Designing a Provenance Analysis for SGX Enclaves

Flavio Toffalini  Mathias Payer  Jianying Zhou  Lorenzo Cavallaro
EPFL  EPFL  SUTD  UCL
Switzerland  Switzerland  Singapore  United Kingdom
flavio.toffalini@epfl.ch mathias.payer@nebelwelt.net jianying_zhou@sutd.edu.sg l.cavallaro@ucl.ac.uk

ABSTRACT

SGX enclaves are trusted user-space memory regions that ensure isolation from the host, which is considered malicious. However, enclaves may suffer from vulnerabilities that allow adversaries to compromise their trustworthiness. Consequently, the SGX isolation may hinder defenders from recognizing an intrusion. Ideally, to identify compromised enclaves, the owner should have privileged access to the enclave memory and a policy to recognize the attack. Most importantly, these operations should not break the SGX properties.

In this work, we propose SgxMonitor, a novel provenance analysis to monitor and identify compromised enclaves. SgxMonitor is composed of two elements: (i) a technique to extract contextual runtime information from an enclave, and (ii) a novel model to recognize enclaves’ intrusions. Our evaluation shows that SgxMonitor successfully identifies enclave intrusions against state-of-the-art attacks without undermining the SGX isolation. Our experiments did not report false positives and negatives during normal enclave executions, while incurring a marginal overhead that does not affect real use cases deployment, thus supporting the use of SgxMonitor in realistic scenarios.

KEYWORDS

TEE, SGX, provenance analysis

ACM Reference Format:

1 INTRODUCTION

Intel Software Guard eXtension (SGX) is an ISA abstraction that allows developers to define enclaves [41, 64], small user-space regions isolated from the underlying untrusted OS. 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.

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 pre-boot TPM static code measurement. However, SGX alone has no mechanisms to guarantee the correct runtime execution of enclaves, which remain vulnerable against confused deputy attacks that cause 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 countering 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 [21]. In this regard, having information about the attack vector becomes crucial for improving the defenses. In normal scenarios (e.g., OSes) one can employ provenance analyses [36, 42, 62, 89] based on streams of events (e.g., system logs, syscall invocation). However, SGX disallows standard monitoring mechanisms (e.g., Intel PT [44] or Intel LBR [28, 90]) a priori [74], thus hindering the adoption of these approaches. Recent works [91] propose techniques to dump arbitrary enclave memory regions in a secure fashion, however, these mechanisms do not provide a continuous tracing and may leave room for attacks.

Provenance techniques for SGX need to deal with two challenges: (i) streaming information out of an enclave without introducing undesired side effects, and (ii) a model to identify an attack from the information gathered. We address these challenges with SgxMonitor: a system to allow an external (and legitimate) entity to inspect an enclave runtime state, retrieve evidence of intrusion, and not undermining the SGX isolation. To achieve this, we first design a secure tracing mechanism for SGX enclaves, and second, propose a model to represent useful intrusion information. Our monitor combines 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. Most importantly, our monitor is designed not to amplify other attack vectors such as side-channels. For detecting intrusions, we propose a novel Finite-State Machine (FSM) that extends the current models used in SGX [23]. We automatically build the enclave model through a combination of symbolic execution and insensitive static analysis to create an FSM of the code in an enclave. 1

Intuitively, an enclave deviating from its FSM gives insights 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 [21]. In this regard, having information about the attack vector becomes crucial for improving the defenses. In normal scenarios (e.g., OSes) one can employ provenance analyses [36, 42, 62, 89] based on streams of events (e.g., system logs, syscall invocation). However, SGX disallows standard monitoring mechanisms (e.g., Intel PT [44] or Intel LBR [28, 90]) a priori [74], thus hindering the adoption of these approaches. Recent works [91] propose techniques to dump arbitrary enclave memory regions in a secure fashion, however, these mechanisms do not provide a continuous tracing and may leave room for attacks.

