted, attacker-controlled memory. Likewise, CFSan found that, depending on the optimization level, in-enclave relocation code for the ecall table was missing in the dispatcher of the Linux selftest enclave.

Finally, due to the lack of secure stack switching in GoTEE, CFSan reported unconstrained RET targets.

8. Discussion

We see Pandora as a mature prototype of an enclave-aware symbolic execution tool that can serve as a basis for future science. In particular, we designed Pandora with great care for usability, through a well-documented command line interface and detailed HTML reports, and reusability through our plugin-based approach that makes it easy to implement additional security analyses. Pandora has demonstrated its usefulness by automatically finding vulnerabilities in production runtimes. Hence, we believe that Pandora is a valuable step forward in vulnerability detection for enclaves.

Coverage. We consider the main limitation of Pandora to be incomplete coverage, *i.e.*, the infeasibility to explore a binary as a whole with symbolic execution. This is due to the fact that Pandora, as any symbolic-execution tool, suffers from the well-known limitation of state explosion, which can make exhaustive exploration of larger binaries practically infeasible. Hence, vulnerabilities can still remain undetected in unexplored paths. We implemented novel, enclave-aware performance optimizations, including uninitialized memory and state-uniqueness reductions (cf. Section 4.5), and we utilized both breadth-first and depth-first exploration in our evaluation to cover more enclave behavior.

Our choice for angr [52] as the underlying symbolic-execution engine may also in itself be a source of incomplete coverage, as angr is not guaranteed to be sound and may concretize values during symbolic execution. To avoid missing program behavior, we adopted the most conservative approach whenever possible and tried to refrain from unnecessary concretization of symbolic values. Despite these limitations, angr is particularly powerful for rapid development of vulnerability plugins in comparison to fully fledged code verification tools.

A further possible technical, but not inherent, limitation concerns Pandora's coverage of any encrypted code that would be loaded at runtime to execute a confidential enclave application. Such code could be transparently supported by providing Pandora with the decryption key, which could then be used by the symbolic execution engine to automatically decrypt and execute the code. That being said, the primary focus of Pandora are runtimes, which are usually not utilizing such encrypted code loading themselves.

We consider the fact that Pandora was able to automatically uncover vulnerabilities in production runtimes as clear evidence for the practicality of our approach to validate enclave shielding runtimes. The (orthogonal) extension of Pandora's truthful enclave-aware symbolic-exploration to also analyze arbitrary (and potentially larger and deeper) enclave application logic would require further scaling that could conceivably benefit from optimizations proposed in previous work [37], [39].

Accuracy. As any automatic vulnerability scanner, Pandora may report false-positive issues, which could lead to overly exhaustive outputs. We attempt to limit the strain on the human analyst via two steps. First, potential issues are classified into multiple levels of criticality, and the reports are formatted in modern HTML forms that allow to filter criticality levels. Second, plugins may downgrade the severity of issues via sensible heuristics, e.g., Section 6.2 explained how PTRSan downgrades attacker-tainted pointers when they are constrained to a region entirely inside the enclave, closely resembling the benign pattern of an attacker-controlled index in a trusted enclave buffer. All *critical* issues found by Pandora listed in Table 2 were reported to the vendors who acknowledged the vulnerabilities. Hence, we are not aware of any false-positive results for these critical issues. Beyond this, Pandora heuristically downgraded 124 of 452 (27 %) vulnerabilities to warnings, where we are not aware of any of those being exploitable.

Regarding false negatives, there is unfortunately no standardized ground truth of existing vulnerabilities for Intel SGX runtimes, and a direct comparison of Pandora to related approaches is not feasible as their target (*i.e.*, enclave application logic) is orthogonal. Therefore, we followed a

best-effort approach and let Pandora successfully reproduce known runtime vulnerabilities (cf. Table 2). Our analysis clearly shows that Pandora reproduced all known vulnerabilities from selected work [21], [22] and even found an overlooked issue (cf. §7.2.2). We consider the main limitation to be incomplete coverage, which may lead to vulnerabilities on unexplored paths not being detected (e.g., CVE-2021-44421 in Occlum).

