Towards Formal Verification of State Continuity for Enclave Programs

Mohit Kumar Jangid
The Ohio State University
jangid.6@osu.edu

Guoxing Chen
Shanghai Jiao Tong University
guoxingchen@sjtu.edu.cn

Yinqian Zhang[*] ✉
Southern University of Science and Technology
yinqianz@acm.org

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Abstract

Trusted Execution Environments such as Intel SGX provide software applications with hardware support for preventing attacks from privileged software. However, these applications are still subject to rollback or replay attacks due to their lack of state continuity protection from the hardware. Therefore, maintaining state continuity has become a burden of software developers, which is not only challenging to implement but also difficult to validate. In this paper, we make the first attempt towards formally verifying the property of state continuity for SGX enclave programs by leveraging the symbolic verification tool, Tamarin Prover, to model SGX-specific program semantics and operations, and verify the property of state continuity with respect to monotonic counters, global variables, and sealed data, respectively. We apply this method to analyze these three types of state continuity issues exhibited in three open-source SGX applications. We show that our method can successfully identify the flaws that lead to failures of maintaining state continuity, and formally verify the corrected implementation with respect to the desired property. The discovered flaws have been reported to the developers and some have been addressed.

1 Introduction

The demand for confidential computing has driven the recent rapid development of trusted execution environments (TEE), such as Intel Software Guard Extension (SGX) and AMD Secure Encrypted Virtualization (SEV), in mainstream processors. These hardware-assisted TEEs allow the applications to compute directly on confidential data without leaking secrets to powerful adversaries who control the computing infrastructures (*e.g.*, operating systems). Introduced in 2013 [7, 30, 39] and officially released in 2015, Intel SGX becomes a leading TEE product that gains significant attractions from both academia and industry in developing various novel systems (e.g., [8, 13, 32, 50, 52]) and applications (e.g., [43, 45, 51, 56]).

However, software built with TEE support is not secure by default. Building a secure SGX application comes with many challenges, one of which is the lack of *state continuity* protection in SGX. State continuity is a well-known research problem in the literature of trusted computing (e.g., [6, 16, 22, 38, 41, 49, 50, 57]). It states that when a protected module resumes execution from an interruption (*e.g.*, reboots or system crashes), it should resume from the same state before the interruption [41].

Unfortunately, the issue of state continuity becomes even more complex in the context of Intel SGX. An SGX application is divided into untrusted and trusted components; the trusted components running inside the protected memory regions (dubbed *enclaves*) form the trusted computing base (TCB) of the application. Because the trusted components cannot directly access system services, such as file systems, network I/O, and memory management, the execution of the TCB is interleaved with frequent requests to the untrusted part for such services. The support of enclave multi-threading further complicates the execution states of the TCB, which allows concurrent updates of the TCB states.

The SGX hardware cannot ensure state continuity of the enclave programs for two reasons. First, the execution state can be distorted by data input from the untrusted component. Even when such data is encrypted and integrity protected, *e.g.*, monotonic counters, sealed storage, and authenticated messages, a previously used data can be replayed to the enclave program—bypassing decryption and integrity checks—and effectively rolling the TCB state back to a previous one. Second, the execution state can be affected by global enclave variables altered by concurrently executed enclave threads. As thread scheduling can be manipulated by the adversary, thread-unsafe enclave code is particularly susceptible to data races [53]. As such, improperly implemented enclave programs may find itself vulnerable to attacks due to its lack of state continuity protection.

[*]The bulk of the research was done while the author was a faculty member at The Ohio State University.

Ensuring state continuity in an enclave program is not easy. To do so, the developer must clearly understand the boundary between trusted and untrusted components, carefully use the SGX SDK to implement synchronization locks and the remote/local attestation logic, and properly implement accesses to monotonic counters, secure clocks, sealed storage, and various cryptographic primitives. This is unquestionably tedious and error-prone. Validating the correctness of enclave implementation, nevertheless, in a programtic and automated manner is a research problem yet to be solved.

In this paper, we make the first step towards formal verification of state continuity for enclave programs. Specifically, we resort to symbolic verification, which has been shown with significant success in proving protocol security [10]. Symbolic verification tools, such as Tamarin [40] and ProVerif [15], typically come with built-in support for standard cryptographic primitives, Dolev-Yao Model adversary capabilities, and desired properties specified in first order logic. These tools have been used to analyze TLS 1.3 [14, 21, 26], the Noise framework [29, 36], the secure messaging protocol (*e.g.*, Signal) [35], 5G authentication key exchange [12], and so on. However, symbolic verification has never been applied to verify state continuity for SGX enclave programs. Modeling state continuity involves interaction among the CPU hardware, the operating systems, and the enclave software, which is intuitively more complex than modeling message passing between network entities. Therefore, prior to this work, whether symbolic verification can be applied to this problem remains unclear.

Our key observation is that the operations of enclave programs can be approximated by the execution logic of Tamarin and the Dolev-Yao model [27]. Because the memory of enclaves is encrypted, the untrusted software cannot direct inspect the internal states of the enclave program; however the untrusted software may act as a man-in-the-middle adversary, who is capable of eavesdropping, reordering, blocking, delaying, replaying, modifying, or even generating messages between trusted entities (*e.g.*, enclaves, remote users), by manipulating the instantiation, data inputs, and execution ordering of enclave threads. Moreover, the property of state continuity can be modeled as the problem of uniqueness of objects and events, one-to-one mapping of requests and responses, and specific ordering of events. As such, first-order logic commonly used in symbolic verification tools should be sufficient for reasoning state continuity.

Therefore, we propose to automate formal verification of state continuity for SGX programs using Tamarin prover [40], a well-known symbolic verification tool. Specifically, we use Tamarin to model the execution of SGX programs, including enclave APIs, isolated memory, monotonic counters, SGX derived keys, *etc*., and then verify their state continuity properties. Tamarin is chosen over other similar tools, such as ProVerif, for several reasons. First, Tamarin supports the abstraction of mutable global states and adopts a more generic and low-level encoding language [10]—Multiset Rewriting rules—than ProVerif. This capability allows us to model the execution of SGX applications in sufficient details. Second, whereas ProVerif uses approximation to make the prover automatic and efficient, Tamarin's prover engine does not make any approximation over the model developed by its users. Therefore, the use of Tamarin gives us fine-grained control of the model and the execution of the prover.