Provenance techniques for SGX need to deal with two challenges: (i) streaming information out of an enclave without introducing undesired side effects, and (ii) a model to identify an attack from the information gathered. We address these challenges with SgxMonitor: a system to allow an external (and legitimate) entity to inspect an enclave runtime state, retrieve evidence of intrusion, and not undermining the SGX isolation. To achieve this, we first design a secure tracing mechanism for SGX enclaves, and second, propose a model to represent useful intrusion information. Our monitor combines 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. Most importantly, our monitor is designed not to amplify other attack vectors such as side-channels. For detecting intrusions, we propose a novel Finite-State Machine (FSM) that extends the current models used in SGX [23]. We automatically build the enclave model through a combination of symbolic execution and insensitive static analysis to create an FSM of the code in an enclave. 1

Intuitively, an enclave deviating from its FSM gives insights 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 [21]. In this regard, having information about the attack vector becomes crucial for improving the defenses. In normal scenarios (e.g., OSes) one can employ provenance analyses [36, 42, 62, 89] based on streams of events (e.g., system logs, syscall invocation). However, SGX disallows standard monitoring mechanisms (e.g., Intel PT [44] or Intel LBR [28, 90]) a priori [74], thus hindering the adoption of these approaches. Recent works [91] propose techniques to dump arbitrary enclave memory regions in a secure fashion, however, these mechanisms do not provide a continuous tracing and may leave room for attacks.

Provenance techniques for SGX need to deal with two challenges: (i) streaming information out of an enclave without introducing undesired side effects, and (ii) a model to identify an attack from the information gathered. We address these challenges with SgxMonitor: a system to allow an external (and legitimate) entity to inspect an enclave runtime state, retrieve evidence of intrusion, and not undermining the SGX isolation. To achieve this, we first design a secure tracing mechanism for SGX enclaves, and second, propose a model to represent useful intrusion information. Our monitor combines 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. Most importantly, our monitor is designed not to amplify other attack vectors such as side-channels. For detecting intrusions, we propose a novel Finite-State Machine (FSM) that extends the current models used in SGX [23]. We automatically build the enclave model through a combination of symbolic execution and insensitive static analysis to create an FSM of the code in an enclave. 1

1 For simplicity, we use the term insensitive static analysis to refer to a context- and path-insensitive analysis throughout the paper.
data-only malware for SGX enclaves, and specifically-crafted security benchmarks (§7.1.1). Moreover, we discuss if our solution braces the attacker surface of SGX enclaves (§7.1.2). To assess whether SgxMonitor is usable in practice, we deploy it across five use cases and measure micro- and macro-benchmark (§7.2). Our results show SgxMonitor incurs in an overhead between 1.6% and 10% for macro-benchmark, which is in line with the state-of-the-art.

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 (§4); (ii) a stateful representation of the SGX enclaves runtime properties (§5).
- We assess the security properties of SgxMonitor against SnakeGX and specifically-crafted security benchmarks (§7.1.1). Moreover, we conduct a security analysis of our design (§7.1.2).
- We likewise evaluate the usability of SgxMonitor by measuring micro/macro-benchmark, and the completeness of our model (§7.2).

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 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 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

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], e.g., overwriting forward (function pointers) or backward jumps (return instructions), (ii) Altering the network enclave communication by overhearing, intercepting, and forging packets such as the Dolev Yao attacker [27]. Finally, we assume a scenario in which an adversary takes control of the victim enclave. In this case, SgxMonitor must report sufficient correct information to describe the intrusion.

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

Out-of-Scope Attacks. We exclude data-only attacks, i.e., memory corruptions that do not hijack the execution flow. Moreover, we assume the CPU is correctly implemented, thus not prone to rollback attacks [70] or micro-architectural vulnerabilities [35, 45, 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

Natively, SGX forbids any external observer to inspect enclave’s content. With SgxMonitor, we allow an enclave to securely stream runtime fine-grain information, namely actions, similarly to Intel PT. Intuitively, actions represent meaningful enclave events (e.g., control-flow transfers, functions invoked) that enable an outside monitor to recognize an intrusion. Our system plays a crucial role since it has to transfer (potential) sensitive information without amplifying the attacker capabilities. This section focuses on the technical description, while we conduct a security analysis in §7.1.2.