Future Work. Potential future extensions of Pandora concern novel vulnerability-detection plugins, as well as the investigation of transient execution access patterns in enclaves [66]–[68]. Furthermore, we see Pandora as a useful tool for a broad ecosystem analysis of the Intel SGX landscape and how fast vulnerability patches propagate across runtimes. Ultimately, future work could even explore automated exploit generation and binary patching using Pandora's precise vulnerability reports.

There are additionally some performance improvements that could allow Pandora to explore enclaves in more depth. While we already implemented a depth-first extension to Pandora that severely limits the memory use necessary during exploration, angr still only uses one single CPU core. Future work could thus investigate how angr symbolic exploration can be split up onto multiple cores while retaining the same enclave-aware characteristics of Pandora that are necessary to e.g., identify enclave boundaries. Additionally, to mitigate path explosion, we could also adopt state-merging [69] or path prioritization strategies [49], [70].

9. Conclusion

In recent years, a sizable ecosystem of Intel SGX enclave shielding runtimes has emerged. However, writing secure SGX software has proven to be particularly challenging due to the moving nature of the threat landscape, and not even well-designed and vetted shielding runtimes have been immune to missing nuanced attack vectors or to reintroducing already known vulnerabilities into their code. The research community has only recently started to look into SGX-aware symbolic execution, but has focused on application logic only, while largely skipping the crucial enclave shielding runtime itself. In this work, we presented Pandora, the first enclave-aware and pluggable symbolic-execution tool that allows *truthfully* validating arbitrary enclave binaries, including low-level runtime initialization and entry phases. With 4 diverse prototype plugins, we found 200 new and 69 known vulnerable code locations across a wide selection of 11 SGX runtimes. Ultimately, we envision Pandora not only as a practical validation tool for real-world enclave runtimes today, but also as a solid, extensible and open-source foundation for future science on SGX software validation of enclave shielding runtimes.

Acknowledgments. This research is partially funded by grants of the Research Foundation – Flanders (FWO), under grant numbers 11E5120N, 1261222N, 12B2A24N and G081322N, and by the Flemish Research Programme Cybersecurity. This research was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/V000454/1, and EP/S030867/1. The results feed into DsbDtech. Some computations described in this paper were performed using the University of Birmingham's BlueBEAR HPC service, which provides a High Performance Computing service to the University's research community.

References

[1] V. Costan and S. Devadas, "Intel SGX explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, 2016.

[2] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

[3] Intel, "Intel Trust Domain Extensions," Feb. 2022. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/690419

[4] ——, "Intel Software Guard Extensions – Get Started with the SDK," 2023. [Online]. Available: https://software.intel.com/en-us/sgx/sdk

[5] Microsoft, "Open Enclave SDK," 2023. [Online]. Available: https://openenclave.io/

[6] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2014.

[7] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell *et al.*, "SCONE: Secure Linux containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016.

[8] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *USENIX Annual Technical Conference (ATC)*, 2017.

[9] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "SGX-LKL: securing the host OS interface for trusted execution," *arXiv preprint arXiv:1908.11143*, 2019.

[10] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[11] Fortanix, "Fortanix enclave development platform – rust edp," 2023. [Online]. Available: https://edp.fortanix.com/

[12] A. Ghosn, J. R. Larus, and E. Bugnion, "Secured routines: Language-based construction of trusted execution environments," in *USENIX Annual Technical Conference (ATC)*, 2019.

[13] Enarx Project, "Enarx: Webassembly + confidential computing," https://enarx.dev/, 2023.

[14] Edgeless Systems, "Edgeless RT," https://github.com/edgelesssys/edgelessrt, 2022.

[15] A. Nilsson, P. N. Bideh, and J. Brorsson, "A survey of published attacks on intel sgx," *arXiv preprint arXiv:2006.13598*, 2020.

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

[17] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS'19)*. ACM, Nov. 2019.

[18] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *Proceedings of the 41th IEEE Symposium on Security and Privacy (S&P'20)*, May 2020.

[19] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.

[20] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre attacks: Stealing Intel secrets from SGX enclaves via speculative execution," in *4th IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE, 2019.

[21] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *26th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2019.

[22] F. Alder, J. Van Bulck, D. Oswald, and F. Piessens, "Faulty point unit: ABI poisoning attacks on Intel SGX," in *36th Annual Computer Security Applications Conference (ACSAC)*, Dec. 2020.