We apply our method on three categories of flaws that allow violation of state continuity; in these three categories, the TCB states are stored in monotonic counters, global variables, and data in the sealed storage, respectively. We have discovered such problems in many open source SGX applications and selected one application from each category, namely, Hyperledger Sawtooth [1], SGXEnabledAccess [19], and BI-SGX [42]. We developed Tamarin models[1] for the core part of each of these three applications.

While expertise of using the Taramin Prover is still required, templates for modeling individual SGX primitives and state continuity properties could significantly facilitate the construction of the Tamarin models. Experiments suggest that our method can successfully identify the state continuity vulnerabilities in these applications. We have empirically validated the identified flaws and found that they can indeed be exploited by the adversary to alter the integrity of the execution of the vulnerable enclave programs. We also show that Tamarin can provide proofs of the absence of such vulnerabilities after these flaws have been mitigated.

Contributions. The contributions of this paper are three-fold:

- It makes the first attempt towards using symbolic verification tools to verify the property of state continuity for SGX enclave programs in a semi-automated manner. To the best of our knowledge, there is no prior work on the automated detection or verification of logic flaws like state continuity.
- It presents new techniques of utilizing the Tamarin Prover to model SGX primitives and reason about the state continuity property with first-order logic. Prior to our work, use cases of Tamarin are limited to verification of cryptography protocols against Dolev-Yao adversaries. This work for the first time extends the application of Tamarin to verify program logic.
- It applies the new techniques on the verification of three open-source SGX applications. Our proposed method can successfully identify the state continuity flaws and verify the absence of such flaws in the modified versions. The discovered flaws have been reported to the developers of these applications and some have been addressed in later versions of these applications.

[1] Our Tamarin code is released at Github: https://github.com/OSUSecLab/SGX-Enclave-Formal-Verification.

2 Background

2.1 Intel Software Guard Extension

Intel Software Guard eXtensions (SGX) [30] is microarchitectural extensions introduced in recent Intel processors, aiming at providing shielded execution environment, dubbed *enclaves*, for applications. An application for Intel SGX is divided into trusted and untrusted components, with the trusted components protected by the enclaves. Each enclave could support multiple threads running concurrently; the thread metadata is managed in a particular data structure called *thread control structure* (TCS).

Enclave identities. When an enclave is created, the hash value of its initial code and data is calculated by hardware as the *enclave identity* (*i.e.*, MRENCLAVE). Additionally, each enclave will be signed by its developer—Independent Software Vendor (ISV) as coined by Intel—before release. The hash value of the public signature verification key is used as the enclave's *sealing identity* (*i.e.*, MRSIGNER).

Remote attestation (RA). Intel provides a remote attestation mechanism for the remote client to verify that the enclave code is running on a legitimate Intel CPU with proper microcode patches, and the enclave identity is the same as expected by the client. A successful RA will allow the client to trust the execution environment of the enclave program and then provision secrets into the enclave.

Sealing. Intel SGX provides a mechanism called *sealing* to enable enclaves to securely store sensitive data outside the protected memory. A private key called *sealing key* can be derived within the enclave to encrypt the sensitive data before storing it outside the enclave memory. The sealing key, like other SGX-specific secrets, can be configured accessible to all enclaves with the same MRENCLAVE or with the same MRSIGNER.

Ecalls and Ocalls. To facilitate the development of SGX applications, Intel provides an official SGX SDK [4], which provides standard interfaces (ecalls) for calling into the enclave code from the untrusted application and interfaces (ocalls) for the enclave code to call untrusted functions for system services. The SDK also provides standard cryptographic APIs and high-level functions for sealing and remote attestation.

Platform service enclave. Intel provides a privileged enclave, called Platform Service Enclave (PSE), to access Converged Security and Manageability Engine (CSME), a secure coprocessor on the same machine. PSE provides other enclaves an interface to access trusted monotonic counters and trusted clocks that are maintained in the CSME.

2.2 Tamarin Prover

Tamarin [40] is a software tool for symbolic modeling of cryptographic protocols and verification of desired security properties. In particular, it models agents of a security protocol and messages passed among them, a desired security property that the protocol aims to maintain, and a proactive or passive adversary. The foundation of the Tamarin prover is a multiset rewriting rules (MSR) for modeling a protocol, including a set of equational theories dictating cryptographic operations, and a first-order logic formula specifying the desired property. Tamarin offers automated or semi-automated construction of proofs by checking the satisfiability of the negated formula of the desired security property.

The input to a Tamarin tool comprises a model of a cryptographic protocol, in the form of a set of MSRs, and the desired property, which is represented in first-order logic. Each agent of the protocol is modeled by several MSRs, each of which abstracts one or multiple actions of the agent, *e.g.*, receiving requests, performing operations cryptographic operations, or producing responses. Tamarin then outputs reports of whether the property is satisfied in all possible executions of the model. If so, a proof is provided; otherwise, Tamarin produces a counter-example execution of the model (which is visually presented in a graph). Since proving a property for a given model is undecidable, Tamarin does not always terminate.

2.2.1 Terms and Functions

In Tamarin, cryptographic messages are modeled as terms, which are categorized into *fresh terms* and *public terms*. The former are used to model nonces and private keys, and the latter are used to model publicly known values.

Cryptographic primitives are modeled as functions. A function symbol $f : t_1 \times \cdots \times t_n \leftarrow t$ takes n terms as inputs and outputs a term representing the return value. For example, a symmetric encryption scheme can be modeled as two functions: $enc(m,k)$ takes a message m and a key k as inputs and outputs a ciphertext, and $dec(c,k)$ takes a ciphertext c and a symmetric key k as inputs and outputs a plaintext.

Properties of functions are modeled as equational theories. For example, the equational theory $dec(enc(m,k),k) = m$ indicates that decryption of a ciphertext using the same key as the encryption returns the original plaintext. Tamarin provides a set of built-in functions and equational theories to model standard cryptographic operations (*e.g.*, symmetric and asymmetric encryption, cryptographic hash, digital signature, bilinear pairing, and multiplication and exponentiation in Diffie-Hellman key exchange) and a limited arithmetic operations (*e.g.*, multi-set union, XOR, and concatenation). User-defined equational theories can be used to provide additional operators, as long as the theory falls in the class of convergent equational theory with finite variance.

Tamarin provides built-in pairing and projection functions to model tuple terms. Particularly, the function $pair(x,y)$ models the pair of two terms x and y, and functions $fst(p)$ and $snd(p)$ models the projections of the first and second arguments with the following equations: $fst(pair(x,y)) = x$ and

$snd(pair(x,y)) = y$. A tuple term $< t_1, \ldots, t_n >$ is represented as $pair(t_1, pair(\ldots, pair(t_{n-1}, t_n)\ldots))$.