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 §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.

Recalling our threat model (§3), T or its host may be compromised. Since SGX disallows enclave memory segmentation, SgxMonitor may report attacker-controlled actions after an intrusion. To cope with this issue, we design SgxMonitor to guarantee that, at least, the first action representing the attack is correctly reported. Additionally, we move M into a separate host to reduce the likelihood of
compromising M. R is a legitimate remote entity that desires to validate the integrity of T, we ensure R’s trustworthiness by employing the standard SGX RA [6] (see §2).

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 (➊). Then, we seal D to prevent a malicious host to tamper with it (➋). 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 §4.2 (➌). 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 verifying the actions A adhere to the model D (➍). The Model Extractor (➊) and Verifier (➋), along with further model details, are described in §5.5 and §5.6, respectively.

Once M correctly receives A from T, R uses the SGX RA to communicate with T and M. Specifically, R uses the SGX RA to verify the software integrity and the identity of T (➏). Likewise, R uses the SGX RA to attest the identity of M and inquiry the runtime state of T, i.e., if T still follows the model D and, in case, where the model diverges and how (➋).

4.1 Action Reporting Mechanism

T relies on an action reporting mechanism that is resilient against the threat model described in §3: an intrusion inside T (e.g., exploiting a T internal error), and a malicious host. More precisely, we guarantee SgxMonitor reports at least the first action of an intrusion.

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 (§4.2). This section mainly focuses on the reporting mechanism, while a complete description of actions is presented in §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 (e.g., a return instruction or an indirect call). We exemplify this mechanism in Figure 9, and leave further examples in Appendix D. In Figure 9, the program traces an action representing a return edge to the caller (Line 4). 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 (§4.2) that avoids forging and tampering with already reported actions. Therefore, an adversary cannot hijack T without reporting evidence about the attack.

Our instrumentation is robust against attempts of overwriting trace( ) due to the SGX security properties. Here, we 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 (§4.2), provides the base for our resilient provenance analysis (more info in §7.1.2).

4.2 Secure Communication Protocol

T and M exchange actions relying on a secure communication channel that ensures three 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; and
Algorithm 1: Procedure used by the target enclave to report logs in a secure fashion.

1. `reportLog(A)`
2. \( \text{mac} \leftarrow H_1(A|K) \)
3. \( C \leftarrow (A|\text{mac}) @K \)
4. \( K \leftarrow H_2(K) \)
5. `write(C)`

(iii) the protocol does not facilitate side-channel attacks. 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. Our intuition is that each action is encrypted with a different key, and new keys are obtained from chaining previous ones \[40\] (i.e., \( K_t = \text{hash}(K_{t-1}) \)). Therefore, each action is encrypted with a fresh key. Moreover, leaking the current key (e.g., due to an intrusion) does not reveal information about the previous keys used. We now show how to achieve this goal.

Protocol establishment. The channel requires three steps to be established (\( \mathbf{3} \) in Figure 1): (i) T and M issue a standard SGX RA [6] to verify their identity; (ii) M sends a secure key \( K \) to T using the SGX RA; and (iii) T uses the key \( K \) to send the actions to M. Importantly, after the step (ii) M and T are synchronized with the same key \( K \). Moreover, 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 communications. The adoption of a shared key \( K \) avoids an adversary to use the insecure channel (algorithm 1 line 5). Therefore, the execution steps (ii) and (iii) can be synchronized.