[23] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, "Smashex: Smashing sgx enclaves using exceptions," in *28th ACM Conference on Computer and Communications Security (CCS)*, 2021.

[24] Intel Corporation, "Deep dive: Load value injection," 2020.

[25] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC Leak: Architecturally leaking uninitialized data from the microarchitecture," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[26] Intel, "Processor MMIO stale data vulnerabilities," June 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/processor-mmio-stale-data-vulnerabilities.html

[27] ——, "Mxcsr configuration dependent timing," Aug. 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/mxcsr-configuration-dependent-timing.html

[28] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking transient execution through microarchitectural load value injection," in *41st IEEE Symposium on Security and Privacy (S&P'20)*, May 2020.

[29] A. Kogler, D. Gruss, and M. Schwarz, "Minefield: A software-only protection for SGX enclaves against DVFS attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[30] L. Giner, A. Kogler, C. Canella, M. Schwarz, and D. Gruss, "Repurposing segmentation as a practical LVI-NULL mitigation in SGX," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[31] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[32] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd, "Mitigating branch-shadowing attacks on intel sgx using control flow randomization," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018.

[33] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "Dr. SGX: automated and adjustable side-channel protection for SGX using data location randomization," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.

[34] T. Cloosters, J. Willbold, T. Holz, and L. Davi, "SGXFuzz: Efficiently synthesizing nested structures for SGX enclave fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[35] R. Cui, L. Zhao, and D. Lie, "Emilia: Catching iago in legacy code." in *NDSS*, 2021.

[36] M. Orenbach, B. Raveh, A. Berkenstadt, Y. Michalevsky, S. Itzhaky, and M. Silberstein, "Securing access to untrusted services from TEEs with GateKeeper," *arXiv preprint arXiv:2211.07185*, 2022.

[37] T. Cloosters, M. Rodler, and L. Davi, "Teerex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[38] P. Antonino, W. A. Woloszyn, and A. Roscoe, "Guardian: Symbolic validation of orderliness in sgx enclaves," in *Proceedings of the 2021 on Cloud Computing Security Workshop*, 2021.

[39] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[40] Intel, "Stale data read from legacy xAPIC," Aug. 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/stale-data-read-from-xapic.html

[41] J. Van Bulck, F. Alder, and F. Piessens, "A case for unified ABI shielding in Intel SGX runtimes," in *5th Workshop on System Software for Trusted Execution (SysTEX)*. ACM, Mar. 2022.

[42] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, 1976.

[43] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, 2012.

[44] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012.

[45] T. Yavuz, F. Fowze, G. Hernandez, K. Y. Bai, K. R. Butler, and D. J. Tian, "ENCIDER: Detecting Timing and Cache Side Channels in SGX Enclaves and Cryptographic APIs," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[46] G. Duan, Y. Fu, B. Zhang, P. Deng, J. Sun, H. Chen, and Z. Chen, "Teefuzzer: A fuzzing framework for trusted execution environments with heuristic seed mutation," *Future Generation Computer Systems*, 2023.

[47] A. Khan, M. Zou, K. Kim, D. Xu, A. Bianchi, and D. J. Tian, "Fuzzing sgx enclaves via host program mutations," in *8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023.

[48] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[49] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[50] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.

[51] Y. Wang, Z. Zhang, N. He, Z. Zhong, S. Guo, Q. Bao, D. Li, Y. Guo, and X. Chen, "Symgx: Detecting cross-boundary pointer vulnerabilities of sgx applications via static symbolic execution," in *30th ACM Conference on Computer and Communications Security (CCS)*, 2023, p. 2710–2724.

[52] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *Cybersecurity Development (SecDev)*. IEEE, 2017.

[53] Intel Corporation, *Intel 64 and IA-32 architectures software developer's manual*, 2020, reference no. 325462-062US.

[54] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld, "Explicit secrecy: A policy for taint tracking," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016.

[55] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *15th USENIX Security Symposium*, 2006.

[56] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *2nd Workshop on System Software for Trusted Execution (SysTEX 2017)*. ACM, Oct. 2017.

[57] H. Lu, D. L. Kreitzer, M. Girkar, and Z. Ansari, "System V application binary interface," *Intel386 Architecture Processor Supplement, Version 1.1*, December 2015.