2.2.2 Facts

The security protocol to be verified is depicted as a sequence of interactions between agents. The state of an agent is represented as a set of *facts*. Each fact models the information the agent holds, *e.g.*, a private key. A fact is of the form of $F(t_1, t_2, \cdots, t_n)$, where F is the name of the fact, and t_i refers to a variable or a constant of the protocol. Note that from the fact, the adversary could not extract the variables t_i within. Hence, t_i can be private data. There are two types of facts: linear facts and persistent facts. Linear facts can be consumed only once by the agent during state transition (represented as MSR rules, which will be explained later), and thus they do not appear in all states of the transition system; in contrast, persistent facts persist during transitions. There are four special built-in facts: Fr, In, Out, and K. Fr is used to generate fresh random variable; In and out are used to receive and send data over public channel, respectively; K is used to directly add data to the adversary's knowledge base.

2.2.3 Multiset Rewriting Rules

The actions of an agent are modeled as multi-set rewriting rules, which dictate state transitions of the agent. Every rule consists of three components: the left-hand side component (a.k.a., *premise*), the middle component (a.k.a., *action*), and the right-hand side component (a.k.a., *conclusion*). Each of these components consists of a set of facts. Roughly, the premise serves as the input of the rule, the conclusion serves as the output, and the action are marked by action labels to log rule execution (a.k.a., instantiation). Each action label is tagged with variables that allow Tamarin to reason about the execution of the rule, in terms of relationship between the variables. In addition, *constraints* can be specified over the action labels to restrict the execution of the rule. An example of a Tamarin rewrite rule is shown as follows:

$$[F1(t1), F2(t2)] - [Eq(t1,t2), Act1(t2)] \rightarrow [Out(t2)]$$

where $F1()$ and $F2()$ are linear facts, $Eq()$ is a constraint, $Act1()$ is the label of the action, and $t1$, $t2$, and $t3$ are symbols. This rule specifies that if the agent has knowledge of the two facts $F1$ and $F2$ and the two related variables $t1$ and $t2$ are equal, the agent will send $t2$ to public channels.

2.2.4 Restrictions on State Transitions

A user of the Tamarin prover can explicitly exclude invalid execution traces in three ways: 1) restriction axioms, which are expressed in first-order logic. During the verification process, Tamarin considers only the model traces that satisfy the axiom; 2) type restriction prefix $\sim$. If a variable is prefixed with the $\sim$ symbol, the rule cannot execute repeatedly with the same value of the variable; 3) implicit *pattern matching*, which dictates that two variables of a rule with the same name should be instantiated with the same value.

2.2.5 Properties and Proofs

Tamarin's property is expressed as first-order propositional logic over the action labels. With the help of timepoint variables, the relative order of action labels can be encoded as well. To prove or disprove a property, Tamarin maintains a system state as it explores valid traces of the model. A trace is maintained using a graph data structure with rules as nodes and fact dependencies (fact production and consumption) as edges. The system state consists of session variables, messages in the network, and the current knowledge base of the adversary. For the target property, Tamarin's goal is to either find a trace that contradicts the property or show that all traces satisfy the property.

Tamarin's proof algorithm begins with an empty system state. It first derives the negation of the target property, and assumes its premise, *i.e.*, the part to the left of the implication sign $\Rightarrow$, to be true. Then it instantiates all MSRs that can be applied given this assumption. Starting from these rules, Tamarin tries to build an execution trace of the model using a backward search algorithm [48]. In this process, Tamarin derives a set of constraints from the dependencies among facts, the ordering of action labels, the adversary's knowledge base, the variable relationship as specified in the target property, and other components of the model, such as type restrictions, pattern matching, helper lemma, and so on.

Based upon various heuristics implemented in Tamarin, one of these constraints is picked from the system state and resolved by Tamarin's constraint-solver. The resolution step produces further constraints or eliminates some of the existing constraints. A constraint can be satisfied from multiple source rules, thus building up multiple proof sub-case branches, each representing a potentially valid trace of the model. Users can additionally influence the proof process by adding *Helper lemmas*, *i.e.*, lemmas with the *reuse* annotations, in the model. These lemmas are added to the system constraints in the proof process. Each helper lemma needs to be proven by Tamarin first before being used as a constraint.

2.2.6 Adversary Model

Tamarin follows the Dolev-Yao Model [27] to define the capabilities of an adversary, which includes eavesdropping, creating, modifying (including combining or splitting), and replaying messages in a public channel. Additionally, the adversary is armed with message deduction and construction rules, which allows her to apply cryptographic rules or model-specific knowledge to advance her current knowledge base.

2.2.7 Common Assumptions in Tamarin

As a symbolic verification tool, each Tamarin model inherits the following assumptions.

- The standard cryptographic primitives are perfect, *i.e.*, the only way to subvert decryption or forge signatures is to obtain the corresponding secret key. Hash operations are purely one-way operation and collision resistant.
- Each symbol or term is atomic that cannot be broken into multiple terms.
- Fresh variables (generated using fact `Fr(.)`) are pure random variables and each instantiation is guaranteed to produce a unique value.
- Multiple operations within one rule execute as one unit. Tamarin does not consider interleaving of such operations in the proof process.
- Each declared variable in a rule is local, *i.e.*, variable used in two different rules with the same name are different.
- Tamarin can argue about relative ordering of a rule execution, but it cannot measure the elapsed time between two executions of rules.

3 Overview

3.1 Problem Statement

In this paper, we aim to address the problem of state continuity in the context of Intel SGX enclave programs. The concept of state continuity was proposed in the context of *protected modules* [18,28]—code running in isolated environments with limited APIs to the outside—isolated by a combination of hardware and software components. It states that the protected module must resume from the same execution state after TCB interrupts due to reboot or crash [41].

However, the TCB of SGX enclaves is more complex than that of protected modules. As a user-space TEE, SGX maintains its software TCB in the enclave memory, monotonic counters, and sealed storage. This TCB can be updated by any code inside the same enclave, whose execution states can be initiated, interrupted, suspended, and terminated by privileged software at any time and in an arbitrary order. As the execution of the TCB depends on the input data from the untrusted software (in the form of ecall parameters and ocall return values), the execution state can be easily manipulated. Moreover, even when such input data is encrypted and integrity protected, *e.g.*, monotonic counters, sealed storage, and authenticated messages, a previously used data can be replayed to the enclave program—bypassing decryption and integrity checks—and effectively rolling the TCB state back to a previous one. Nevertheless, the support of multi-threading in the software TCB makes the protection of state continuity even more challenging, as the execution integrity of the TCB can be affected by the interleaved accesses to global variables.