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

1. `verifyLog(C)`
2. \((A|\text{mac}) \leftarrow C @K \)
3. \( \text{mac}' \leftarrow H_1(A|K) \)
4. if `mac' != `mac` then
5. \( \rightarrow \) untrusted()
6. else
7. \( \rightarrow \) process(A)
8. end
9. \( K \leftarrow H_2(K) \)

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

5 SGXMONITOR: THE ENCLAVE MODEL

We model the normal enclaves’ behavior by extending the standard Finite-State Machine of SGX enclave life-cycle, which is shown in Figure 3.\(^3\) This model assumes the host interacts with a correctly loaded enclave by means of the opcodes in §2. The enclave state can assume only two values: `non-in-use` and `in-use`. In particular, an enclave transits to `in-use` state when an `ENTER` or `ERESUME` is issued. Then, the state returns to `non-in-use` when an `EXIT` or `AEX` happens. The microcode already implements this model in the microcode: the same thread cannot enter (i.e., `ENTER`) in an enclave which is already in `in-use` state; it cannot exit (i.e., `EXIT`) when the enclave is in `non-in-use`. However, this model does not provide fine-grain information about enclave health, i.e., an attack against the enclave execution [14, 31, 48] cannot be traced thus precluding provenance analysis 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 of four elements:

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

\(^{3}\)This model is a simplified version of [23].
In the rest of the section, we also describe the Model Extractor and Verifier in §5.5 and §5.6, respectively.

5.1 State Definition

Our model integrates important global structures used by the Intel SGX SDK to handle outside function invocation and exception handling (§2). These structures are targeted in known attacks [14, 48], thus reveal information about the tactic adopted for the intrusion.

Since SGX supports multi-threading, 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 assumes two values: in-use and non-in-use. Structure, instead, is an hash representation of the current structure used (or ⊘ if no structure is used). Finally, operation represents if the structure was generated (i.e., G), consumed (i.e., C), or null if no operation has been performed (i.e., ⊘).

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 handling. 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 (§5.4), that allow the evolution of the enclave state; and to build graphs of actions (§5.3), that we use to validate the runtime transactions. In particular, we distinguish two types of actions: generic and stop.

Generic actions. They identify standard software behaviors such as: (i) control-flow events; e.g., jmp, call, ret; (ii) conditional branches (e.g., jcc); and (iii) function pointer/virtual table assignment. Generic actions do not alter the state of the enclave and identify correct executions [28, 39, 44, 72, 90].

Stop actions. We consider SGX opcodes and structure manipulation that alter the state of the enclave. For 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 it turns to non-in-use otherwise. For structures manipulations, instead, we trace whenever the enclave generates or consumes a structure. These actions alter the fields structure and operation of the state; i.e., when an operation generates a structure, we store its 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:

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

where type identifies the nature of the action (e.g., function call, EENTER), src is the virtual address where action occurred, and value depends on the action semantic. For instance: value contains the callee address for a function call; a boolean value for conditional branches; or null if not required. Finally, cond contains extra conditions (e.g., $\text{value} \geq 0$). We provide the complete action list in Table 1 grouped by generic and stop.

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

<table>
<thead>
<tr>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic</strong></td>
</tr>
<tr>
<td>(E, src[0], dst[0])</td>
</tr>
<tr>
<td>(B, src, 0[1])</td>
</tr>
<tr>
<td>(A, src, addr)</td>
</tr>
<tr>
<td>(V, src, vptr)</td>
</tr>
<tr>
<td><strong>Stop</strong></td>
</tr>
<tr>
<td>(G, src, ctx)</td>
</tr>
<tr>
<td>(C, src, ctx)</td>
</tr>
<tr>
<td>(J, src, ctx)</td>
</tr>
<tr>
<td>(K, src, ctx)</td>
</tr>
<tr>
<td>(N, src, idx)</td>
</tr>
<tr>
<td>(R, src, ⊘)</td>
</tr>
<tr>
<td>(T, src, ⊘)</td>
</tr>
<tr>
<td>(D, src, ⊘)</td>
</tr>
</tbody>
</table>

5.3 Graphs of Actions Definition

Graphs of actions are composed of vertexes and edges, whose vertexes are in a bijective relationship with actions: 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.

The graph representation simplifies loops detection, that otherwise would 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 §5.5 and §5.6, respectively.