[58] A. Fog, "Calling conventions for different c++ compilers and operating systems," http://www.agner.org/optimize/calling_conventions.pdf, Apr. 2018.

[59] L. Torvalds, "Linux operating system," kernel.org, 2023.

[60] J. Sakkinen and N. McCallum, "selftests/x86: Add a selftest for sgx," Mar. 2020. [Online]. Available: https://lkml.kernel.org/lkml/04362c0cf66bf66e8f7c25a531830b9f294d2d09.camel@linux.intel.com/

[61] AliBaba, "Inclavare containers: The future of cloud-native confidential computing," Jun. 2022. [Online]. Available: https://www.alibabacloud.com/blog/inclavare-containers-the-future-of-cloud-native-confidential-computing_598992

[62] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, "Supporting third party attestation for Intel SGX with Intel data center attestation primitives," White paper, 2018.

[63] Fortanix, "Fortanix runtime encryption platform and enclaveos," https://www.fortanix.com/platform/runtime-encryption, 2023.

[64] Scontain GmbH, "Scone – a secure container environment," 2023. [Online]. Available: https://scontain.com/

[65] The Gramine Workgroup, "Gramine – a library os for unmodified applications," https://gramineproject.io/, 2023.

[66] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *IEEE Symposium on Security and Privacy*. IEEE, 2020.

[67] L. Daniel, S. Bardin, and T. Rezk, "Hunting the haunter - efficient relational symbolic execution for spectre with haunted relse," in *NDSS*. The Internet Society, 2021.

[68] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *PLDI*. ACM, 2020.

[69] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *PLDI*. ACM, 2012.

[70] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," *SIGPLAN Not.*, vol. 48, no. 10, oct 2013.

Appendix A.
Pandora CLI and Report Generation (*D1*)

We designed Pandora with great care for usability (*D1*), through a well-documented command line interface (CLI) and detailed HTML reports. Figure 3 shows an example of a human-readable, interactive HTML report from the `PTRSan` plugin discovering unconstrained pointer dereferences in the Linux selftest enclave (cf. Section 7.1). Figure 4 shows a part of the interactive command line interface of Pandora, which is intended to be highly usable for both rapid prototyping of new plugins and for long, unattended exploration runs. Issues are reported on the command line during a run, but are also logged in a JSON file that can later be expanded into the fully-fledged HTML reports visible in Figure 3. If the human analyst wishes, multiple breakpoints regarding

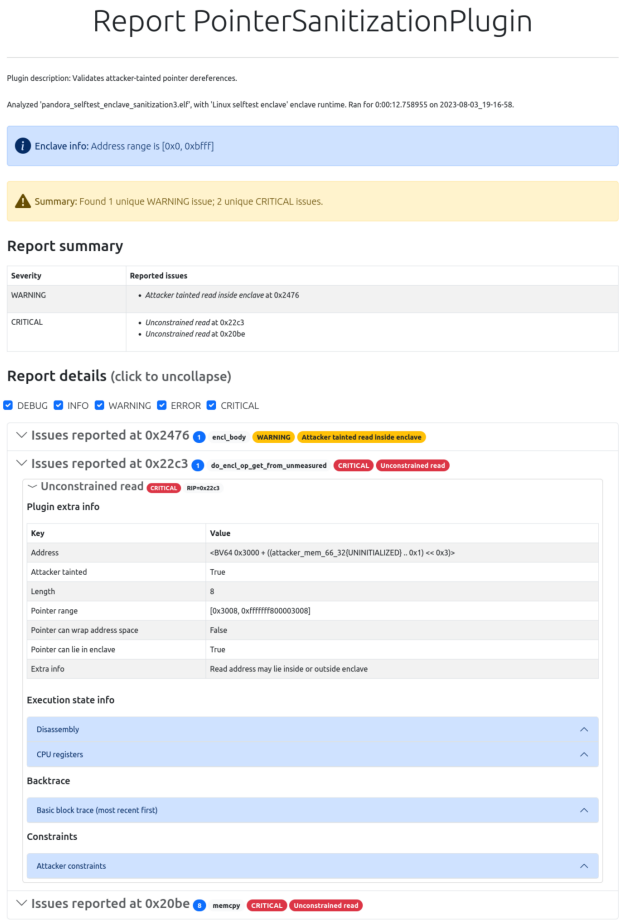

Figure 3. Example of an HTML report generated by Pandora.

the exploration and the plugins are readily available from the command line, *i.e.*, to interrupt execution on interesting events and switch into a Python shell. This allows to quickly implement and troubleshoot plugins. Lastly, several options of Pandora are, in addition to the CLI, exposed via configuration files, allowing to define long-lasting analysis setups that can be controlled by changing few program options.

