Designing a Provenance Analysis for SGX Enclaves

Flavio Toffalini
EPFL
Lausanne, Switzerland

Mathias Payer
EPFL
Lausanne, Switzerland

Jianying Zhou
SUTD
Singapore, Singapore

Lorenzo Cavallaro
UCL
London, United Kingdom

ABSTRACT

Intel SGX enables memory isolation and static integrity verification of code and data stored in user-space memory regions called enclaves. SGX effectively shields the execution of enclaves from the underlying untrusted OS. Attackers cannot tamper nor examine enclaves' content. However, these properties equally challenge defenders as they are precluded from any provenance analysis to infer intrusions inside SGX enclaves.

In this work, we propose SgxMonitor, a novel provenance analysis to monitor and identify anomalous executions of enclave code. To this end, we design a technique to extract contextual runtime information from an enclave and propose a novel model to represent enclaves' intrusions. Our experiments show that not only SgxMonitor incurs an overhead comparable to traditional provenance tools, but it also exhibits macro-benchmarks' overheads and slowdowns that marginally affect real use cases deployment. Our evaluation shows SgxMonitor successfully identifies enclave intrusions carried out by state of the art attacks while reporting no false positives and negatives during normal enclaves executions, thus supporting the use of SgxMonitor in realistic scenarios.

KEYWORDS

TEE, SGX, provenance analysis

ACM Reference Format:
Flavio Toffalini, Mathias Payer, Jianying Zhou, and Lorenzo Cavallaro. 2018. Designing a Provenance Analysis for SGX Enclaves. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Intel Software Guard eXtension (SGX) is an an ISA abstraction that allows developers to define *enclaves* [40, 64], small user-space regions with strong security properties. SGX provides memory isolation of enclaves from the underlying untrusted OS, and a remote attestation mechanism, so-called SGX RA, to verify their integrity. Although enclaves may host arbitrary programs, they are primarily aimed at protecting software components that carry out specific security- and privacy-sensitive tasks [19, 21, 50, 68, 73]. Both academic [82] and industry [15, 32, 47, 59, 60] proposals embrace SGX to execute such sensitive components.

In a nutshell, SGX guarantees that an enclave is properly loaded in memory, while SGX Remote Attestation (RA) allows a remote entity to verify the correct enclave initialization, similar to a preboot TPM static code measurement. Attestation of arbitrary enclave state (*e.g.,* during or after requests) is, so far, out of scope. As such, SGX alone has no mechanisms to guarantee the correct runtime execution of enclaves, which remain vulnerable against confused deputy attacks aimed at causing deviations from enclaves' expected legitimate behaviors and lead to data leakage [14, 21, 31, 48, 77].

Although one can equip enclaves with mechanisms tailored at counteracting specific threats (*e.g.,* CFI or shadow stacks), these solutions simply stop an attack without providing the analyst information about the intrusion. In real scenarios, however, solely blocking an intrusion does not prevent further attempts in similar contexts. Moreover, recent works highlighted the difficulties of removing all vulnerabilities from SGX enclaves by design [21]. In this regard, having insights about the attack vector thus becomes crucial for helping analysts and engineers to improve the defenses. This problem grows in importance when paired with memory isolated environments, such as SGX, that shield the inspection of enclaves a-priori [74]. In normal scenarios, such as standard applications or OSs, one can extract forensic evidence of an intrusion by employing provenance analyses [36, 41, 62, 89], that allow one to inspect the adversary movements from a stream of events (*e.g.,* system logs, syscall invocation). Unfortunately, the SGX memory isolation hinders provenance analysis by disallowing any mechanism to monitor enclave executions (*e.g.,* Intel PT [43] or Intel LBR [28, 90]).

Observing the lack of provenance techniques for SGX, we introduce SgxMonitor, which allows an external (and legitimate) entity to inspect an enclave runtime state and retrieve evidence of intrusion. Having a robust provenance analysis for SGX enclaves requires us to overcome two challenges: first, design a secure tracing mechanism for SGX enclaves, and second, propose a model to represent useful intrusion information. For the first challenge, we combine a lightweight enclave instrumentation with a novel communication protocol that allows the emission of contextual runtime information in the presence of a compromised OS, thus adhering to the standard SGX threat model. Our tracing is designed to offer a similar granularity as Intel PT but for SGX enclaves, forming the foundation for provenance analyses. Addressing the second challenge, we detect intrusion through a novel Finite-State Machine (FSM) that extends the current models used in SGX [23]. We then rely on a combination of

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

symbolic execution and a flow-, path-, and context-insensitive static analysis to create a FSM of the code in an enclave. Intuitively, an enclave deviating from its FSM gives insights about the attack vector.

To support our claims, we evaluate the properties of SgxMonitor in terms of security guarantees and usability. To assess the security properties of SgxMonitor, we test it against SnakeGX [31], a novel data-only malware for SGX enclaves, and specifically-crafted security benchmarks (Section 7.1.1). Moreover, we discuss if our communication protocol may introduce information leakage and outline mitigation (sections 7.1.2 and 7.1.3). Finally, we provide a security analysis of SgxMonitor (Section 7.1.4). To assess whether SgxMonitor is usable in practice, we deploy it across five use cases (Section 7.2): (i) Signal Contact Discovery Service [7] (Contact), a privacy-preserving service that finds new contacts in the Signal app [8]; (ii) libdvdcss [80], a portable DRM library used by the VLC media player [61]; (iii) StealthDB [82], a plugin for PostgreSQL [56] that relies on SGX; (iv) SGX-Biniax2 [12], a video game ported to SGX; and (v) a unit-test specifically designed to stress specific enclave behaviors not covered by the other use cases (*i.e.,* exception handling).

In summary, we make the following contributions:

- We propose SgxMonitor, a novel provenance analysis system designed for SGX enclaves that provides: (i) a new design for tracing the enclaves runtime behavior in the presence of an adversarial *host* without relying on additional hardware isolation (Section 4); (ii) a stateful representation of the SGX enclaves runtime properties (Section 5).
- We assess the security properties of SgxMonitor against SnakeGX and a specifically-crafted security benchmarks (Section 7.1.1). Moreover, we discuss possible information leakage and propose mitigation (sections 7.1.2 and 7.1.3). Finally, we illustrate a security analysis (Section 7.1.4).
- We likewise evaluate the usability of SgxMonitor, in particular: (i) the micro-benchmark shows a median overhead of 3.9x (Section 7.2.1); (ii) the provenance analysis speed of SgxMonitor is in line with state-of-the-art works (a median of 260K *actions*/s) (Section 7.2.2); (iii) the deployment of SgxMonitor does not affect the final user experience (*e.g.,* we smoothly played a DVD on VLC and a video game, and measured an average 1.6x slowdown on PostgreSQL) (Section 7.2.3); (iv) we show a 96% enclave coverage with *zero* false positive, and investigate the trade-off between symbolic execution and static insensitive analysis (Section 7.2.4).

2 SGX BACKGROUND

Enclaves stand at the base of the SGX programming pattern. They are contiguous memory regions that contain critical pieces of software and data (*e.g.,* cryptographic keys). The isolation of SGX enclaves is handled at microcode level and is independent of the Operating System (OS) which is considered malicious.

SGX specifies new opcodes to interact with *enclaves*. For our work, we consider three of them: (i) `EENTER`, to trigger the enclave execution; (ii) `EEXIT`, to leave the enclave execution; and (iii) `ERESUME`, to resume the enclave execution after an exception. Moreover, SGX uses Asynchronously Enclave Exit (`AEX`) to handle runtime exceptions.

On top of the former opcodes, Intel provides a Software Development Kit (Intel SGX SDK) that organizes the enclave code as *secure functions*. A process can interact with an enclave by means of simple primitives: ECALL, to invoke a *secure function*; ERET, to return the execution from a *secure function*; OCALL, to invoke a function outside the enclave (*i.e., outside function*); and ORET, to resume a *secure function* execution from an *outside function*. In addition, the Intel SGX SDK defines dedicated *secure functions* to handle exceptions. The security guarantees provided by SGX ensure a strong protection against direct memory manipulations. However, such protections do not hold against memory corruption vulnerabilities that lead to code-reuse attacks.

In addition to memory isolation, SGX introduces a Remote Attestation protocol (SGX RA) [81] that allows an external entity to verify the integrity of an enclave. The SGX RA relies on the isolation offered by the CPU to protect the cryptographic keys. In particular, the SGX RA guarantees two properties: (i) the host machine has correctly loaded the enclave in memory, (ii) a remote entity can check the identity of the enclave and the machine (*i.e.,* CPU) that is loading it. Therefore, the SGX RA does not capture *runtime* attacks that may deviate the enclave execution. The SGX RA provides a proof of a correctly initialized enclave but does not consider running enclaves. SgxMonitor builds on SGX RA for enclave initialization but later continuously verifies enclave integrity during execution.

3 THREAT MODEL

In this section, we describe the threat model for SgxMonitor.

Adversary Assumptions: In line with the SGX assumptions [64], we assume the adversary is a host, that can attack the enclave in two ways. (i) Exploiting classic memory-corruption errors in enclave code [21, 30, 77] that lead to hijacking the enclave execution path [14, 48]. (ii) Altering the enclave communication by overhearing, intercepting, and forging packets such as the Dolev Yao attacker [27]. Since the enclave has no direct access to peripherals, it requires the OS assistance to communicate with the outside world. Therefore, a malicious OS can intercept messages reported/received by the enclave in the attempt to induce a wrong enclave behavior.

Enclave Assumptions: We assume an enclave developed for SgxMonitor follows the specification described in sections 5 and 4. In particular, SgxMonitor requires the source code of the enclave, that will be instrumented at compilation time to trace runtime enclave information (Section 6).

Out-of-Scope Attacks: We assume the CPU is correctly implemented, thus not prone to rollback attacks [70], micro-architectural vulnerabilities [35, 44, 76, 78, 85, 88], cache timing attacks [16, 33, 55], and denial-of-service from the host. We also assume enclaves with a correct exception handler implementation [24]. Such problems are considered orthogonal to SgxMonitor.