5.4 Transaction Definition

A transaction identifies a valid execution path in an enclave and is composed of a valid sequence of actions (§5.2) that makes the enclave state evolve. Formally, we indicate a transaction $P$ as following $P = [g_1,...,g_n,t]$, 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 §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 (○ in Figure 1) is to automatically infer the behavior for a given enclave. As preliminary approach, we investigated an interprocedural symbolic execution [43] over the entire enclave. Even though this strategy would benefit from a context to prune infeasible paths, we found it too resource-intensive, i.e., requiring hours of computations, encountering timeouts, and leading to incomplete explorations in practice. Another approach would use
insensitive static analysis [22] to extract the control-flow graphs of each function. However, this approach introduces impossible paths that increase the attacker surface. Therefore, we build on the benefits of symbolic execution and an insensitive static analysis while limiting their drawbacks (we expand this discussion in Appendix B).

In our scenario, we assume that the code in an enclave implements straightforward 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 relatively small number of indirect calls and its software base is given. Therefore, we take inspiration from previous compositional analyses [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.

The Model Extractor takes as input a target enclave T which has been instrumented at compilation time for tracing 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 [41].

Overall, the extraction algorithm is described in algorithm 3. Given an instrumented target enclave T, we analyze each instrumented function separately (algorithm 3 line 3). The rest of the section details each point of the analysis.

### Symbolic Global Variables (algorithm 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 for each function analyzed.

### Loop Analysis (algorithm 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 good coverage while keeping low false positive.

### Free Arguments Inferring (algorithm 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 argument, 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 function pointers by employing a conservative function type analysis [3].

#### Symbolic Exploration (algorithm 3 line 7)
We primarily employ a symbolic exploration [43] 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 (algorithm 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 (algorithm 3 line 11)**. The final enclave model is an association between functions and their model that is finally sealed in the monitor enclave host to avoid tampering.

### 5.6 Model Verifier
The Model Verifier (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., return 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 adhere 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 [41]. 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_ocal1 and asm_oret to handle ocall1_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 §4.2).