We leave it for future work to rigorously investigate to what extent Pandora achieved *D1*, e.g., by means of a comprehensive and unbiased user study. Such a user study should investigate whether the reporting generated by Pandora is factually useful for a human analyst, and perform a quantitative analysis on the benefit of additional CLI and reporting features.

Appendix B.
Static Analysis of Enclave Runtimes

This appendix describes optional support we added to Pandora to load enclave binaries from selected runtimes with purely static analysis only, *i.e.*, without first requiring the SGX-TRACER dynamic memory extraction phase described

Figure 4. Part of the command line interface of Pandora depicting helpful command options to the user.

in Section 5. The difficulty in adequately supporting arbitrary enclave runtimes in this way lies in parsing opaque enclave memory layout metadata from the binary and loading SGX-specific data structures into the symbolic execution memory after the ELF file has been loaded. Although inherently fragile and version-specific, we show that it is in principle possible to implement such support entirely statically for three exemplary runtime loaders.

While we consider some of these static loaders to be mature and satisfying our truthful initial memory layout criterion (*G1b*), we note that this highly labor-intensive static-analysis approach is evidently not runtime-agnostic (vs. *G2*). Furthermore, even for the individual runtimes that are supported, the static-analysis approach remains inherently fragile, as new versions of these runtimes may completely break or change the way runtime-specific data structures are utilized in the enclave.[4] Thus, we use our novel SGX-TRACER enclave memory extractor approach as the default runtime-agnostic and truthful loader in Pandora, as also used in the evaluation of Section 7.

Linux Selftest Enclave. First, the Linux selftest enclave [59] is a minimal, self-contained enclave that has a fixed memory layout, with the TCS always being stored at the start of the enclave range. This makes it an ideal baseline runtime as no enclave initialization is necessary and all relevant addresses are statically known at compile time. The Linux selftest enclave serves as the foundation for Pandora's unit-test validation framework, discussed in Section 7.1.

Intel SGX SDK. Second, the Intel SGX SDK [4] encodes all information for the loading process in an additional ELF metadata section. Based on manual analysis of the open-source code of the Intel SGX SDK enclave loader, we added full support in Pandora to decode this opaque blob and extract the expected locations of TCSs, stack and heap regions, and patches to initialize enclave global data structures. We implemented mature support to perform these steps in Intel SGX SDK version 2.18.1 and also validated backwards compatibility and added support for version-specific fields in versions 2.18 and 2.17.1.

SCONE. Lastly, we show that, in principle, the static enclave loading approach is even feasible without access to source code by implementing an elementary (incomplete) static loader for the proprietary SCONE [7] runtime. Specifically, we manually reverse engineered the enclave layout and location of TCS data structures and thread-local memory using a debugger. Based on this partial layout, our static loader inserts the required data structures into the symbolic memory layout when loading the SCONE runtime binary ELF file.

Appendix C. Pandora Breakpoints

Table 3 lists all enclave-aware breakpoints added by Pandora. To accommodate various investigation scenarios, all breakpoints can be triggered before and after the event happened, *i.e.*, to investigate an event both before or after it had an impact on a Pandora state. For example, Pandora memory read breakpoint, similarly to angr memory read breakpoints, can be triggered before the read has happened, exposing, among other, its address and size; or after the read has happened, additionally exposing its value.

Appendix D. Vulnerability Details

Table 4 provides a more detailed breakdown of the vulnerable code locations found by Pandora, as also summarized in Table 2 and discussed in Section 7.

4. Examples of such changes in the past were versions 2.4, 2.14, and 2.17 of the Intel SGX SDK when the internal `_global_data_t` C structure was modified which resulted in altered offsets for the address of the enclave base address, a crucial piece of information to properly resolve addresses inside the enclave.