4 SGXMONITOR: SYSTEM DESIGN

Designing a provenance analysis that fits the SGX realm requires a re-thinking of existing approaches. In the original SGX threat model, the enclave content is protected against the whole (malicious) system. In fact, we cannot observe enclave behavior externally (*e.g.,* through Intel PT [43] or Intel LBR [28, 90]). Considering this limitation, we propose a pure software design that allows an enclave to securely stream runtime fine-grain information, namely *actions*, similarly to

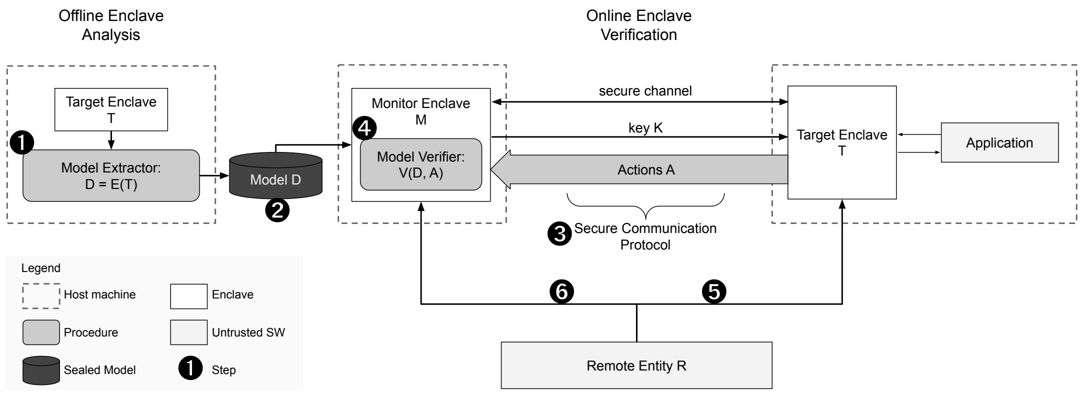


Figure 1: The SgxMonitor design is composed of two distinct phases: Offline Enclave Analysis and Online Enclave Verification. During the Offline Enclave Analysis, the *Module Extractor* analyses the *target enclave* T (❶) to obtain a *Model* D that represents the correct behavior of T (❷). During the Online Enclave Verification, an *Application* interacts with T by following standard SGX mechanisms (*e.g.,* ECALL, OCALL), meanwhile, T sends a stream of *actions* A to the *monitor enclave* M through a *secure communication protocol* (❸). M, then, uses a *Model Verifier* to validate A against D (❹). Finally, a *remote entity* R can perform provenance analysis of M. Specifically, R can verify the T static software integrity through the standard SGX protocol [6] (❺) and the runtime integrity of T by inquiring M about T runtime status (❻).

Intel PT allowing an outside monitor to track execution provenance inside the enclave without observing the computed data flow.

This section will mainly focus on the system design by providing an overview in Section 4.1, discuss the *action* emission mechanism in Section 4.2, and explaining the communication protocol in Section 4.3. The points strictly related to the model and the provenance analysis will be detailed in Section 5.

4.1 Overview

Figure 1 illustrates the SgxMonitor design, that involves seven actors:

- a *target enclave* T, the enclave to monitor against attacks under the threat model described in Section 3.
- a *monitor enclave* M, that receives the *actions* A generated by T.
- an *Application*, that interacts with T through standard SGX specifications (*e.g.,* ECALL, OCALL),
- the *Model* D, that represents the correct behavior of T.
- the *Model Extractor*, that generates a model containing the correct behavior of T.
- the *Model Verifier*, that validates the runtime status of T according to A and D.
- a *remote entity* R, that attempts to validate both software and runtime integrity of T.

Goal and Assumptions. Our system lets the monitor M securely collect *actions* A from the enclave T, and later allow a remote entity R to verify the state of T through M. We assume T, or its host, may be compromised. Moreover, we move M into a separate host to limit the effects of attacks against M. Finally, R represents a system administrator that desires to validate the integrity of T, we ensure the trustworthiness of R by employing the standard SGX RA [6].

Our goal partially overlaps with runtime remote attestation works [4, 72], in which T and M are merged into a single entity. However, such a solution does not fit our requirements because SGX enclaves cannot be internally segmented (i.e., an enclave forms a single inseparable fault domain). Therefore, in case of intrusion, we cannot ensure T is following the intended design, *e.g.,* the adversary may alter the *action* emission or leak communication keys. Conversely, uncoupling T and M (and moving the latter into a separate host) raises the bar for attacks against T.

Overall, the design of SgxMonitor is split into two distinct phases: *Offline Enclave Analysis*, and *Online Enclave Verification.* During the *Offline Enclave Analysis*, the *Model Extractor* generates the *Model* D representing the correct behavior of the *target enclave* T (❶). Then, we seal D to prevent a malicious host to tamper with it (❷). During the *Online Enclave Verification*, we assume that M and T are correctly loaded in the respective hosts. Once T is loaded, it establishes a *secure communication channel* with M by using the standard SGX RA [6], as described in Section 4.3 (❸). This channel allows T to send a stream of *actions* A to M, while an *Application* can interact with T by following standard SGX mechanisms (*e.g.,* ECALL, OCALL). Finally, M uses the *Model Verifier* to validate the runtime integrity of T by controlling A against D (❹). The *Model Extractor* (❶) and *Verifier* (❹), along with further model details, are described in sections 5.5 and 5.6, respectively.

Once M correctly receives A from T, a *remote entity* R can attest the software integrity of T through the standard SGX RA [6] (see Section 2). This ensures that the software in T has been loaded properly and is not tampered with (❺). Since we employ the standard SGX RA, we do not provide further details. Finally, R can inquiry M regarding the runtime state of T, *i.e.,* if T still follows the model D and, in case, where the model diverges and how (❻).

4.2 Action Reporting Mechanism

T relies on an *action* reporting mechanism that is resilient against the threat model described in Section 3: an intrusion inside T (*e.g.,* exploiting a T internal error), and a malicious host.

```
1 int fun(int a) {
2   /* function body */
3
4   // trace the indirect jump to the caller
5   trace(__builtin_return_address(0));
6   return 0;
7 }
8
```

Figure 2: Example of code instrumentation. We report the action before critical program edges are traversed. This disallow an adversary to hijack T without reporting an *action*. The secure protocol then ensure the adversary cannot forge an *action* (Section 4.3).

We design the *action* reporting as a dedicated function, called `trace()`, that is included in crucial code locations of T at compilation time. Without loss of generality, we say all the *actions* are reported through `trace()` over a secure channel between M and T (Section 4.3). This section mainly focuses on the reporting mechanism, while a complete description of *actions* is presented in Section 5.2. Finally, we assume `trace()` is free from errors and an adversary cannot exploit it to take control of T. This is reasonable since `trace()` has a minimal implementation tailored for *action* reporting.

The intuition of our mechanism is to report an *action before* a critical control-flow location is traversed (*i.e.,* a return instruction). We exemplify this mechanism in Figure 2, in which the program traces an *action* representing a return edge to the caller (line 5). In this scenario, an adversary could attempt an intrusion by injecting a ROP chain, report arbitrary actions, and finally hiding her presence in T. In this case, T will report an *action* representing the anomalous return address (*i.e.,* the first ROP gadget) right before the payload is executed, thereby producing evidence of the intrusion. We can generalize this approach such that T reports every *action before* they are actually executed, *i.e.,* before an intrusion begins. We paired this mechanism with the secure communication protocol (Section 4.3) that avoids forging and tampering with already reported *actions*. Therefore, an adversary cannot hijack T without reporting evidence about the attack.

Our solution is robust against attempts of overwriting `trace()`. In this case, we use the standard SGX security properties and distinguish two cases. First, in SGX 1.0 [1], the host cannot arbitrary alter the page permission of an enclave, this blocks any overwrite attempts by design. Second, for SGX 2.0, a host can change the enclave memory layout (*i.e.,* change page permission) only upon an enclave request. However, for this to happen an adversary has to first complete an intrusion in T, thus reporting evidence of the attack similarly to the previous scenario.

We thus claim the *action* emission, when paired with the secure communication protocol (Section 4.3), provides the base for our resilient provenance analysis. We further investigate adversarial scenarios through a dedicated security analysis im Section 7.1.4.

4.3 Secure Communication Protocol

T and M exchange messages relying on a secure communication channel resilient against an adversarial host that may alter, eavesdrop, or forge the packets.

Algorithm 1: Procedure used by the *target* enclave to report logs in a secure fashion.

1 reportLog(A)
2 $\quad \text{mac} \leftarrow H_1(A|K)$
3 $\quad C \leftarrow (A|\text{mac}) \oplus K$
4 $\quad K \leftarrow H_2(K)$
5 $\quad write(C)$

Protocol properties. Our protocol ensures two properties: (i) the host cannot tamper with the packets reported by T; (ii) an adversary cannot alter or forge the packets already reported even if she takes control of T. Note that we accept an adversary that performs a denial-of-service between T and M. In this case, M considers T as untrusted after a timeout.

Workflow. The channel requires three steps to be established (❸ in Figure 1): (i) T issues a standard SGX RA [6] with M, thus ensuring a respective identity verification; (ii) M sends a secure *key* K to T; and (iii) T sends the *actions* to M. The secure channel is shared among the threads of T, that refer to the same key K. We also include a thread ID into the exchanged packets, this allows M and T to multiplex and demultiplex the communication. The adoption of a shared key K avoids an adversary to use the technique discussed in Dark-ROP [48], we provide more details in the Section 7.1.4.

The validation of the transmitted *actions* relies on two algorithms, `reportLog()` and `verifyLog()`, that are illustrated in the algorithms 1 and 2, respectively. Both `reportLog()` and `verifyLog()` use a lock to avoid concurrency problems. K has the same size of the packets transmitted, thus avoiding crypto-analysis [37]. Finally, we assume `reportLog()`, `verifyLog()`, and the other supporting functions do not contain implementation errors. We consider this reasonable since these functions are specialized for this task.

T reports a new action A through instrumented code (described in Section 4.2). A is given as an input to `reportLog()` that encrypts and transfers it to M over an insecure channel. First, `reportLog()` creates a *mac* by using an hash function H_1 and the concatenation of A and the key K (Line 2 Alg. 1). Then, it generates C by *xor*-ing the concatenation of *action* A and *mac* with the key K (Line 3 Alg. 1). At this point, it generates a new key K by hashing the current key K with the function H_2 (Line 4 Alg. 1). Finally, the function writes C into an insecure channel (Line 5 Alg. 1).

On the other side, M relies on `verifyLog()` to decrypt and validate the encrypted packets C. We also assume that M receives the packets in order.[1] First, M decrypts the pair $(A|\text{mac})$ by *xor*-ing the packet C and the key K (Line 2 Alg. 2). Then, M verifies the correctness of the packet received by independently computing *mac′* (Line 3 Alg. 2). If *mac* and *mac′* does not agree, C was tampered during the transmission and M sets T as untrusted (Line 5 Alg. 2). Otherwise, A is considered correct and is processed as described in Section 5.6 (Line 7 Alg. 2). Finally, M generates the next key K similarly to T (Line 9 Alg. 2).