---

**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)
        r ← symbolicExploration(f)
        if r.isTimeout() then
            r ← insensitiveAnalysis(f)
        end
        m ← m ∪ (f,r,graph_of_action)
    end
    return m
```
Model Extractor. The Model Extractor is based on angr version 8.18 and implements the algorithms described in §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 §4.2) 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 reportTLog() and verifyfLog(), 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 adopt 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 §7.1 by testing the SgxMonitor security guarantees against a set of modern SGX attacks. We answer R2 in §7.2 by measuring micro/macro-benchmark, and discussing the model extraction.

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 (§7.1.1). Then, we illustrate a security analysis of the SgxMonitor design against a battery of protocol(side-channel)/non-control data attacks to prove our solution does not amplify such threats (§7.1.2).

7.1.1 Execution-flow attacks. We choose two security benchmarks: SnakeGX [31], which is an enclave SGX enclaves infector; and a benchmark that evaluates the shadow stack defense.

SnakeGX. This is a data-only malware designed to implant a permanent backdoor into legitimate SGX enclaves. 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 deploy SgxMonitor into the PoC delivered by the authors of SnakeGX, extract the model, and finally, analyze the actions reported. The results show that SgxMonitor recognizes either the installation phase and the backdoor activation. In particular, the installation relies on a classic ROP-chain, therefore, SgxMonitor identifies an unknown action pointing to a gadget. In this way, SgxMonitor gives an insight about an intrusion inside the enclave. The backdoor activation, instead, restores a corrupted ocall context (crafted during the installation). In this case, SgxMonitor observes the restoring an anomalous state. Notably, previous bug detection works did not identify the error design used in the installation phase [21]. Recent introspection works [91], instead, allow one to find traces of a payload. However, these works are requested-based, therefore, the analyst has to inspect the enclave at the right time to find the payload in memory. Conversely, SgxMonitor continuously traces the enclave, thus overcoming the limitation of requested-based introspection techniques.

Shadow stack protection. We evaluate the shadow stack implemented in SgxMonitor. In particular, we want to identify a corrupted return address that points to a valid function. To this end, we build a custom enclave that allows such attacks and deploy SgxMonitor in it. 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. Again, recent introspection works [91] require to dump the enclave context when the payload is still present in the enclave. However, code-reuse attacks are considered one-shot, meaning they do not leave consistent traces after their execution [31]. Therefore, the introspection must happen before the payload is activated, which we argue is unlikely in real cases. On the contrary, SgxMonitor does not suffer from this limitation due to the stream of actions reported.

Final Notes. We remark that standard mitigation deployed inside an enclave (e.g., CFI or shadow stacks) lacks any insight about the attack performed. Moreover, request-based introspection must catch the payload at the right timing. On the contrary, SgxMonitor provides a continuous stream of fine-grain information about the intrusion, that facilitates the detection.

7.1.2 Security Analysis of the System Design. We discuss the security properties of the SgxMonitor design (§4) with respect to our threat model (§3). Before discussing the following cases, we remark all the packets have the same size by design, and the cryptographic key changes at any packet reported (§4.2). Therefore, an adversary can only observe the packets’ timestamp.

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 against an adversary that hijacks T. In this case, our code instrumentation encrypts and reports the malicious action before the enclave traverses the hijacked edge (§4.1), thus producing a new key K (§4.2). Here, we face three scenarios: (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 reportTLog() 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
We dissect this scenario in two cases. (i) The adversary manages an SGX porting of the open-source game Biniax2 [75]; and (ii) a plugin that uses SGX to encrypt tables; (iv) StealthDB (sdx) and unit-test enclave (utx) expressed in logarithmic scale. Median overhead is around 3.9x and is depicted as a dashed line.

**7.2 RQ2 - Usage Evaluation**

We describe the use cases, 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 libdvccs [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.

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

**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 4. 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). A major source of overhead is incurred by the hash functions in the secure communication protocol (§4.2), 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 final user experience (§7.2.2). Additionally, we report a measurement of action-per-second in Appendix E to show the overhead is generally constant among the secure functions.

**7.2.2 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) libdvccs [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 replicate the same experiments described in the original paper [82]: we deploy StealthDB over a PostgreSQL [56] version 10.15 and run the benchmark OLTP [25] using same scale factors. Figure 5a and Figure 5b show the requests per second and the latency. 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.

libdvccs. We measure the CPU impact of SgxMonitor over libdvccs, which is a DRM library used in VLC media player [61]. We use a VLC version 3.0.8, on which we deployed three versions of libdvccs [80]: vanilla, with SGX, and with SgxMonitor. We play 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 do not experience any delay or interruption while playing the DVD in any of the three configurations.