As such, we consider a more general definition of *state continuity* in this work: Specifically, we define states of enclave programs as data stored in the enclave memory (*e.g.*, global variables) and non-volatile memory (*e.g.*, monotonic counters) and persistent storage (*e.g.*, sealed data); and state continuity is a property of the enclave program, which states that the enclave program always executes on the *expected* state, even when the execution can be restarted, suspended, and interrupted arbitrarily by the privileged software, or interleaved with another concurrent enclave thread sharing the same set of global variables. Clearly, the traditional definition of state continuity is subsumed by ours.

3.2 Attacker Model

Following SGX's threat model, we assume that the OS and other privileged software is controlled by the adversary. In particular, the adversary can create new processes and threads, instantiate enclaves from an enclave binary, trigger ecalls to an enclave with arbitrary arguments and in arbitrary order, pause the execution of an enclave at a specific instruction, hijack ocalls and return arbitrary values to ocalls. This includes triggering concurrent ecalls with multiple threads as long as multi-threading is supported by the enclave binary. However, other SGX attacks such as side-channel attacks (*e.g.*, [55]), denial-of-service attacks (*e.g.*, [34]), and speculative execution attacks (*e.g.*, [17]) are not considered.

3.3 Overview

In this paper, we aim to tackle the verification of state continuity using symbolic verification tools, which have been previously used to verify security of cryptographic protocols but never applied to reason about system security. However, doing so encounters two major challenges:

First, one must convert semantics of software programs, such as branches, global and local variables, synchronization locks, as well as a variety of SGX primitives such as monotonic counters, sealed storage, derived keys, relationship between developers and enclave code, and adversary capabilities into Tamarin's MSRs. As the first attempt to achieve these goals, this work proposes new ideas of building models using Tamarin MSRs for each of these primitives.

Second, one must encode the desired state continuity properties into first-order logic that can be expressed by Tamarin lemmas. This work explores the modeling of state continuity properties using (1) one-to-one mapping between requests and responses, and (2) uniqueness of variables, messages and sessions.

In this paper, we use three case studies to illustrate the use of this method in the formal verification of state continuity properties for SGX enclave programs. In these three cases, the states of the enclave programs are maintained in the monotonic counters, global variables, and seal data, respectively.

The root cause of the problems varies. For instance, the TCB state may be different at the time-of-check from that at the time-of-use; in other cases, the TCB state may be replaced with a stale one due to improper rollback attacks. We will showcase how each of these state continuity issues can be modeled and verified.

4 Tamarin Models for SGX Primitives

Designing symbolic model for the SGX primitives require unconventional approach, as the execution model of Tamarin MSRs differs significantly from enclave code. In this section, we discuss the techniques and principles of building Tamarin models of each of the considered SGX primitives.

4.1 Structure of SGX Applications

An SGX application consists of a host program (untrusted) and an enclave binary (trusted). Every SGX application is developed by an Independent Software Vendor (ISV), who signs its enclave code and then deploys the entire SGX application to an SGX-enabled machine. One such a machine may run multiple SGX applications from different ISVs. At runtime, each SGX application is instantiated into a process; the process that executes the enclave code is called an enclave process.

Before modeling the operations of SGX applications, we first systematically model these entities and their relationships. For clarity, we use the terms *ISV*, *platform*, *enclave-binary*, and *enclave-process*, to denote an ISV, an SGX-enabled machine, the ISV-signed enclave code, an instantiated process from the SGX application.

The relationship among entities may be modeled in a layered network structure, which we call an *association network*. A node in an association network represents an entity (*i.e.*, a platform, an ISV, an enclave-binary, or an enclave-process). Each entity is modeled by one specific fresh term called *identity*, which is generated using an Fr fact. One example of forming entity association is shown in Figure 1. The top layer of the networks are the ISV entities, the second layer is the enclave-binary entities, and the third is the enclave-process entities. We denote the entities of same type as one *role*. Therefore, the identities of these entities are generated with fresh *role terms*; *isv*, *e*, and *p* are role terms. As a result, each role term instantiates into an distinct entity. The structure of the network may vary depending on the program to be modeled. When it is not necessary to include certain entities of a network, omitting them from the model may be beneficial: (1) it makes the proof of the model more efficient, and (2) the resulting model becomes more general.

An association network can be modeled as a sequence of rules, fact properties (§2.2.2), and restrictions on state transitions (§2.2.4). Besides generating the identities, the rules collect a set of role terms, called *association*, to maintain

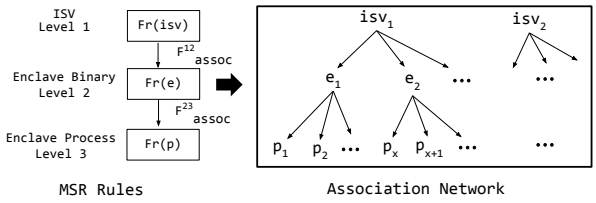

Figure 1: An example of a multi-layer association network. The network on the right suggest that one platform runs multiple ISV-deployed programs; each ISV may have multiple enclave-binaries; one enclave-binary is instantiated into multiple enclave processes.

the association information about the entities. *Association facts*, in the form of $F^{ij}_{assoc}(assoc^i)$, propagates the association information from rule at layer i to the rule at layer j. Also note that there can be multiple role terms at any layer. Each rule at the top layer has an *association* containing its role term. With the association fact $F^{ij}_{assoc}(assoc^i)$ passed from a parent rule i, the rule at layer j produces its association set $assoc^j = assoc^i \cup RT^j$, where RT^j is the set of role terms at layer j, and pass it to the rule at the next layer k using the association fact $F^{jk}_{assoc}(assoc^j)$.

As persistent facts, association facts can instantiate unbounded number of instances of role terms at the next layer. As shown in Figure 1, the top rule can be instantiated unbounded number of times producing unbounded instances of ISVs. For each of the ISV instances, the association fact can be passed down to the second rule to generate unbounded number of enclave-binary instances. As a result, the second rule observes a collection of enclave-binary instances under each ISV instance forming an unbounded networks of entity association. A similar procedure from second rule to the third rule manifests into unbounded 3-layer networks.

The association network structure is crucial to the modeling of enclave thread, scope of variables, and owner-policies, which will be detailed soon.