In Section 7.1.4, we illustrate a security analysis of the security protocol and the *action* emission mechanism (Section 4.2).

Algorithm 2: Algorithm used by the *monitor* enclave to verify the logs reported through *reportLog()* described in Algorithm 1.

verifyLog(C)
  $(A|\text{mac}) \leftarrow C \oplus K$
  $\text{mac}' \leftarrow H_1(A|K)$
  if $mac' \neq mac$ **then**
    untrusted()
  else
    process(A)
  end
  $K \leftarrow H_2(K)$

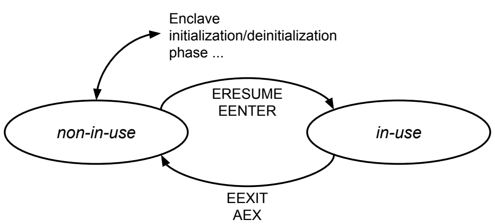


Figure 3: Standard Finite-State Machine representation of SGX Enclaves [23].

5 SGXMONITOR: THE ENCLAVE MODEL

To model the enclave behavior, we design a novel Finite-State Machine that extends the standard SGX enclave life-cycle depicted in Figure 3.[2] The standard SGX model assumes the enclave has been loaded correctly and the host interacts with it by means of the opcodes described in Section 2. The model allows the enclave state to assume only two values: *non-in-use* and *in-use*. In particular, an enclave transits to *in-use* state when an `EENTER` or `ERESUME` is issued. Then, the state returns to *non-in-use* when an `EEXIT` or `AEX` happens. This simple model is already implemented in the microcode: the same thread cannot enter (*i.e.,* `EENTER`) in an enclave which is already in *in-use* state; it cannot exit (*i.e.,* `EEXIT`) when the enclave is in *non-in-use*.

Intuitively, the model in Figure 3 provides limited information about enclave health. In case of new attacks against enclaves' code [14, 31, 48], we cannot trace the enclave execution thus precluding provenance analysis in case a-priori.

Analyzing intrusion techniques for SGX enclaves, we noticed two patterns. Attacks either hijack the enclave execution flow [48, 77], or corrupt internal enclave structures [14, 31]. Therefore, we design the SgxMonitor model to recognize those patterns. Specifically, our model is composed by four elements:

- *states*, that represent the runtime values of global structures (Section 5.1).
- *actions*, that are meaningful binary level events (*e.g.,* `EENTER`, function call) (Section 5.2).
- graphs of *actions*, that are computed offline and used to validate runtime transactions (Section 5.3).
- *transactions*, that are sequences of *actions* leading an enclave from a state to the next. They express correct execution paths (Section 5.4).

In the rest of the section, we detail state, *actions*, transactions, and graphs of *actions*. Then, we describe the *Model Extractor* and *Verifier* in sections 5.5 and 5.6, respectively

5.1 State Definition

Our model employs a state that represents important global structures used by the Intel SGX SDK. These structures handle operations such as *outside function* invocation and *exception handling* (Section 2) and are targeted by the adversaries [14, 48]. Having an enclave that reaches an anomalous state provides information about the tactic adopted for the intrusion.

Due to the multi-threading nature of enclaves, SgxMonitor traces a state for each thread [1]. The state is a triplet defined as ($usage$, $structure$,$operation$). In particular, *usage* recalls the FSM meaning seen in Figure 3 and can assume two values: *in-use* and *non-in-use*. *Structure*, instead, is an hash representation of the current structure used. If no *structure* is used, it assumes *null* value (*i.e.,* $\oslash$). Finally, *operation* represents the last operation performed over the *structure*. In our model, the structures do not change over time, thus, we trace their generation (*i.e.,* G) and consumption (*i.e.,* C). In case no operation has been performed, we consider a *null* action (*i.e.,* $\oslash$).

In our proof of concept, we trace the generation and consumption of (i) `ocall_context`, used in the *outside functions* invocation; and (ii) `sgx_exception_info_t`, used in the *exception handing*. These two structures are handled at thread granularity, thus they fit our model. In Appendix A, we show their FSM representation.

5.2 Action Definition

Generally speaking, an *action* is a meaningful software event. We use the *actions* to represent runtime enclave transactions (Section 5.4), that allow the evolution of the enclave state; and to build graph of *actions* (Section 5.3), that we use to validate the runtime transactions. In particular, we distinguish two type of *actions*: *generic* and *stop*.

Generic actions. They identify standard software behaviors such as: (i) edges generated by *control-flow* events; *e.g.,* `jmp`, `call`, `ret`; (ii) conditional branches (*e.g.,* `jc`); and (iii) function pointer and virtual table assignment. Generic *actions* do not alter the state of the enclave and they are used to identify correct executions. We choose these events because they are key information to represent execution paths [28, 39, 43, 72, 90].

Stop actions. They alter the state of the enclave, in particular, we consider particular SGX opcodes and structures manipulation. For what concerns SGX opcodes, we consider `EENTER`, `EEXIT`, and `ERESUME`, moreover, we distinguish between `EEXIT` used for an ERET or an OCALL, respectively. These actions alter the first field of the state (*i.e., usage*): when an application enters an enclave, *usage* becomes *in-use*, while *usage* turns to *non-in-use* when the enclave exits. For structures manipulations, instead, we trace whenever the enclave generates or consumes a structure. This actions alter the *structure* and the *operation* fields in the state; *i.e.,* when an *action* generates

[1] We assume a reliable channel like TCP as in [72].
[2] This model is a simplified version of [23].

Table 1: *Actions* used to define valid transactions grouped by *generic* and *stop*, respectively.

Actions	
Generic	
(E, `src`|⊘, `dst`|⊘)	Function call, ind. jump, or ret inst. `src` and `dst` can assume null value (*i.e.,* ⊘)
(B, `src`, 0|1)	Conditional branch (0: not taken, 1: taken)
(A, `src`, `addr`)	Function pointer assignment
(V, `src`, `vptr`)	Virtual pointer assignment (for C++ virtual classes)
Stop	
(G, `src`, `ctx`)	`ocall_context` generation
(C, `src`, `ctx`)	`ocall_context` consumption
(J, `src`, `ctx`)	`sgx_exception_info_t` generation
(K, `src`, `ctx`)	`sgx_exception_info_t` consumption
(N, `src`, `idx`)	`EENTER` for the *secure function* `idx`
(R, `src`, ⊘)	`ERESUME`
(T, `src`, ⊘)	`EEXIT` from `enter_enclave` (ERET)
(D, `src`, ⊘)	`EEXIT` from `do_ocall` (OCALL)

a structure, we store the new structure hash and set *operation* as G, while we set *structure* to null (*i.e.,* ⊘) and *operation* to C when the structure gets consumed.

Both *generic* and *stop actions* are formalized as a triplet:

$$a = (type, src, value)_{cond}.$$

In particular, *type* identifies the nature of the *action* (*e.g.,* function call, `EENTER`). `Src`, instead, is the virtual address at which the *action* has been performed. *Value* depends by the actual *action* semantic; for instance; it contains the *callee* address in case of function call; a boolean value (*i.e.,* taken or not) in case of conditional branches; a *null value* (*i.e.,* ⊘) in case the *action* does not require it. Finally, *cond* contains extra condition (*e.g.,* $value \geq 0$). We provide the complete *action* list in Table 1 grouped by *generic* and *stop*.

5.3 Graphs of Actions Definition

Graphs of *actions* are composed of vertexes and edges. More precisely, vertexes and *actions* are in a bijective relationship, *i.e.,* each vertex is paired with exactly one *action* and each *action* is paired with exactly one vertex. The edges, instead, are combinations of *actions* that appear at runtime.

We opted for graphs to efficiently represent loops, that otherwise require an unpredictable sequence of *actions*. Moreover, the graphs of *actions* allow us to implement a shadow stack. We describe the model extraction and verification in sections 5.5 and 5.6, respectively.

Algorithm 3: Extracting model algorithm, it takes as input the target enclave and returns the relative model.

```
extractModel(T)
    m ← ∅
    for f ∈ T.instr_functions do
        setSymbolicGlobalVars(T)
        loopAnalysis(f)
        setSymbolicFreeArgs(f)
        r ← symbolicExploration(f)
        if r.isTimeout() then
            r ← insensitiveAnalysis(f)
        end
        m ← m ∪ (f, r.graph_of_action)
    end
    return m
```

5.4 Transaction Definition

A transaction identifies a valid execution path in an enclave and is composed of a valid sequence of *actions* (Section 5.2) that makes the enclave state evolve. Formally, we indicate a transaction P as following $P = [g_1, ..., g_n, s]$, which is a sequence of *generic actions* g_i that terminates with a *stop action* s. Intuitively, an enclave should reach a new state only through valid transactions, otherwise we observe an anomalous enclave behavior. We perform the transaction validation by matching the *actions* received from the monitored enclave with its graphs of *actions*. We provide the full validation algorithm in Section 5.6. The combination of transactions and graph of *actions* allows one to recognize intrusion tactics [48, 77].

5.5 Model Extractor

The goal of the *Model Extractor* (❶ in Figure 1) is to automatically infer the behavior for a given enclave. A naive approach would use a symbolic execution [42] over the entire enclave. However, this strategy does not scale to the whole code base. Another approach would use insensitive static analysis [22] to extract the control-flow graphs of each function. However, this approach introduces impossible paths that increases the attacker surface. In our scenario, we assume that the code in an enclave implements straight-forward functionality, such as a software daemon that implements different features [3] and not arbitrarily complex like, e.g., a web-browser. An enclave contains a relative small number of indirect call and its software base is given. Therefore, we take inspiration from previous compositional analysis [17] that treats individual functions separately. More precisely, we extract a model for each function of the enclave with a combination of symbolic executions and insensitive static analysis. We detail the model extracted in Section 5 and we describe the metrics used to define *simple* programs in Section 7.2.4.

The *Model Extractor* takes as input a *target enclave* T which has been instrumented at compilation time (Section 6); *i.e.,* it contains extra code that traces runtime enclave information, namely *actions*; and outputs a graph of *action* for each traced function in the enclave. T is compiled without debug information, we solely rely on global symbols to identify the functions entry point and the global variables.