We play the game for around 20 minutes and sample the CPU usage every second. Figure 6 shows the result of our experiment. Similarly to libdvdcss, we observe a first adjusting phase followed by a plateau at around 4%. Furthermore, we do not experience any delay or interruption while playing SGX-Biniax2 in any of the two configurations.

**Final Notes.** 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).

**7.2.3 Model Extractor.** 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, 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. In our experiments, we set the symbolic execution timeout to 10 minutes.

Table 2 shows our coverage results. We apply the analysis described in §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.

In all our experiments, we do not encounter any false positive from any of the micro- and macro-benchmark, we provide a thorough discuss of the precision of our model in Appendix B.

**Final Notes.** 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 analogies with other research areas: provenance analysis works, SGX memory-corruptions, and remote inspection.

Provenance Analysis. Many provenance tools are based on instrumentation to collect 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 a 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 may tamper with these operations. Overall, SgxMonitor is the first provenance analysis suitable for the SGX environment. To achieve this, we design a novel log collection and propose a novel model to represent the normal behavior of an enclave.

SGX and Memory Corruption Errors. CFIs and shadow stacks [26, 28, 39, 44, 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.

SGX Remote Inspection. 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. SMILE [91] is a novel request-based introspection mechanism that allows a remote agent to securely dump enclave memory regions. This tool can be used for forensic analysis in SGX enclaves. However, request-based approaches need to be manually activated thus leaving time to an intrusion to clear any evidence. Conversely, SgxMonitor continuously dumps runtime information, thus blocking evasion movements (§7.1.2).

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 while not undermining the SGX isolation.

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’s 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.

ACKNOWLEDGMENTS

We thank our shepherd and the anonymous reviewers for their careful feedback which greatly improved the clarity of this paper. This project has received funding from the European Research Council (ERC) under the H2020 grant 850868, SNSF PCEGP2_186974, and Fondation Botnar. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

In this section, we discuss the application of SgxMonitor model ($S$) over two important Intel SGX SDK mechanisms: the outside function interaction ($\mathcal{A}_1$) and the exception handling ($\mathcal{A}_2$).

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

$$T = PU[x].$$
T is composed of any valid sequence of generic actions P (according to the specifications in §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 how we model the 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 interact with the host, thus it performs an ERET; or (E2) it interacts 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 ocallContext (i.e., T OCALL1), and (ii) invoking the outside function (i.e., T OCALL2). Once the outside function resumes the secure function execution, it invokes an ORET in two steps: (i) the execution enters in the enclave (i.e., TORET1), and (ii) the ocallContext is restored (i.e., TORET2). 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. During (TH), SGX interrupts its execution as a result of an AEX and transfers the control to the host. Upon an AEX, 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 trtsHandleException, which index is −3 (i.e., T THD1). This function fills an sgxExceptionInfo_t structure with the values previously stored in the SSA (i.e., TTHD2). At the end of (TH), the enclave is ready for the second phase (IH) and transfers the control to the host (i.e., THD3). Then, the host invokes ERESUME to activate the internalHandleException routine (i.e., T ERESUME) and the enclave iterates among a set of custom handlers (i.e., THD1 and TTHD2). Each custom handler attempts at fixing the exception by analyzing the sgxExceptionInfo_t, possibly altering it. Therefore, we update the enclave internal state at each iteration. After invoking the internal handlers, the SGX SDK resumes the secure function through the continueExecution routine (i.e., TCNT). Finally, if the exception is properly handled, the secure function will continue, otherwise, a new AEX will raise a new exception.

B USE CASE ANALYSIS

We discuss the use case complexity, the trade-off between symex/static analysis, and its 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 (§5.5). The concept of simple enclave has already appeared in previous works [21, 73], however, they did not provide comparable metrics. In Table 3, 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. We choose these metrics inspired by previous works in similar contexts [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 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 does not eradicate the problem in SGX enclaves, even for simple ones.