4.2 Enclave Threads

An enclave-process can be configured to support a single enclave-thread or multiple concurrent enclave-threads. We introduce a specific fact, *ecall fact*, in the form of $F_{ecall}(assoc^p)$ where $assoc^p$ is the association set of the enclave-process layer rule that initiates this thread. An ecall fact can be either linear (for single threaded execution) or persistent (for multiple threaded execution).

Each enclave-thread is modeled as another sequence of rules. The first rule of the sequence takes in an ecall fact to initiate the enclave-thread execution. Distinct facts, named as *thread facts*, of linear type are designated between each pair of consecutive thread rules. A thread fact from thread rule i to thread rule j is in the form of $F^{ij}_{thread}(assoc^p \cup \{t\}, state)$

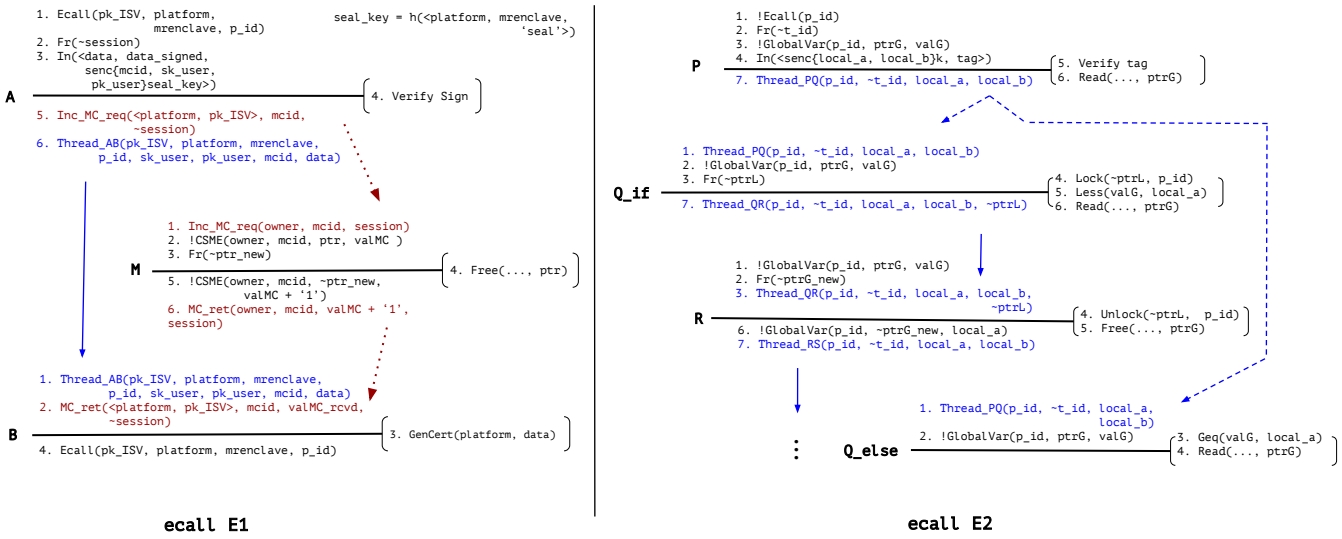

Figure 2: Example Tamarin code of two ecall E1 and E2. The dark and dashed arrow denote thread facts; the latter is used for branched rules. The dotted arrow represent communication between rules A and B of E1 to a monotonic counter rule M.

where t is the enclave-thread role term created using Fr fact at the very first enclave-thread rule, and $state$ records the state of the enclave-thread during the thread execution. We omit the enclave-thread role term for single threaded enclave for model efficiency and keep only $assoc^p$ as the association set.

As linear facts, thread facts enforce a single instance of each thread rule forming a sequence, whose order is defined by the order of the thread facts that are passed from one rule to another. Each thread fact between a consecutive pair of thread rules is assigned a unique name to enforce a sequential fact dependency (§2.2.5), resulting in a sequential execution of thread rules.

We use sample enclave ecalls `E1` and `E2`, as shown in Figure 2, throughout this section to illustrate many primitives. Particularly, `E1` is modeled as a 3-layer association network, with the top layer representing an ISV; the second layer representing user, enclave-binary and the platform; and the third layer representing the enclave-process. Similarly, `E2` is modeled as another only one layer association network with role term `p_id` for enclave-process.

The sequences of rules `A`→`B` and `P`→`Q_if`→`R`/`P`→`Q_else` model ecall `E1` (single-threaded) and `E2` (multiple-threaded), respectively. The rules `Q_if`/`Q_else` are used for branching (§4.8). As the very first rule of both threads, the ecall fact `ECall` at `A1` and at `P1`, provides necessary association information and thread data to start the thread. Further, the thread fact `Thread_AB` (at `A6`, `B1`) in ecall `E1`, and the thread facts `Thread_PQ` (at `P7`, `Q_if1`), `Thread_QR` (at `Q_if7`, `R3`, `Q_else1`), and `Thread_RS` (at `R7`), in ecall `E2`, carry the association information and thread data throughout the thread rules.

For an enclave-thread configured to run as a single-thread, it should also be allowed to start again once the single thread finishes its execution. To restart the thread, the ecall fact will be instantiated again in the end rule of the sequence. For example, in Figure 2, the ecall fact at `A1` in ecall `E1` is produced again in the end rule `B` to allow unbounded sequential threaded runs. On the other hand, multi-threading is supported by default when the ecall fact is persistent.

4.3 Scope of Variables

In SGX processes, roughly, each variable has one of the two types of scopes: local (exclusive to one thread) or global (shared between enclave threads). In this section we describe how we utilize the association network and enclave-thread construction to model the scope of variables.

One way to model a local variable is to keep it in the term $state$ of the *thread facts*. It is local because a linear *thread fact* can be instantiated only once and can be consumed only in the following thread rule instance. For example, *thread facts* `Thread_PQ`, `Thread_QR` and `Thread_RS` in the example ecall `E2` carry `local_a` and `local_b` as the local thread data.

A more generic way to model local and global variables is induced by *pattern matching* (§2.2.4) over association set of thread facts. Specifically, we model local and global variables in the forms of $F_{local}(assoc^p \cup \{t\}, var_l)$ and $F_{global}(assoc^p, var_p)$, respectively. Here $assoc^p$ is the association set of an enclave-process while var_l and var_p are the local and the global variables for an enclave-thread with role term t.

For illustration, consider two facts $F_1(assoc^p \cup \{t_i\}, var_{t_i})$ produced at a thread rule with a thread instance t_i, and $F_2(assoc^{p_i}, var_{p_i})$ produced at the enclave-process layer rule with enclave-process instance p_i. If all the facts received at the

thread rule are modeled to pattern match with the variables of the thread facts, the fact F_1 can only be consumed in the thread instances t_i but not in any other thread with instances, t_j, due to the violation of pattern match constraint, *i.e.*, the inequality of thread identity instances $t_i \neq t_j$. Thus, the facts F_1 and F_2 maintain local and global data, respectively.