The global symbols do not contribute to the enclave measurement, thus we strip them out after extracting the model [40] (Section 2).

Overall, the extraction algorithm is described in Algorithm 3. Given an instrumented *target enclave* T, we analyze each instrumented function separately (Alg. 3 line 3). We describe each point of the analysis in the rest of the section, while we formalize the model in Section 5.

Symbolic Global Variables (Alg. 3 line 4): Global variables might contain default concrete values that affect the symbolic exploration. We mitigate this issues by setting all the global variables as unconstrained symbolic objects. We repeat this operation for each function to clean the symbolic constraints previously set.

Loop Analysis (Alg. 3 line 5): Unbounded loops can lead to infinite symbolic explorations [58]. Since we are interested to reduce false positive alarms, we employed a postdominator tree [63] over the static control-flow-graph to identify the loops header in each function. This approach is conservative and allows us to explore more execution paths, which is our main goal. We set the maximum to three loop iterations, similarly to previous works [84]. Our experiments show that we reach a good coverage while keeping low false positive.

Free Arguments Inferring (Alg. 3 line 6): Some function requires pointers as arguments (*e.g.,* structures, objects, array), however, current symbolic explorations do not fully handle symbolic pointers, that might lead to a wrong or incomplete exploration [22]. Since we are interested to reduce false positive alarms, we opted for a conservative approach based on static backward slicing [87] to identify pointers passed as function arguments. For each free pointer, we build an unconstrained symbolic object to help the exploration. This solution allows us to achieve a good coverage in the majority of the case, as also shown in our experiments. We also introduce custom analysis to handle corner cases, which are though a limited number. Finally, we deal with functions pointers by employing a conservative function type analysis [3].

Symbolic Exploration (Alg. 3 line 7): We primary employ a symbolic exploration [42] to avoid impossible paths that, otherwise, might increase the attacker surface. We execute the symbolic exploration after tuning the function as previously described. Through the exploration, we build a graph of *action* for each function.

Insensitive Static Analysis (Alg. 3 line 9): Since few functions of our use case experienced a symbolic execution timeout due to their complexity (*i.e.,* too many nested loops). We employed a fallback approach based on an insensitive static analysis [65] in which we traverse the static control-flow-graph of the function to build the function graph of *action*. These cases are rare and they are used only if the symbolic approach fails. We measure the frequency of this case in our evaluation.

Building a Model (Alg. 3 line 11): The final enclave model is an association between functions and their model. We refer to Section 5 for further details. Finally, we seal the output in the *monitor enclave* host to avoid tampering.

5.6 Model Verifier

The *Model Verifier* (❹ Figure 1) receives a stream of *actions* from the *target enclave* T and checks whether they adhere to the *Model* D. Every *action* moves T from a state to the next one, the forward jumps are validated directly against the *Model* D, while the back jumps (*e.g.,* `ret` instructions) are validated against a shadow stack [72]. These mechanisms ensure the sequence of *actions* follow a correct path. Moreover, the *Model Verifier* tracks the running state of T and identifies when the enclave reaches a wrong state. Failing to adhering to the model D gives insights about the intrusion tactic used to control the enclave.

6 IMPLEMENTATION

We provide technical details about the *Compilation Unit*, the *Model Extractor*, and the *secure communication channel*.

Compilation Unit: The *Compilation Unit* takes as input the *target enclave* source code and emits the instrumented enclave T. The instrumentation injected at compilation time is considered trusted since SGX disallows an OS to arbitrary change the enclave's page permission, thus avoiding code replacement [40]. The unit is implemented as an LLVM pass for the version 9 (367 LoC) and a modified version of Clang 10 that instruments virtual pointer assignments (15 LoC added). In the link phase, we link T with an instrumented SGX SDK to trace specific parts of the code, *e.g.,* in `do_ocall` and `asm_oret` to handle `ocall_context` generation/consumption; and `enter_enclave` to trace the entrance/exit from the enclave. We opted for this solution because Intel does not officially support the compilation of the SGX SDK with Clang [2]. We based the instrumented SGX SDK on the version 2.6. In this process, we also include an extra secure function that issues the *secure communication channel*, and extra checks that avoid the interaction between T and the *Application* before the channel is established (see Section 4.3).

Model Extractor: The *Model Extractor* is based on angr version 8.18 and implements the algorithms described in Section 5.5. We use PyVex [69] to navigate the static CFG of the functions, and angr symbolic engine to extract the graphs of *actions*. The *Model Extractor* is composed of 8416 LoC in total.

Secure Communication Channel: The communication between the *target enclave* T and the *monitor enclave* M is implemented by combining a TCP connection and a switchless mechanism [71]. T writes encrypted actions (see Section 4.3) into a ring-buffer that resides in the untrusted host. The buffer is then flushed into a TCP socket that connects T and M. On the M side, another ring-buffer feeds the *Module Verifier*. We employ this design to reduce context switch delays [71]. For the functions `reportLog()` and `verifyLog()`, we use the *sha256* implementation provided by Intel SGX SDK. We can improve the efficiency adopting other secure functions such as the Intel SHA extension [34] or Blake2 [9].

7 EVALUATION

We design our evaluation following the guidelines described in [79] to avoid benchmarking flaws. Our evaluation revolves around two main questions: (**RQ1**) *what* insights SgxMonitor provides in a provenance analysis? (**RQ2**) can I use SgxMonitor in a *real scenario*? We answer **R1** in Section 7.1 by testing the SgxMonitor security guarantees against a set of modern SGX attacks. We answer **RQ2** in Section 7.2. More precisely, we measure micro-benchmark (Section 7.2.1), provenance analysis speed (Section 7.2.2), macro-benchmark (Section 7.2.3), and discuss the model extraction (Section 7.2.4).

7.1 RQ1 - Security Evaluation

We evaluate the security guarantees of SgxMonitor from multiple perspectives. First, we demonstrate the provenance capability of SgxMonitor to intercept modern execution-flow attacks (Section 7.1.1). Then, we discuss non-control data attacks and discuss mitigation (Section 7.1.2) and analyze the impact of SgxMonitor in side-channels scenarios (Section 7.1.3). Finally, we provide a security analysis of the SgxMonitor design (Section 7.1.4).

7.1.1 Execution-flow attacks. Since SGX does not allow one to arbitrary change the page permission of a running enclave, researchers adapted memory-corruption errors to hijack the enclave execution. To test the properties of SgxMonitor against this class of attacks, we choose two security benchmarks: SnakeGX [31], which is an enclave infector for SGX enclaves; and a security benchmark that evaluates the correctness of the shadow stack defense.

SnakeGX. This is a data-only malware designed to implant a permanent backdoor into legitimate SGX enclaves. SnakeGX is an extension of the work of Biondo et. al [14] and is based on code-reuse techniques. SnakeGX is composed of two phases: (i) an *installation phase*, that uses a classic ROP-chain [18] to install the payload inside the *target enclave*; and (ii) a *backdoor activation*, that exploits a design error of the Intel SGX SDK to trigger the payload previously installed. SnakeGX managed to bypass the current SGX protections. Therefore, once installed, an external observer cannot realize the presence of SnakeGX in the *target enclave*. For our evaluation, we recompiled the victim enclave including SgxMonitor, and we adjusted the *gadgets* addresses of SnakeGX accordingly. Then, we extracted the model, execute the malware, and finally, traced the *actions* reported. The results show that SgxMonitor recognized either the *installation phase* and the *backdoor activation*. In particular, the *installation* relies on a classic ROP-chain, therefore, SgxMonitor identified an unknown *action* pointing a *gadget*. In this way, SgxMonitor gave an information about an intrusion inside the enclave. The *backdoor activation*, instead, restores a corrupted `ocall_context` (crafted during the installation). In this case, SgxMonitor observed the restoring of an anomalous state. Notably, previous works cannot identify the error design used in this phase [21], unlike SgxMonitor.

To sum up, SgxMonitor gave insights about an intrusion into the *target enclave* by revealing the gadget used for the *installation phase* and the deviated patch that lead to the tampered structure in the *backdoor activation*.

Shadow stack protection. We evaluate the shadow stack implemented in SgxMonitor. In particular, we want to identify an adversary able to overwrite the *return address* of a function with a valid location that is, however, incoherent with the call stack. To this end, we built a custom enclave that allows such attacks, we compiled it with SgxMonitor, extracted the model, and finally, run the attack. The results show that SgxMonitor managed to identify execution flows incoherent with the call stack, thus pinpointing a possible local buffer overflow and in which function it happened.

Final Notes. We remark that standard mitigation deployed inside an enclave (*e.g.,* CFI or shadow stacks) lack any insight about the attack performed. On the contrary, SgxMonitor provides fine-grain information about the intrusion.

7.1.2 Non-control data attacks. We discuss if the communication protocol between *monitor* and *target enclave* may brace the adversary capabilities in non-control data attacks [20, 38]. Before we analyze this problem, we remark that all the packets have the same size by design, and the cryptographic key changes at any packet reported (see Section 4.3). Therefore, an adversary can only analyze the packets timestamp.

These attacks do not hijack the execution-flow, for instance, an enclave may contain a password checking algorithm that matches one character at time. In this example, the number of packets suggests the number of characters guessed, thus reducing the combination. We can mitigate this attack with the introduction of dummy packets (from 0 to k) and adding a random dummy delay (from 0 to t). This will increase the micro-benchmark overhead of a factor $(k+t)$x in the worst case. However, such defenses would be applied to specific code portions (*e.g.,* in the password checking), thus incurring a minimal overhead footprint overall. (The idea is similar to adding countermeasures against timing-based attacks [13].)

7.1.3 Side-channels attacks. We study the implication of SgxMonitor in side-channel attacks. First, we focus on crypto analysis. In this case, an adversary may use the number of packets reported to attack the cryptographic algorithms in the enclave. However, modern cryptographic algorithms have been proven chosen-ciphertext attack secure [11]. Therefore, leakage of ciphertext packets does not improve the adversary's capabilities [86]. An adversary may however count the packets exchanged by the communication protocol to analyze the enclave execution and locate likely code positions. We dissect this scenario in two cases. (i) The code location could be used in *execution-flow attacks*, therefore, an adversary will trigger an anomalous execution that will be detected by SgxMonitor, as we discuss in Section 7.1.1. (ii) The code location could be used in *non-control data attacks*, that we discuss in Section 7.1.2.

7.1.4 Security Analysis of the System Design. We discuss the security properties of the SgxMonitor design (Section 4) with respect to our threat model (Section 3).