Symbolic and Static Analysis Trade-Offs. We investigate the trade-offs between symbolic execution and the insensitive static analysis SgxMonitor relies on. Specifically, we explore the actions in two conditions: (i) using only the symbolic execution (symex), and (ii) using only the insensitive static analysis (static). We consider only those functions for which symex completes (i.e., no timeout). Moreover, we consider symex as ground truth since we rely on its soundness for a correct function exploration. Then, we compute the Δ = 100 × (#actions static − #actions symex)/#actions symex, and distinguish three cases. If Δ = 0, symex and static identify the same actions, nodes, and edges in the CFG. If Δ > 0, then static over approximates the actual CFG, creating rogue nodes and edges that may lead to false negatives (i.e., attacks the model misses). If Δ < 0, then static under approximates the actual CFG, thus missing nodes and actions that may lead to false positives (i.e., valid enclave events that are flagged as an attack) at runtime.

Table 2 shows the trade-off between symex and static. For almost all the use case, besides StealthDB, Δ is positive, from 4.41% to 21.02%. Therefore, relying on static only would have increased that attack surface up to 21.02%, potentially leading to false negatives. For StealthDB, instead, Δ is negative (−4.16%), meaning that static would have reduced the model accuracy, leading to false positives. This is caused by an indirect jmp that the insensitive static analysis did not resolve, thus reducing the actions explored.

Observing a non-zero Δ confirms our hypothesis on the suitability of symbolic execution for modeling SGX enclaves with high actions and code coverage. We fall back to a lightweight insensitive static analysis only to model a small percentage of actions/code the symbolic execution does not, thus limiting the overall risk of false negatives and positives caused by the analysis’ imprecision. This is analogous to reducing the TCB size to minimize the attack surface.

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 (set at 10min.); static, by using only insensitive static analysis; and symex+static, which is the one described in §5.5. Using only symex models, two secure functions in Contact generate false positives due to the function creci p that was not explored completely. We observe similar cases in SGX-Binax2 and libdvdcss, in which critical functions for crypting/decrypting were not correctly explored with only symex. We register false positives also...
Table 3: 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.

<table>
<thead>
<tr>
<th>Use case</th>
<th>LoC</th>
<th># secure function</th>
<th>cycl. compl.x.</th>
<th># nodes in CFG</th>
<th># edges in CFG</th>
<th># direct calls</th>
<th># indirect calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contact [7]</td>
<td>4138</td>
<td>6</td>
<td>5.03 5.04</td>
<td>24.89</td>
<td>22.74</td>
<td>26.67</td>
<td>27.82</td>
</tr>
<tr>
<td>libdvdcss [80]</td>
<td>3438</td>
<td>4</td>
<td>6.55 6.07</td>
<td>38.71</td>
<td>31.28</td>
<td>39.67</td>
<td>37.95</td>
</tr>
<tr>
<td>StealthDB [82]</td>
<td>10351</td>
<td>3</td>
<td>6.35 4.72</td>
<td>36.14</td>
<td>23.38</td>
<td>40.40</td>
<td>27.51</td>
</tr>
<tr>
<td>SGX-Biniax2 [12]</td>
<td>4696</td>
<td>7</td>
<td>3.73 4.20</td>
<td>18.56</td>
<td>16.25</td>
<td>20.19</td>
<td>20.02</td>
</tr>
<tr>
<td>unit-test</td>
<td>583</td>
<td>3</td>
<td>4.06 5.25</td>
<td>18.44</td>
<td>17.53</td>
<td>18.75</td>
<td>21.95</td>
</tr>
</tbody>
</table>

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 §7.2.2). 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 Appendix C). Therefore, our Model Extractor automatically prunes those paths that never appear...
1 DECLARE_LOCAL_FUNC do_ocall
2 ; . . .
3 ; ocall_context_generation
4 ; . . .
5 ; call trace_context_generation
6 mov %rsp, %rdi
7 call trace_context_generation
8 ; call update_ocall_lastsp
9 mov %rsp, %rdi
10 call update_ocall_lastsp
11 ; . . .
12 ; enclave stack setup
13 ; . . .
14 ; call trace_eexit
15 call trace_eexit
16 ; . . .
17 ; restore outside stack context
18 ; . . .
19 ENCLU

Figure 9: Example of stop actions in do_ocall(), trace_context_generation() and trace_eexit() report the generation of a new ocall_context and the enclave exit (i.e., ENCLU), respectively.

at runtime, i.e., if the enclave does not contain custom handlers, it will never execute part of internal_handle_exception.

C 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 (§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.

1 if (sgx_register_exception_handler
    (1, divide_by_zero_handler) == NULL) {
2     printf("register failed\n");
3 } else {
4     printf("register success\n");
5 }

D STOP ACTIONS

Figure 9 shows two examples of stop actions that trace the generation of ocall_context (i.e., trace_context_generation()) and the enclave exit (i.e., trace_eexit()) inside do_ocall(). Most importantly, there are no attacker-controlled transfers between the two tracing functions, therefore, the monitor can security trace the state change and the enclave exit.

E ACTIONS PER SECOND

Figure 10 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.

The results in Figure 10 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. Overall, this experiment show Sgx-Monitor overhead is generally constant among the secure functions.