When using global facts of linear type to model global variable, a rule modeling a *read* or *write* operation on the global variable requires the same global fact to be in both its premise and its conclusion. If it is not produced in the conclusion, the global fact will be consumed and removed from the system state, resulting in the loss of the global variable.

However, having a global fact in both the premise and conclusion is quite inefficient as it creates a cyclic fact dependency leading to increased verification time. A more efficient way of modeling global variables is through pointers. Pointer version of global facts is in the form of $F_{global}(assoc^p, ptr, var_p)$. A pointer version of the global variable partially avoids the dependency by requiring the global fact to appear only in the premise for a *read* operation. Particularly, pointer version global fact is modeled as a persistent fact is associated with a unique random value acting as a pointer ptr to each declared or updated value of the variable. Note that the persistent facts of the global variable with old values persist in the system state and can be read even after a persistent fact of the global variable with a new value is produced later. To handle this, the following two restriction axioms, 1 and 2, are introduced to preserve the consistent read-write behavior of the global variable.

Restriction 1:

```
All Read(owner, ptr, ...)@t1 & Free(owner, ptr, ...)@t2
==> #t1 < #t2
```

Restriction 2:

```
All Free(owner, ptr, ...)@t1 & Free(owner, ptr, ...)@t2
==> #t1 = #t2
```

The `owner` variable in `Read` and `Free` action-labels is introduced for access control, *i.e.*, which entities can access the global variable. The owner variable is declared with a tuple of identities as described in §4.5 and §4.6. Each `ptr` variable instance points to one update of the global variable. The restriction axiom 1 prohibits reading (action-label `Read`) of old values after an update (action-label `Free`) while the restriction axiom 2 ensures consistent updates of the same global value. In summary, the restriction axioms enforce that after the global variable fact is updated with a new pointer and value, the facts with old pointers and values can neither be read nor be updated.

4.4 SGX Keys Derivations

An enclave can use the `EGETKEY` instruction to derive secret keys from the hardware, including sealing key for encrypting sealed data, report keys for local attestation, and provisioning key for remote attestation. ISVs may choose to enable key sharing between enclave threads with the same `MRSIGNER` (an ISV) or the same `MRENCLAVE` (enclave-binary).

To model the accessibility of the derived keys, the related association set is used during the key derivation. Since the association set is accessible only to the associated entities (fact properties §2.2.2), it can also be used as secrets for deriving secret keys. Hence, entities that are allowed to have shared keys will have the same association set for deriving the same keys.

Derived keys shared between enclave threads with the same `MRSIGNER` on a platform can be modeled using the identities from the association set, *e.g.*, $\{platform, isv\}$. Consider three enclave thread instances, t_1, t_2, t_3, under the same `MRSIGNER`, *i.e.*, the same ISV instance isv_i and the same platform instance, $platform_i$. The descendant threads will inherit the same association set and thus can derive the same keys. Within these enclaves, the derivative values $h(< platform_i, isv_i, 'report' >)$ and $h(< platform_i, isv_i, 'seal' >)$ can be treated as shared report key and sealing key, respectively, among enclave threads t_1, t_2, t_3. Central to the confidentiality of the derived key is to keep at least one of the identities (`platform` in this case) secret in the derivation throughout the model.

The built-in hash operation $h(.)$ is pure collision and preimage resistant. These properties ensure that the derive keys are unique and cannot be interchanged across derived uses. The variable scope principle described in §4.3 ensures that the seal key with isv_i cannot be accessed by other enclaves with a different isv_j.

4.5 Monotonic Counters

Intel provides monotonic counters (MC) to enclaves (through PSE) to prevent rollback attacks. Once created, the values of the MCs will only get increased monotonically. An MC can be created, read, and incremented. Therefore, we model MC by creating one rule for each operation. The enclave-thread and MC communicate using a linear fact to establish a private channel. To ensure one-to-one mapping of the request and response MC counter we include a fresh variable *session* in the communication fact.

The MC memory is abstracted with a dedicated fact in the form of $!F_{CSME}(owner, mcid, ptr, counter\_val)$ where *owner* represents the owner policy defined for MC, *mcid* is the unique identity of the counter, the pointer *ptr* and the variable *counter_val* hold the reference and value of the counter, respectively. This fact is used only in the MC rules. In the MC creation request, enclave-thread use identities from association set to initiate $!F_{CSME}$ with desired owner policy. Across the three MC rules, the same owner binding also ensures that only one copy of CSME memory fact, $!F_{CSME}$, is used to hold consistent values of MC. The MC create rule returns a unique fresh *mcid* for enclave thread to keep and use later for *read* and *increment* requests.

We utilize the operator '+' (multiset union) over symbols from Tamarin's built-in multiset package to model addition operation over counters. The operator '+' along with restriction axiom logic, allows comparison (greater, less, equal) of any two symbols. The *increment* operations on counters can be modeled by rules that consumes a $!F_{CSME}$ with a counter value x in its premise and produces another counter fact with value x+'1' in its conclusion. The restriction action-labels `Read` and `Free`, as described in §4.3, are used to enforce counter to increase monotonically and maintain a consistent counter value for *read* and *increment* operation.

For example, the rule `M` of ecall `E1` (Figure 2) acts as *Increment* MC rule. The fact `CSME(owner, mcid, ptr, valMC)` (at `M2`, `M5`) models the CSME memory. The owner-policy term, `owner`, received in the private channel fact `Inc_MC_req` (at `M2`) as `<platform, pk_ISV>` (at `A5`) binds the CSME memory with the same signing key policy—one MC for all enclave-threads with the same signing identity (`pk_ISV`) for a given platform (`platform`).

4.6 Sealed Storage

With a sealing key, an enclave can store and retrieve the encrypted sealed data to and from untrusted storage via public channel modeled by `Out(.)` and `In(.)` facts, respectively. This allows the adversary to perform potential rollback attacks if applicable. In ecall `E1` of Figure 2, the received sealed data is encrypted with the sealing key derived from the platform secret and enclave-binary identities; this means the `MREENCLAVE` sealing owner policy is used. To use `MRSIGNER` sealing policy, the secret key can be derived with platform and ISV identities as described in §4.4.