Attacks before protocol establishing. An adversary may target T before it establishes the secure channel with M. To mitigate this attack surface, we enforce that all the security functions of T are disabled until T and M completely initialize the security protocol. In particular, the *Application* must invoke a dedicated secure function of T before it may use any other secure function. We insert additional checks that ensure no other functionality of T is active until T and M successfully established the channel. This design avoids an adversary to attack T before M starts monitoring it.

Defense against a tampered enclave T. Our protocol resists an adversary that exploits T. In this case, the adversary may abuse a memory corruption error to divert the enclave execution path. However, we instrument the code of T such that it reports the *action* before the enclave traverses the hijacked edge (Section 4.2). Therefore, the *action* results are already encrypted and shipped, while K has been altered by the hash function (Section 4.3). We face three scenarios here: (S1) the compromised *action* reaches M, thus M recognizes the attack; (S2) the host drops the *action* before reaching M, thus M recognizes the attack after a timeout; and (S3) the adversary attempts to forge a new valid *action*, however, she cannot retrieve

K after `reportLog()` invocation (*i.e.,* a new K is produced). In all these cases, M will observe an anomaly in the protocol or T behavior, finally setting T as untrusted.

Sharing the same key K among the threads defeats the tactic described in modern enclave attacks [48]. In their scenario, an *adversary* exploits a thread to leak information (*i.e.,* the key K) from another thread. In our design, leaking K forces a thread to report an *action* X representing the attack. Moreover, `reportLog()` ensures the *actions* follows a specific order. Therefore, either X reaches M, thus revealing the attack; or X is dropped, thus showing an anomaly.

Security Evaluation—Take Away. Our evaluation shows that SgxMonitor provides useful information about the payload used in modern state-of-the-art attacks (*i.e.,* SnakeGX and shadow stack protection). Moreover, we discuss the possible information leakage and we show that, in practice, it does not improve the adversary capabilities. We also propose information leaking mitigation and discuss the scenarios for which they are more suitable. Finally, we discuss a security analysis of our system design.

7.2 RQ2 - Usage Evaluation

We describe the use cases used, the experiment setup, and discuss the impact of SgxMonitor in real projects.

Use Cases. We identified 10 open-source projects that use SGX. Most of them do not compile because they refer to old SGX features or they are incompatible with Clang. Among them, we choose five ones: (i) Contact [7], the contact discovery service used by Signal app [8]; (ii) an SGX porting of libdvdcss [80], a portable DRM algorithm used by VLC media player [61]; (iii) StealthDB [82], a PostgreSQL [56] plugin that uses SGX to encrypt tables; (iv) SGX-Biniax2 [12], an SGX porting of the open-source game Biniax2 [75]; and (v) a unit-test to validate corner cases of the enclave behaviors not covered previously, like exception handling. In Table 2, we indicate the line of code (LoC) and the number of secure functions for each use case.

We use Contact, StealthDB, SGX-Biniax2, and the unit-test to stress micro-benchmarks (Section 7.2.1) and provenance analysis speed (Section 7.2.2). We use libdvdcss, StealthDB, and SGX-Biniax2 for macro-benchmarks (Section 7.2.3). All the five use cases are used for model extraction analysis (Section 7.2.4).

Experiment Setup. All the experiments were performed on a Linux machine with kernel version 4.15.0 and equipped with an Intel i7 processor and 16GB of memory. We set the CPU power governor as *power save*. Moreover, we perform a warm-up round for each *secure function* before actually recording the performances.

7.2.1 Micro-benchmark. In this experiment, we measure the overhead of the single secure functions with SgxMonitor and without (*i.e.,* vanilla). We perform this experiment on Contact, SGX-Biniax2, StealthDB and the unit-test enclave. The results are shown in Figure 4a. In most of the cases, SgxMonitor introduces an overhead less than or equal to 10x (bx1-7, ct1-2, ct4, ct6, ut1-3) with a median overhead of 3.9x. Only two secure functions show an overhead over 100x (ct3 and ct5).

Micro-benchmark—Take Away. A major source of overhead is incurred by the hash functions in the secure communication protocol (Section 4.3), as observed in similar works [4, 5, 72]. Different hash functions can ease the overhead, *e.g.,* the Intel SHA extension [34] or Blake2 [9]. However, This result does not really affect the performance of SgxMonitor that is in line with similar works [72] for the of analysis speed (Section 7.2.2) and final user experience (Section 7.2.3).

7.2.2 Provenance Analysis Speed. Figure 4b measures the provenance analysis speed in terms of number of *actions* reported and validated per second (on the y-axes) for each *secure functions* of Contact, SGX-Biniax2, StealthDB, and the unit-test enclave (on the x-axes). The execution time encompasses the context-switch delay, *actions* emission, transmission, and verification at the *monitor* side.

All the secure functions, but ct1, ct5 and bx7, express a throughput that ranges from 167K *action*/sec (bx2) to 496K *action*/sec (ct6), with a median value of 260K *action*/sec.

Provenance Analysis Speed—Take Away. These figures are in line with the previous works [72]. ct1, instead, reports a fewer number of *actions* and biases the analysis speed. Finally, bx7 and ct5 perform sealing operations [6] and thus introduce an extra delay per *action.*

7.2.3 Macro-benchmark. We investigate the impact of SgxMonitor in three real applications. (A1) StealthDB [82], which is a plugin for PostgreSQL [56] based on SGX. (A2) libdvdcss [80], which is a DRM library used in VLC media player [61]. (A3) SGX-Biniax2 [12], which is an SGX porting of the open-source game Biniax2 [75].

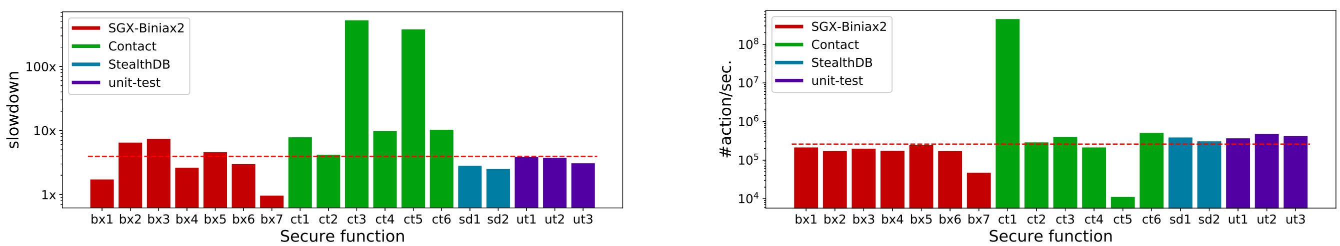
StealthDB. We replicated the same experiments described in the original paper [82]. In particular, we deployed StealthDB over a PostgreSQL [56] version 10.15 and we run the database benchmarking tool OLTP [25] by using the five scale factors indicated in the original work. Then, we reported the requests per second and the latency in figure 5a and 5b, respectively. For each scale factor, we run 10 experiments and indicate average and standard deviation. Overall, SgxMonitor introduces an average slowdown of 1.68x and an overhead of 1.25% in terms of requests per second and latency, respectively.

libdvdcss. We measured the CPU impact of SgxMonitor over libdvdcss, which is an DRM library used in VLC media player [61]. For the experiment, we used a VLC version 3.0.8, on which we deployed three versions of libdvdcss [80]: vanilla, with SGX, and with SgxMonitor. During the experiment, we played a DVD for around one hour and half while sampling the CPU usage every second. Figure 6a shows the result of our experiment, after a first adjusting phase, the overhead reaches a plateau below 10%. Furthermore, we did not experience any delay or interruption while playing the DVD in any of the three configurations.

SGX-Biniax2. . We measured the CPU impact of SgxMonitor over SGX-Biniax2 [12], an example of video game porting that uses SGX for data protection. In particular, we played the game for around 20 minutes and we sampled the CPU usage every second. Figure 6b shows the result of our experiment, similarly to libdvdcss, we observed a first adjusting phase followed by a plateau at around 5%.

Table 2: Detailed information for the of five use cases used in our evaluation: Contact [7], libdvdcss [80], StealthDB [82], SGX-Biniax2 [12], and a unit-test.

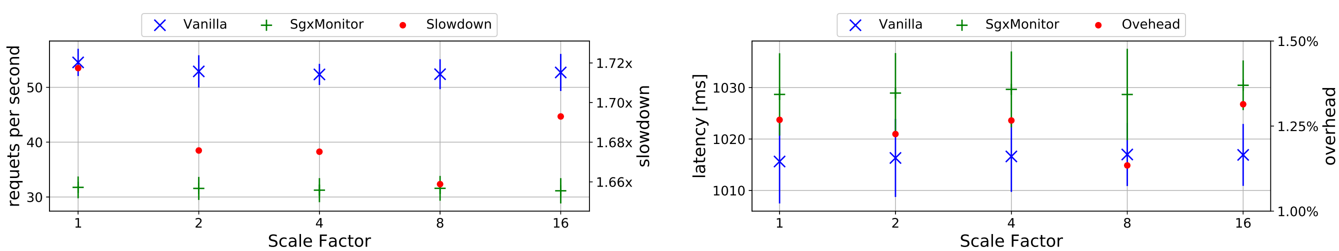
Use case	LoC	# secure function	*cycl. cmplx.* μ	*cycl. cmplx.* σ	# nodes in CFG μ	# nodes in CFG σ	# edges in CFG μ	# edges in CFG σ	# direct calls	# indirect calls
Contact [7]	4138	6	5.03	5.04	24.89	22.74	26.67	27.82	1085	16
libdvdcss [80]	3438	4	6.55	6.07	38.71	31.28	39.67	37.95	1084	2
StealthDB [82]	10351	3	6.35	4.72	36.14	23.38	40.40	27.51	1203	2
SGX-Biniax2 [12]	4696	7	3.73	4.20	18.56	16.25	20.19	20.02	583	2
unit-test	583	3	4.06	5.25	18.44	17.53	18.75	21.95	137	2



(a) Overhead of vanilla secure functions versus SgxMonitor secure functions of Contact (ct$x$), SGX-Biniax2 (bx$x$), StealthDB (sd$x$) and unit-test enclave (ut$x$) expressed in logarithmic scale. Median overhead is around 3.9x and is depicted as a dashed line.

(b) Number of *actions* processed per second of Contact (ct$x$), SGX-Biniax2 (bx$x$), StealthDB (sd$x$) and unit-test enclave (ut$x$). Median value is 260K *action* per second and is depicted as a dashed line.