TABLE 3. LIST OF BREAKPOINTS ADDED BY PANDORA. PLUGINS CAN HOOK THESE NEW BREAKPOINTS, IN ADDITION TO ALL LEGACY ANGR BREAKPOINTS, TO INVESTIGATE SPECIFIC EVENTS DURING EXPLORATION. ALL EVENTS CAN BE HOOKED BEFORE AND AFTER THEY ARE EXPLORED. INDIVIDUAL BREAKPOINTS MAY ADDITIONALLY EXPOSE SPECIFIC ARGUMENTS, E.G., SYMBOLIC MEMORY ADDRESSES AND SIZES.

Breakpoint event	Triggered by Pandora module	Description
`eenter`	Enclave (Re)entry	A state is prepared to (re)enter the enclave
`eexit`	SGX Instructions	An `EEXIT ENCLU` is executed
`untrusted_mem_read`	Enclave Memory	Reads that fully lie in untrusted memory
`trusted_mem_read`	Enclave Memory	Reads that fully lie in enclave memory
`inside_or_outside_mem_read`	Enclave Memory	Reads that may lie in either region
`untrusted_mem_write`	Enclave Memory	Writes that fully lie in untrusted memory
`trusted_mem_write`	Enclave Memory	Writes that fully lie in enclave memory
`inside_or_outside_mem_write`	Enclave Memory	Writes that may lie in either region

TABLE 4. DETAILED EVIDENCE OF PANDORA FINDING AND REPRODUCING VULNERABILITIES BOTH IN PRODUCTION AND RESEARCH RUNTIMES, WHERE THE "DEPTH" COLUMN LISTS THE NUMBER OF BASIC BLOCKS EXPLORED BEFORE THE VULNERABILITY (MIN–MAX); "L" INDICATES THE LOCATION (ENTRY, INITIALIZATION, APPLICATION) OF THE VULNERABILITY; AND COLUMN "O" INDICATES WHETHER THE VULNERABILITY COULD HAVE BEEN FOUND BY EXISTING, STATE-OF-THE-ART SGX SYMBOLIC-EXECUTION TOOLS [37], [38].