4.7 Locks

Following Kremer and Künnemann [37], we model locks using restriction axioms. It introduces two action-labels, `Lock(pointer, association)` and `Unlock(pointer, association)` to the rules acquiring and releasing locks. The first variable in the action-label is a random pointer variable which establishes a unique pairing of the lock and unlock action-labels. The pointer is passed on with *thread facts* all the way through ecall sequences of rules from lock-acquire to lock-release actions. All instructions covered in these rules are locked per owner instance. The second variable associates the lock with entities that use the lock (*e.g.*, a single enclave-process layer lock among multiple threads).

The restriction axiom shown below utilizes time points, random pointer variables, and entity identities to enforce the correct lock behaviors. For the case when $\#t1 < \#t2$, the variable *ptr*_1, *ptr*_2 represent pointers and *owner* represent the owner entity identity. The constraint at line 3 prohibits overlapping of two different lock-unlock pairs. The constraint at line 4 prohibits creating two lock-unlock pairs with the same

SGX Threat model	Realized by
Thread and process instantiation	Using a thread policy based on the ecall facts (F_{ecall}) in the first enclave thread rule and binding ecall sequences of rules using *thread facts* (F_{thread}) (§4.2)
Permute or reorder ecalls	Modeling the first enclave thread rule open to executability without order dependencies of timepoints and facts
Pause enclave execution at instruction level	Modeling instructions in individual rules and utilizing atomic rule executability (§2.2.7)
Read access to ecall returns; Read-/Modify access to ecall or ocall arguments and returns	Arguments and returns pass through public channel
Replay, modify of sealing, ecall or arguments and returns	Public channel use in combination Tamarin's built-in Dolev Yao adversary capabilities

Table 1: SGX threat model construction

pointer. These two constraints effectively establish a unique pair instance $lock@t1$ and $unlock@t3$ with pointer instance n. Constraints at line 5-6 enforce any other lock-unlock pair, represented by pointer variable *ptr*_2, must occur either before or after the lock-unlock pair established at line 3-4. Line 7 covers the other possible order of two arbitrary lock instances in the premise. Particularly, the lock behavior enforced at lines 2-6 for $\#t1 < \#t2$ also applies to the order $\#t2 < \#t1$. Finally, line 8 completes the lock constraints by freely allowing a single lock instance to be applied anywhere in the model.

```
1. All Lock(ptr_1, owner)@t1 & Lock(ptr_2, owner)@t2
     ==>
2.   ( #t1<#t2
3.     & (Ex Unlock(ptr_1, owner)@t3 &#t1<#t3 & #t3<#t2
4.       & (All Unlock(ptr_1, owner)@t ==> #t=#t3)
5.       & (All Lock(ptr_2, owner)@t ==>
                                #t<#t1 | #t=#t1 | #t3<#t)
6.       & (All Unlock(ptr_2, owner)@t ==>
                              #t<#t1 | #t3<#t | #t3=#t)
         )
   )
7. | #t2<#t1
8. | #t1=#t2
```

In ecall `E2`, the restriction axiom action-labels `Lock(~ptrL, p_id)` and `Unlock(~ptrL, p_id)` enforce a per process lock (with `p_id` representing the enclave-process identity), which locks all the operations of `Q_if` and `R` rules.

4.8 Common Programming Primitives

Some common programming data structure—an *append only* indexable-database and control structures—loops and branching are expressible in MSR encoding.

Indexable-database. Consider a rule as shown below where the database and the counter facts are initialized as $!DB(owner,'0',nil)$, $!Counter(owner,ptr,'1')$ in a separate rule with ptr as a fresh term; $owner$ to encode owner policy the database fact.

$$[In(data), Fr(ptr_{new}), !Counter(owner, ptr, i), !DB(owner, i, x)] - [Free(owner, ptr)] \rightarrow [!Counter(ptr_{new}, i+1), !DB(owner, i, data)]$$

The addition operation is abstracted with '+' multiset union operator. The action-label $Free(owner, ptr)$ restricts the counter value to increase monotonically using the restriction axiom as described in §4.3. An abounded instantiation of the rule will introduce the unbounded copy of database fact in the system state with *data* received from other sources (public channel in this case) and unique counter values as indexes. Any receiving rule can utilize *pattern matching* to ensure that the received database fact's index matches with the requested index received from other sources. This approach models a *readable* and *append-only* key-value database, as the persistent fact cannot be deleted from Tamarin system once introduced.

Loops. In the rule described for the indexable database, the fact *Counter* creates a loop where each instantiation of the rule represents one iteration of the loop. Therefore, a loop is modeled with a persistent counter fact which consumes a value i in its premise and produces the the same fact with value i+'1' in the conclusion. Restriction over the action-label $Free(owner, ptr)$ controls the monotonicity of the loop. However, this approach models an infinite loop. In order to limit the loop to maximum n iterations, an action-label $Log(i)$ can be added to the rule's action with restriction axiom $\forall\, Log(i) \Rightarrow not(\exists\, i = n + z)$ where n represents '+' operator applied over symbol '1' n times. The axiom enforces that after reaching a counter value of n, further addition for any value of z is not allowed.

Branching. Consider an enclave-process modeled as a sequence of three rules `r1`→`r2`→`r3`. To introduce a branch in place of `r2`, replace the rule `r2` with `r2_if` and `r2_else` rules with the if and $else$ conditions enforced with action-label controlled by restriction axioms in the rule's action part. Identities are passed on via a linear thread facts as described in §4.3 in both rules. Since only one of the $if-else$ conditions will be true, only one of the `r1`→`r2_if`→`r3` or `r1`→`r2_else` sequence will be realized at a time.

For example, in ecall `E2` of Figure 2, `Q_if` and `Q_else` rules models a $if-else$ statement. The restriction axiom action-label `Less(valG, local_a)` (at `Q_if5`) enforces `valG < local_a` and `Geq(x,y)` (at `Q_else3`) enforces `valG >= local_a`. For each instance of an enclave-thread, only one of the condition holds true based on the enclave-thread specific instances of local variable `local_a` and the global variable `valG`.

5 Case Studies

In this section, we present three case studies to showcase our approach towards automated verification of state continuity.

5.1 State Continuity w/ Monotonic Counters

Hyperledger Sawtooth [1] is a permissioned blockchain framework to build customized decentralized applications. Sawtooth supports multiple consensus protocols, including a Proof-of-Elapsed-Time (PoET) that leverages Intel SGX to ensure each node's fair participation in the consensus protocol. PoET protocol works in two phases: the sign-up phase and the election phase.