Figure 4: SgxMonitor micro-benchmark and *action* speed measurement evaluation.



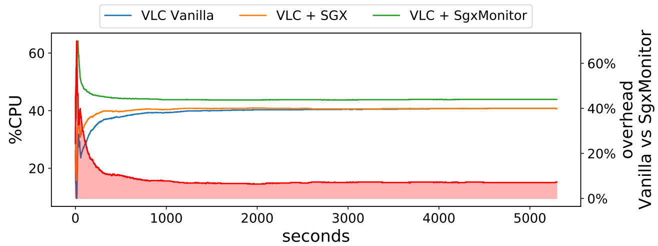
(a) Overhead of StealthDB vanilla and with SgxMonitor measured as requests per second. Overall, SgxMonitor introduces an average slowdown of 1.68x with a standard deviation of 0.02x.

(b) Overhead of StealthDB vanilla and with SgxMonitor measured as latency (ms). Overall, SgxMonitor introduces an average overhead of 1.24% with a standard deviation of 0.06%.

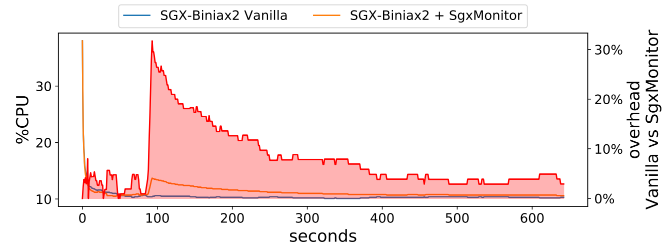
Figure 5: StealthDB [82] performances measured against OLTP [25] benchmark and expressed as request per second and latency. We evaluated StealthDB vanilla and with SgxMonitor, in particular, we run 10 measurements for each scale factor (from 1 to 16) and plot average and standard deviation for requests per second and latency, respectively.

Furthermore, we did not experience any delay or interruption while playing SGX-Biniax2 in any of the two configurations.

Macro-benchmark—Take Away. Our results show that the overhead introduced by SgxMonitor is overall limited, *e.g.,* the slowdown in StealthDB is lower than the micro-benchmarks (*i.e.,* 1.6x vs 3.9x) and the CPU overhead expressed by libdvdcss and SGX-Biniax2 shows a limited plateau. Therefore, we conclude that SgxMonitor does not affect the final user experience and can be included into projects that either require occasional enclave interactions (like DRM protection) or are more computational intense (like a database).



(a) **Overhead of VLC with libdvdcss vanilla, plus SGX, and plus SgxMonitor, respectively. We measure the percentage of CPU usage while playing the same DVD with the three settings. After an initial adjusting phase, the overhead drops and reaches a plateau lower then 10%.**



(b) **Overhead of SGX-Biniax2 vanilla and with SgxMonitor, respectively. We measure the percentage of CPU usage while playing the game for the same amount of time (around 20m). After an initial adjusting phase, the overhead drops and reaches a plateau at around 5%.**

Figure 6: Macro-benchmark of libdvdcss [80], deployed over VLC media player [61], and SGX-Biniax2 [12]. In both cases, we measured the CPU usage and the overhead introduced by SgxMonitor versus the vanilla version of the software.

Table 3: Coverage analysis over our five use cases: Contact [7], libdvdcss [80], StealthDB [82], SGX-Biniax2 [12], and a unit-test. The results show that the analysis covers from 91.4% to 96.6% of the *actions* in around 2 hours and 20 minutes in total (8146.11s). Furthermore, we did not observe any false positive during our experiments, meaning we covered a significant portion of code. In the right part of the table, we indicate the *actions* explored adopting *only* static or symbolic execution (*symex*) and their difference.

Use case	# func.	*action*		edge		*% action*	# func.	analysis time [s]			trade-off *actions* explored		
		μ	σ	μ	σ	explored	static	μ	σ	total	*static*	*symex*	Δ(%)
Contact [7]	71	12.77	12.59	15.09	17.64	96.4%	1	20.20	85.9	1397.12	1042	998	4.41
libdvdcss [80]	56	18.50	18.98	23.84	26.06	91.4%	9	70.19	179.65	3790.19	904	747	21.02
StealthDB [82]	44	18.29	13.53	21.97	18.05	96.6%	0	6.16	24.5	258.89	967	1009	−4.16
SGX-Biniax2 [12]	49	8.55	8.75	9.29	11.71	91.6%	4	52.46	168.8	2465.62	451	413	9.20
Unit-test	17	6.88	7.47	7.17	10.52	94.0%	0	15.60	53.4	234.29	122	107	14.02
total	237	-	-	-	-	-	14	-	-	8146.11	3486	3274	6.48

7.2.4 Model Extractor. We analyze the Model Extractor (Section 5.5). Specifically, we measure coverage and precision.

Use cases complexity: As stated in introduction, we assume the enclave's code is *simple* enough to be modeled with a combination of symbolic execution and static analysis (Section 5.5). The concept of *simple enclave* has already appeared in previous works [21, 73], however, they did not provide comparable metrics. In Table 2, we show a set of metrics that describe the software analyzed in our use cases. Specifically, we indicate the line of code (LoC), the number of secure functions, and the cyclomatic complexity [29]. We additionally measure the control-flow graph for each enclave's function and report the average (and standard deviation) number of nodes and edges per function. Similar metrics have been previously used to indicate the effectiveness of symbolic execution to explore a piece of software [10]. Finally, we count the number of direct and indirect function calls as the most important for the security guarantee. Intuitively, the less indirect calls an enclave has, the less likely an adversary can carry out a mimicry attack (*e.g.,* COOP [67]). One may argue that, since we assume an enclave with few indirect calls, then bound checks can effectively stop the memory corruption attacks. However, previous works [21] showed that a compromised OS can input *malicious* pointers to internal enclave structures. This allows an adversary to overwrite internal enclave data structures even with boundary checks in place. Therefore, using only bounds checks do not eradicate the problem in SGX enclaves, even for *simple* ones.

Coverage: In the context of SgxMonitor, the *action* coverage is a suitable metric for estimating the quality of an extracted model. This comes from two observations. First, assuming a sound symbolic execution, if no timeout is reached (*e.g.,* 10 minutes), we can state the analysis covered meaningful *actions*. We measure this with the percentage of traversed *actions* (over 91.4% in our experiments). Conversely, if the symbolic execution times out, we fallback to an insensitive static analysis. This traverses all the CFG of a function, thus completing the exploration of the *actions*. Of course, being the analysis insensitive, we trade-off precision for a low overhead in the construction of the model: we might observe rogue *actions*, which potentially increase the attack's surface.

Table 3 shows our coverage results. We applied the analysis described in Section 5.5 to our uses cases: Contact, libdvdcss, StealthDB, SGX-Biniax2, and the unit-test. The five use cases show a varying degree of complexity; Contact contains the highest number of single functions (71) among our use cases that are however quite simple (12 *actions* on average). Conversely, StealthDB has fewer (44) but more complex (18 *actions* on average) functions. libdvdcss and SGX-Biniax2 have a complexity similar to StealthDB (18.29 and 8.55 *actions* on average, respectively). Finally, the unit-test is self-contained and primarily leveraged to validate SgxMonitor and *exception handling* of enclaves. Overall, our analysis covers from 91.4% to 96.6% of the *actions*.

Precision: We want to inspect if the unexplored *actions* caused by symbolic execution timeout may cause false positives. To this end, we extract three models for each use case, namely: *symex*, by using only symbolic execution and interrupting the exploration once reached timeout; *static*, by using only insensitive static analysis; and *symex+static*, which is the one described in Section 5.5. Using only *symex* models, two secure functions in Contact generate *false positives*, this due to the function `crecip` that was not explored completely. Moreover, we observe similar cases in SGX-Biniax2 and libdvdcss, in which critical functions for crypting/decrypting were not correctly explored with only *symex*. We register false positives also using *static* models, in particular, one secure function in StealthDB gave false positive because of a `jmp` not correctly resolved (see the previous paragraph). Finally, *symex+static* models did not generate any *false positive* when compared with all our tests, thus showing that the combination of *symex+static* can significantly model the enclave behavior. Specifically, we stress libdvdcss, StealthDB, and SGX-Biniax2 with long macro-benchmarks (see Section 7.2.3). For Contact and the unit-test, we first run our micro-benchmarks, without observing any false positives. Then, we also manually investigated the cause of the unexplored *actions*. In most of the cases, pruned *actions* are corner cases that never happen in real executions (*e.g.,* a function that tests a null-pointer that never happens).

Notably, the exception handler mechanism of Intel SGX SDK always introduces a few non-traversed *actions*. This is caused by the routine `internal_handle_exception` that relies on a list of pointers created at runtime. Our Model Extractor automatically infers this structure and resolves the indirect call in `internal_handle_exception` (further details in Appendix A.3). Therefore, our Model Extractor automatically prunes those paths that never appear at runtime, *i.e.,* if the enclave does not contain custom handlers, it will never execute part of `internal_handle_exception`.

To sum up, our precision analysis shows that the combination of the symbolic execution and the insensitive static analyses achieve no false positives in our use cases, *i.e.,* there are no legal *actions* that are erroneously flagged as an instance of an attack.

Model Extractor—Take Away. Our results show that (i) the symbolic execution is suitable to cover the small functions in SGX enclaves (*i.e.,* only 14 functions out of 237 (5.9%) required an insensitive static analysis) and effectively cuts out unused *actions* thus reducing the attack surface; (ii) the static analysis can support the symbolic one in case of timeout; (iii) our approach is practical since it can be completed in around an hour (*i.e.,* 60m for libdvdcss); and (iv) our analysis explores a significant portion of the code since it does not rise false positive alarms.

8 RELATED WORKS

SgxMonitor shares common points with different research areas. Here, we discuss previous provenance analysis works (Section 8.1), runtime RA schema (Section 8.2), and finally, SGX and memory-corruptions (Section 8.3).

8.1 Provenance Analysis

Many provenance tools are based on instrumentation to collect specific logs from diverse sources [49, 52, 53]. SgxMonitor applies provenance to a novel area, we gather information from an isolated enclave while the analysis runs in an zero-trust environment. We overcome this issue with a novel technique to collect enclave runtime fine-grain information in the presence of a malicious OS.

Other provenance techniques focus on long term intrusion, such as APT [36, 89]. In our scenario, instead, we focus on code-reuse attacks that affect SGX enclaves. SgxMonitor helps an analyst to rebuild the intrusion by leveraging on a novel model suited for enclaves. SgxMonitor shares some similarities with runtime provenance works [62] that rely on a healthy OS to collect and analyze logs. Conversely, SgxMonitor assumes a malicious OS that might tamper with these operations.