Runtime	Version	Prod	Src	Plugin	L	Depth	Instances	Description	O
Newly found vulnerabilities in shielding runtimes (total 200 instances)									
EnclaveOS	3.28	✓	✗†	`ABISan`	E	8	1	MXCSR dependent timing	✗
EnclaveOS	3.28	✓	✗†	`PTRSan`	E	14–48	10	Compiler removed overflow check	✗
EnclaveOS	3.28	✓	✗†	`PTRSan`	I	15495–15521	5	`strlen` on unconstrained ptr (CVE-2023-38022)	✗
EnclaveOS	3.28	✓	✗†	`ÆPICSan`	I	14–100	33	Various SBDR issues (CVE-2023-38021)	✗
EnclaveOS	3.28	✓	✗†	`CFSan`	I	51	2	PIC jump before relocation	✗
GoTEE	014b35f	✗	✓	`PTRSan`	E/I	2–82	31	Various unconstrained pointers	✗
GoTEE	014b35f	✗	✓	`ÆPICSan`	E/I	2–82	18	Various SBDR/DRPW issues	✗
GoTEE	014b35f	✗	✓	`CFSan`	I	82	1	Unconstrained `RET` targets	✗
Gramine	1.4	✓	✓	`ABISan`	E	8	1	MXCSR dependent timing	✗
Intel SDK	2.15.1	✓	✓	`PTRSan`	I	29–30	2	Unconstrained pointer (CVE-2022-26509)	✗
Intel SDK	2.19	✓	✓	`ÆPICSan`	I	234	22	SBDR in enclave initialization	✗
↪ Occlum	0.29.4	✓	✓	`ÆPICSan`	I	17222	11	↪ SBDR inherited	✗
Linux selftest	5.18	✗	✓	`ABISan`	E	1	1	Unsanitized AC/DF, MXCSR, and FPU	✗
↪ DCAP	1.16	✓	✓	`ABISan`	E	1	1	↪ Missing sanitization on entry	✗
↪ Inclavare	0.6.2	✗	✓	`ABISan`	E	1	1	↪ Missing sanitization on entry	✗
Linux selftest	5.18	✗	✓	`PTRSan`	A	4–7	5	Various unconstrained pointers	✗
↪ DCAP	1.16	✓	✓	`PTRSan`	A	3–1075	17	Various unconstrained pointers	✗
↪ Inclavare	0.6.2	✗	✓	`PTRSan`	A	8–539	2	Unconstrained src/dst addresses in `memcpy`	✗
Linux selftest	5.18	✗	✓	`CFSan`	A	5	1	PIC jump before relocation	✗
↪ Inclavare	0.6.2	✗	✓	`CFSan`	E	3	1	Unsigned jump target comparison in `ecall` array	✗
Open Enclave	0.19.0	✓	✓	`ABISan`	E	11	1	Unsanitized AC (regression) (CVE-2023-37479)	✗
Open Enclave	0.19.0	✓	✓	`ABISan`	E	11	1	MXCSR dependent timing	✗
Rust EDP	1.71	✓	✓	`ABISan`	E	7	1	MXCSR dependent timing	✗
SCONE	5.7.0	✓	✗	`ABISan`	E	3	1	Unsanitized FPU (CVE-2022-46487)	✗
SCONE	5.7.0	✓	✗	`PTRSan`	I	25–1827	10	Various pointer issues (CVE-2022-46486)	✗
SCONE	5.7.0	✓	✗	`ÆPICSan`	I	25–1827	11	Various SBDR/DRPW issues (CVE-2023-38023)	✗
SCONE	5.8.0	✓	✗	`ABISan`	I	5	1	MXCSR dependent timing	✗
SCONE	5.8.0	✓	✗	`ÆPICSan`	I	1342–1624	3	Various SBDR issues	✗
SCONE	5.8.0	✓	✗	`PTRSan`	I	1621	3	Unconstrained read	✗
SCONE	5.8.0	✓	✗	`CFSan`	I	864	1	PIC jump before relocation	✗
Reproduced vulnerabilities in older versions (total 69 instances)									
GoTEE	014b35f	✗	✓	`ABISan`	E	3	1	Unsanitized FPU [22]	✗
Gramine	1.2	✓	✓	`ÆPICSan`	I	22–55	10	Various SBDR/DRPW issues	✗
Intel SDK	2.1.1	✓	✓	`ABISan`	E	3	1	Unsanitized DF/AC [21]; FPU [22]	✓
Intel SDK	2.13.3	✓	✓	`ÆPICSan`	I	207–6198	28	Various SBDR/DRPW issues	✗
Open Enclave	0.4.1	✓	✓	`ABISan`	E	4	1	Unsanitized DF [21]	✗
Open Enclave	0.4.1	✓	✓	`PTRSan`	I	402–1712	13	Unconstrained pointers [21]	✗
Open Enclave	0.4.1	✓	✓	`ÆPICSan`	I	442–1712	13	Various SBDR/DRPW issues	✗
Rust EDP	1.63	✓	✓	`ÆPICSan`	I	1041–1043	2	Various SBDR/DRPW issues	✗

Legend: † Not open source, but source code was made privately available; ↪ Based on above runtime.

Appendix E.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

E.1. Summary

This paper introduces Pandora, a symbolic execution tool that analyzes the security of enclave runtimes. The design of Pandora is focused on providing end-to-end analysis (including low-level runtime initialization and entry phases) and being runtime agnostic, extensible, and accessible. Pandora is shown to provide for the first time a comprehensive analysis of enclave shielding runtimes, discovering 200 new vulnerabilities across 11 widely used enclave shielding runtimes.

E.2. Scientific Contributions

- Creates a new tool to enable future science
- Identifies an impactful vulnerability
- Provides a valuable step forward in an established field

E.3. Reasons for Acceptance

1) The paper introduces Pandora, a symbolic execution tool for enclave shielding runtimes. The central goal of Pandora is on truthful validation of SGX binaries, considering critical initialization code that prior systems have overlooked. The tool is runtime agnostic and therefore applicable to many enclave applications.
2) The paper identifies several new vulnerabilities. The experiments demonstrate the Pandora discovers 200 new vulnerabilities across 11 widely used enclave shielding runtimes.
3) The paper provides a valuable step forward in the study of enclave security. The paper describes the challenges associated with a sound end-to-end analysis of enclave interfaces, and the results show that there is still a significant amount of work to be done to improve the security of enclave applications.