Overall, SgxMonitor is the first provenance analysis suitable for the challenging SGX environment, providing runtime provenance analysis. To achieve this, we design a novel log collection and propose a novel model to represent the normal behavior of an enclave.

8.2 Runtime Remote Attestation

Extracting and verifying runtime information remotely is similar to Runtime Remote Attestation works [4, 5, 45, 72]. However, SgxMonitor underlies different assumptions and goals compared to these works. First, runtime remote attestation works are meant to detect intrusion at runtime. Conversely, SgxMonitor aims at collecting and information that could be analyzed later on. Since, all the previous works assume having an isolated trusted anchor to inspect the target (*i.e.,* the enclave). This is not possible in SGX since this feature is precluded by design. We overcome this limitation with a novel pure software design. Finally, the model employed by previous works are not suitable for SGX enclaves. SgxMonitor, instead, uses a novel model to capture and describe the enclave execution.

In GuaranTEE [57], the authors propose a runtime attestation for SGX. However, their model is stateless and cannot identify advanced malware such as SnakeGX. On the contrary, both model and design of SgxMonitor are designed to cover a broader attacker model, moreover, we performed a more comprehensive security evaluation.

8.3 SGX and Memory Corruption Errors

CFIs and shadow stacks [26, 28, 39, 43, 51] are orthogonal defenses to SgxMonitor and complement the protection of enclaves. In addition, one can remove corruptions errors in SGX enclaves, as studied in several forms [21, 46, 54, 66, 83]. All these works can be considered orthogonal to SgxMonitor since they contribute to reduce the attack surface. However, these solutions do not provide information about the intrusion. SgxMonitor, instead, helps one rebuild the cause of an attack.

9 CONCLUSION

We proposed SgxMonitor, a novel provenance analysis for SGX enclaves. As enclaves are designed to secure code that performs specific security- and privacy-sensitive tasks, SgxMonitor relies on a combination of symbolic execution and static analysis to model the expected behavior of enclaves with high code coverage and low false positives. Moreover, SgxMonitor designs a novel protocol to securely extract runtime enclave information in the presence of an adversarial OS.

We assessed SgxMonitor security properties against novel SGX code-reuse attacks. Moreover, we tested SgxMonitor across four real use cases (*i.e.,* Contact, StealthDB, libdvdcss, SGX-Biniax2) and a unit test to validate enclaves' corner cases.

SgxMonitor overhead is similar to the state-of-the-art provenance analysis works showing low macro-benchmark overhead and high precision with 96% code coverage and zero false positives support SgxMonitor in realistic deployments to extract insight about runtime anomalous executions of SGX enclaves.

REFERENCES

[1] 2013 (accessed September 2020). *Intel® Software Guard Extensions (Intel®SGX) - Developer Guide.* https://download.01.org/intel-sgx/linux-2.1.3/docs/Intel_SGX_Developer_Guide.pdf.

[2] . 2020 (accessed September 24, 2020). *Build SGX Enclave using Clang/LLVM.* https://community.intel.com/t5/Intel-Software-Guard-Extensions/Build-SGX-Enclave-using-Clang-LLVM/td-p/1161847.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

[4] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 743–754.

[5] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. [n.d.]. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. ([n. d.]).

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. 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. Citeseer, 7.

[7] Signal App. 2017 (accessed September 14, 2020). *Private Contact Discovery Service (Beta).* https://github.com/signalapp/ContactDiscoveryService.

[8] Signal App. 2017 (accessed September 14, 2020). *Signal App.* https://signal.org/en/.

[9] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. 2013. BLAKE2: simpler, smaller, fast as MD5. In *International Conference on Applied Cryptography and Network Security.* Springer, 119–135.

[10] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. 2017. Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. In *26th USENIX Security Symposium (USENIX Security 17).* USENIX Association, Vancouver, BC, 661–678. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/banescu

[11] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. 2011. Beyond provable security verifiable IND-CCA security of OAEP. In *Cryptographers' Track at the RSA Conference.* Springer, 180–196.

[12] Erick Bauman and Zhiqiang Lin. 2016. A case for protecting computer games with SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution.* 1–6.

[13] Mihir Bellare, David Cash, and Rachel Miller. 2011. Cryptography Secure against Related-Key Attacks and Tampering. In *Advances in Cryptology – ASIACRYPT 2011*, Dong Hoon Lee and Xiaoyun Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 486–503.

[14] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The guard's dilemma: Efficient code-reuse attacks against intel sgx. In *Proceedings of 27th USENIX Security Symposium.*

[15] Dan Bogdanov. 2018 (accessed September 14, 2020). *Dashlane and Intel join forces to bring built-in password protection to PCs.* https://sharemind.cyber.ee/introducing-sharemind-hi/.

[16] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17).* USENIX Association, Vancouver, BC. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

[17] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 289–300.

[18] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses.. In *USENIX Security Symposium.* 385–399.

[19] Chia che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, 505–522. https://www.usenix.org/conference/usenixsecurity20/presentation/tsai

[20] Shuo Chen, Jun Xu, and Emre C. Sezer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *14th USENIX Security Symposium (USENIX Security 05).* USENIX Association, Baltimore, MD. https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats

[21] Tobias Cloosters, Michael Rodler, and Lucas Davi. 2020. TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves. In *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, 841–858. https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters

[22] Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Rethinking pointer reasoning in symbolic execution. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 613–618.

[23] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[24] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. 2021. SmashEx: Smashing SGX Enclaves Using Exceptions. *arXiv preprint arXiv:2110.06657* (2021).

[25] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.

[26] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *26th {USENIX} Security Symposium ({USENIX} Security 17).* 131–148.

[27] Danny Dolev and Andrew Yao. 1983. On the security of public key protocols. *IEEE Transactions on information theory* 29, 2 (1983), 198–208.

[28] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. 2017. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* 37, 2 (2017), 52–62.

[29] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. 2016. Cyclomatic complexity. *IEEE software* 33, 6 (2016), 27–29.

[30] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15).* Association for Computing Machinery, New York, NY, USA, 901–913. https://doi.org/10.1145/2810103.2813646

[31] Toffalini Flavio, Graziano Mariano, Conti Mauro, and Zhou Jianying. 2021. SnakeGX: a sneaky attack against SGX Enclaves. In *International Conference on Applied Cryptography and Network Security.*

[32] Fortanix. 2018 (accessed September 14, 2020). *Secure enclaves & Intel®SGX.* https://edp.fortanix.com/docs/concepts/sgx/.

[33] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security* (Belgrade, Serbia) *(EuroSec'17).* Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3065913.3065915

[34] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, J Guilford, and Gil Wolrich. 2013. Intel sha extensions–new instructions supporting the secure hash algorithm on intel architecture processor. *Intel White Paper* (2013).

[35] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17).* USENIX Association, Santa Clara, CA, 299–312. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel

[36] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. [n.d.]. Unicorn: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *Proceedings 2020 Network and Distributed System Security Symposium* (San Diego, CA, 2020). Internet Society. https://doi.org/10.14722/ndss.2020.24046

[37] Roarke Horstmeyer, Benjamin Judkewitz, Ivo M Vellekoop, Sid Assawaworrarit, and Changhuei Yang. 2013. Physical key-protected one-time pad. *Scientific reports* 3 (2013), 3543.

[38] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic generation of data-oriented exploits. In *24th {USENIX} Security Symposium ({USENIX} Security 15).* 177–192.

[39] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 1470–1486.

[40] Intel. 2014 (accessed September 14, 2020). *Intel® Software Guard Extensions Programming Reference.* https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[41] Hassaan Irshad, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Kyu Hyung Lee, Jignesh Patel, Somesh Jha, Yonghwi Kwon, Dongyan Xu, and Xiangyu Zhang. [n.d.]. TRACE: Enterprise-Wide Provenance Tracking for Real-Time APT Detection. 16 ([n. d.]), 4363–4376. https://doi.org/10.1109/TIFS.2021.3098977 Conference Name: IEEE Transactions on Information Forensics and Security.

[42] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[43] Andi Kleen and Beeman Strong. 2015. Intel processor trace on linux. *Tracing Summit* 2015 (2015).

[44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[45] Nikos Koutroumpouchos, Christoforos Ntantogian, Sofia-Anna Menesidou, Kaitai Liang, Panagiotis Gouvas, Christos Xenakis, and Thanassis Giannetsos. 2019. Secure edge computing with lightweight control-flow property-based attestation. In *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 84–92.

[46] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*. 205–221.

[47] Oasis Labs. 2018 (accessed September 14, 2020). *Cleanrooms & Secure Enclaves*. https://www.oasislabs.com/cleanrooms-secure-enclaves.

[48] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent B Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*. 523–539.

[49] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *NDSS*. 16.

[50] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 285–298. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind

[51] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1867–1881. https://doi.org/10.1145/3319535.3354244

[52] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. {MPI}: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 1111–1128.

[53] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. Protracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting.. In *NDSS*.

[54] Shachee Mishra and Michalis Polychronakis. 2021. SGXPecial: Specializing SGX Interfaces against Code Reuse Attacks. In *Proceedings of the 14th European Workshop on Systems Security*. 48–54.

[55] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 69–90.

[56] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

[57] Mathias Morbitzer, Benedikt Kopf, and Philipp Zieris. 2022. GuaranTEE: Introducing Control-Flow Attestation for Trusted Execution Environments. *arXiv preprint arXiv:2202.07380* (2022).

[58] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. 2013. Handling Unbounded Loops with ESBMC 1.20. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 619–622.

[59] Moxie0. 2017 (accessed September 14, 2020). *Technology preview: Private contact discovery for Signal*. https://signal.org/blog/private-contact-discovery/.

[60] Malaika Nicholas. 2017 (accessed September 14, 2020). *Dashlane and Intel join forces to bring built-in password protection to PCs*. https://blog.dashlane.com/dashlane-intel-sgx-bring-built-password-protection-to-pcs/.

[61] VideoLAN organization. 2009 (accessed September 24, 2020). *VLC media player*. https://www.videolan.org/.

[62] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. [n.d.]. Runtime Analysis of Whole-System Provenance. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto Canada, 2018-10-15). ACM, 1601–1616. https://doi.org/10.1145/3243734.3243776

[63] Reese T. Prosser. 1959. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference* (Boston, Massachusetts) *(IRE-AIEE-ACM '59 (Eastern))*. Association for Computing Machinery, New York, NY, USA, 133–138. https://doi.org/10.1145/1460299.1460314

[64] Carlos Rozas. 2013. Intel® Software Guard Extensions (Intel® SGX). (2013).

[65] Dipanwita Sarkar, Muthu Jagannathan, Jay Thiagarajan, and Ramanathan Venkatapathy. 2007. Flow-insensitive static analysis for detecting integer anomalies in programs. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*. ACTA Press, 334–340.

[66] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 38–54.

[67] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 745–762.

[68] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. 2020. BesFS: A POSIX Filesystem for Enclaves with a Mechanized Safety Proof. In *USENIX Security*.

[69] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. (2015).

[70] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 875–892. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/strackx

[71] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. 22–27.

[72] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. 2019. ScaRR: Scalable Runtime Remote Attestation for Complex Systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 121–134. https://www.usenix.org/conference/raid2019/presentation/toffalini

[73] Flavio Toffalini, Martín Ochoa, Jun Sun, and Jianying Zhou. 2019. Careful-Packing: A Practical and Scalable Anti-Tampering Software Protection enforced by Trusted Computing. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. 231–242.

[74] Flavio Toffalini, Andrea Oliveri, Mariano Graziano, Jianying Zhou, and Davide Balzarotti. 2021. The evidence beyond the wall: Memory forensics in SGX environments. *Forensic Science International: Digital Investigation* 39 (2021), 301313. https://doi.org/10.1016/j.fsidi.2021.301313

[75] Jordan Tuzsuzov. 2005 (accessed April 4, 2021). *Biniax-2*. http://www.tuzsuzov.com/biniax/index2.html.

[76] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 54–72.

[77] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1741–1758.

[78] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 1041–1056.

[79] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2019. SoK: Benchmarking flaws in systems security. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 310–325.

[80] VideoLAN. 2017 (accessed September 14, 2020). *libdvdcss*. https://code.videolan.org/videolan/libdvdcss.

[81] Hiie Vill. 2017. SGX attestation process.

[82] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a scalable encrypted database with full SQL query support. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 370–388.

[83] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2333–2350. https://doi.org/10.1145/3319535.3354241

[84] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution.. In *NDSS*. Citeseer.

[85] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2421–2434. https://doi.org/10.1145/3133956.3134038

[86] Hoeteck Wee. 2010. Efficient chosen-ciphertext security via extractable hash proofs. In *Annual Cryptology Conference*. Springer, 314–332.

[87] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[88] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on*

Security and Privacy. 640–656.
[89] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E Urias, Han Wei Lin, Gabriela Ciocarlie, Vinod Yegneswaran, et al. 2021. ALchemist: Fusing Application and Audit Logs for Precise Attack Provenance without Instrumentation. (2021).
[90] H. Zhou, K. Kang, and J. Yuan. 2019. HardStack: Prevent Stack Buffer Overflow Attack with LBR. In *2019 International Conference on Intelligent Computing, Automation and Systems (ICICAS)*. 888–892.

A MODEL EXAMPLES

In this section, we discuss the application of SgxMonitor model (Section 5) over two important Intel SGX SDK mechanisms: the outside function interaction (Section A.1) and the exception handling (Section A.2).

Transaction syntax. For the sake of simplicity, we indicate the transactions in tables 7a and 8a with the following syntax:

$$T = P \cup [s].$$

T is composed of any *valid* sequence of *generic actions* P (according to the specification of Section 5) that terminates with the *stop action* s. In case T does not contain any *generic action*, we omit P.

A.1 Outside Function Modeling

Figure 7 shows the application of SgxMonitor to the enclave *outside function* interaction.

After the enclave initialization, the host invokes a *secure function*, which activates an `EENTER` opcode with the `idx` greater or equal than *zero* (*i.e.,* T^{ECALL}). From this point, the *secure function* can evolve in two ways: (E1) it does not need any interaction with the host, thus it performs an ERET; or (E2) it requires an interaction with the host, thus it performs an ORET. In case (E1), the enclave does not generate any context and, therefore, it performs a valid execution path that ends with an `EEXIT` opcode (*i.e.,* T^{ERET}). In case (E2), instead, we need two steps to accomplish an OCALL: (i) generating an `ocall_context` (*i.e.,* T^{OCALL1}), and (ii) invoking the *outside function* (*i.e.,* T^{OCALL2}).

Once the *outside function* needs to resume the *secure function* execution, it invokes an ORET, that is composed of two steps: (i) the execution enters in the enclave (*i.e.,* T^{ORET1}), and (ii) the `ocall_context` is restored (*i.e.,* T^{ORET2}). From this point ahead, the *secure function* can exit the enclave through an ERET (E1) or perform further OCALLs (E2).

A.2 Exception Handling Modeling

In Figure 8b, we depict the SgxMonitor representation of the SGX SDK exception handling. Overall, the SGX SDK handles exceptions in two phases, called *trusted handle* (TH) and *internal handle* (IH), respectively. In the first phase (TH), the SGX interrupts its execution as a result of an `AEX`, and passes the control to the host. As soon as an exception is triggered, the microcode saves the CPU registers in a dedicated page, called SSA, for later stages [23]. After an `AEX`, the SDK expects the invocation of a dedicated *secure function*, called `trts_handle_exception`, which index is −3 (*i.e.,* T^{THD1}). This function fills an `sgx_exception_info_t` structure with the values previously stored in the SSA (*i.e.,* T^{THD2}). At the end of (TH), the enclave is ready for the second phase (IH) and thus it leaves the control to the host (*i.e.,* T^{THD3}). The host invokes an `ERESUME` to activate the `internal_handle_exception` routine (*i.e.,* $\mathrm{T}^{\text{ERESUME}}$). Now, the enclave iterates among the custom handlers eventually registered (*i.e.,* T^{IHD1} and T^{IHD2}). Each custom handler attempts at fixing the exception by analyzing the `sgx_exception_info_t`, possibly altering it. Therefore, we update the enclave internal state at each iteration. After invoking all the internal handlers, the SGX SDK uses the `continue_execution` routine to resume the *secure function* (*i.e.,* T^{CONT}). Finally, if the exception is properly handled, the *secure function* will continue, otherwise, a new `AEX` happens and the exception workflow starts again.

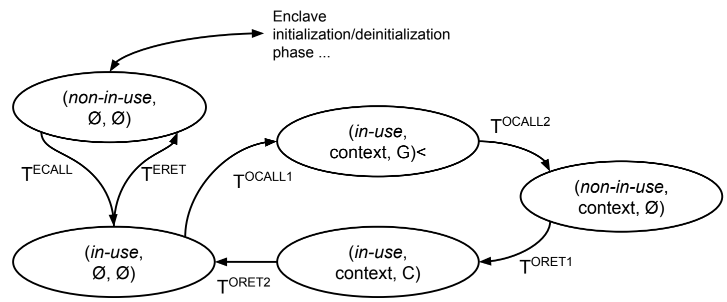
A.3 SGX SDK Exception Handling

In the following, we show an example of registration of a custom exception handler, that happens by invoking the function `sgx_register_exception_handler`. The enclave passes the address of the exception handler as an argument, *e.g.,* `divide_by_zero_handler`. The Model Extractor (Section 5.5) parses the enclave code and identifies all the `sgx_register_exception_handler` invocations. Then, it performs a taint analysis to infer the address of the custom exception handler passed as second parameter to `sgx_register_exception_handler`. Finally, it uses this information to build a symbolic structure that will be used to explore the function `internal_handle_exception`, that actually dispatches the exception to the correct handler, if any.

```
if (sgx_register_exception_handler
      (1, divide_by_zero_handler) == NULL) {
    printf("register failed\n");
} else {
    printf("register success\n");
}
```

Transaction	Definition
T^{ECALL}	$[(\text{N},\texttt{src},\texttt{idx})_{\texttt{idx}\geq 0}]$
T^{ERET}	$P\cup[(\text{T},\texttt{src},\oslash)]$
T^{OCALL1}	$P\cup[(\text{G},\texttt{src},\texttt{ctx})]$
T^{OCALL2}	$P\cup[(\text{D},\texttt{src},\oslash)]$
T^{ORET1}	$[(\text{N},\texttt{src},\texttt{idx})_{\texttt{idx}=-2}]$
T^{ORET2}	$P\cup[(\text{C},\texttt{src},\texttt{ctx})]$

(a) Transaction definition of SgxMonitor model for the *outside function* interaction.

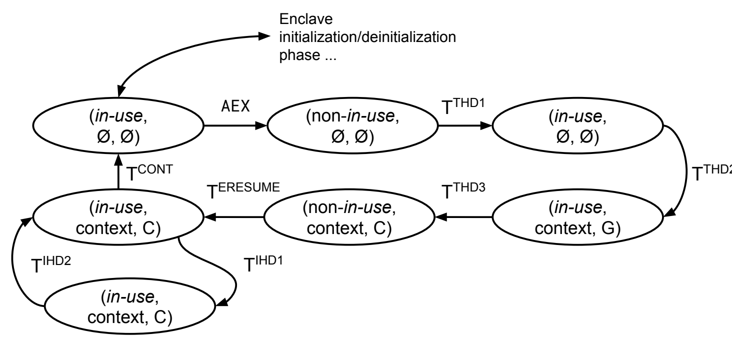


(b) SgxMonitor representation of *outside functions* interaction.

Figure 7: Example of *outside functions* interaction modeling. We show the FSM representation and the transaction definitions, respectively.

Transaction	Definition
AEX	*handled at microcode level*
T^{THD1}	$[(\text{N},\texttt{src},\texttt{idx})_{\texttt{idx}=-3}]$
T^{THD2}	$P\cup[(\text{J},\texttt{src},\texttt{ctx})]$
T^{THD3}	$P\cup[(\text{T},\texttt{src},\oslash)]$
T^{ERESUME}	$P\cup[(\text{R},\texttt{src},\oslash)]$
T^{IHD1}	$P\cup[(\text{K},\texttt{src},\texttt{ctx})]$
T^{IHD2}	$P\cup[(\text{J},\texttt{src},\texttt{ctx})]$
T^{CONT}	$P\cup[(\text{K},\texttt{src},\texttt{ctx})]$

(a) Transaction definition of SgxMonitor model for the exception handling interaction.



(b) SgxMonitor representation of exception handling.

Figure 8: Example of *exception handling* modeling. We show the FSM representation and the transaction definitions, respectively.