To join the distributed network, a node launches an enclave which generates a pair of asymmetric keys and sends the public key certificate (together with a linkable attestation signature) to the network. Thus the identity of an enclave (and the node) can be uniquely identified by the certificate. Additionally, a trusted MC will be created to enforce unique PoET certificate generation per node. After this sign-up phase, the node qualifies for the node election phase.

The election phase of Sawtooth V1.0.5 [3] is described in Figure 3. Two ecalls are implemented to allow the node to participate in the block leader election. The first ecall `CreateWaitTimer` (CWT) performs three major steps. First, it records the current time as the reference start time using trusted time API `time_ref` and generates a random time duration as `wait_duration` that the nodes must wait. Second, it increments the associated MC and records the counter value in `MC_ref`. Third, it encapsulates `time_ref`, `MC_ref`, and `wait_duration` in an object `waitTimerObject`, which is signed with the private key of the node and transferred out to the application, so that the node can wait outside the enclave for the wait duration before invoking the second ecall.

The second ecall `CreateWaitCertificate` (CWC) performs several checks to ensure the fairness of the protocol: First, it unseals the approved sign-up data created during the sign-up phase. Second, `waitTimerObject` is verified with PoET node's public key to ensure the integrity of encapsulated variables. Third, the latest MC value is read and compared against the reference value. Fourth, by reading the current time, it calculates whether the elapsed time is greater than the expected wait duration. Only after all the checks pass does the enclave generate a PoET certificate to establish the proof of the elapsed wait time. Before returning, the MC is incremented in order to prevent another certificate generation without any wait time. Once the certificate is broadcasted into the peer-to-peer network, the node with the certificate of the smallest wait duration wins the round and is allowed to publish a block in the ledger.

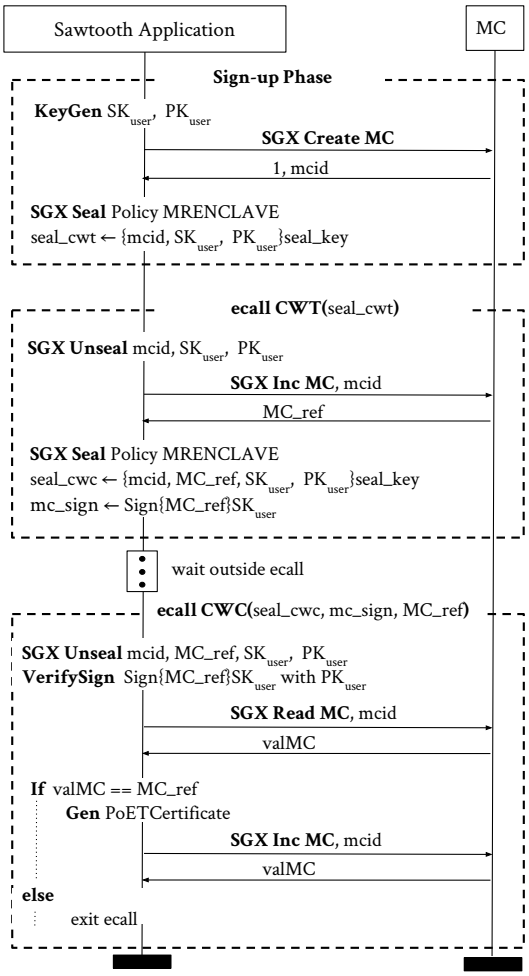

Figure 3: Protocol workflow of Sawtooth PoET.

5.1.1 Tamarin Model

We model one ISV and multiple nodes in the blockchain network. The association network has three layers: *ISV*, *Platform/MRENCLAVE/Users*, and *enclave-process*. The ecall CWT is modeled as a sequence of two rules and CWC is modeled as three rules. MC is modeled in the same way as described in §4.5. MC counters are associated with the same `MRENCLAVE` owner policy. The ecall CWT receives sign-up information, and returns the reference MC value and its signature, and the sealed sign-up data, which will be used by the ecall `CWC`. CWC performs checks of the input data, and generates a certificate if all checks pass. The wait operation is abstracted away.

Rules representing critical events are designated with specific action-labels. For example, the CWC rule for generating the certificate generation event is marked with the action-label `PoETCertificate_valMC(platform, MC_ref)`, where `platform` represents the node's identity and `MC_ref` is the reference MC value obtained from the MC.

In order to aid the termination of the proof, we also included two helper lemmas (see §2.2.5). The first lemma states that each MC read or increment rule instance must have a corresponding antecedent MC create rule instance. The second lemma ensures that the MC must increase monotonically.

5.1.2 Security Property

The security property studied here is to ensure fairness of the protocol. Specifically, for each CWT ecall, only one CWC is allowed to generate a certificate after the duration has passed. This is enforced by the increasing MC values. The applications state transits in the following sequence:

1. MC value is less than the reference value ⇒ the certificate is not generated yet;
2. MC value equals the reference value ⇒ generate the certificate and increment the MC value;
3. MC value is greater than the reference value ⇒ Abort (certificate as already been generated).

The property, as shown below, states that a node cannot generate two certificates with the same `MC_ref`.

```
All PoETCertificate_ex(platform, MC_ref) @t1
  & PoETCertificate_ex(platform, MC_ref) @t2
==> #t1 =#t2
```

5.1.3 Analysis Results

For the vulnerable version V1.0.5, Tamarin shows that the security property does not hold. In the proof graph, process identity helps in tracking different enclave-processes. The attack is shown by instantiating two parallel enclave-processes, with shared MC, which can read the same reference MC value using read API before certificate generation. The detailed attack graph, produced by Tamarin, is shown in Appendix B for readers of interests. The vulnerability exists due to using a non-incremental API, `sgx_read_monotonic_counter`, to gauge the certificate generation state, especially one where an adversary can repeat this state by exploiting multiple enclave-processes. We have confirmed the attack validity in the Sawtooth SGX code.

The vulnerability is fixed in the latest version of Sawtooth [2] by revising the implementation of the ecall `CWC`. Specifically, the call to `sgx_increment_monotonic_counter` was moved to the beginning of ecall. This prevents the second concurrent ecall from generating the certificate without increasing the counter. We accommodate the change into the Tamarin safe model by replacing *Read* MC API with *Increment* API and omitting the *Increment* MC API after successful certification generation. After this change, the desired property is proved. That is, only one certificate with unique reference MC value can be generated per node per election round.

5.2 State Continuity with Global Variables

SGXEnabledAccess [19] is a secure remote monitoring framework for IoT devices. Due to limited computing power and resources of the IoT devices (*e.g.*, Samsung SmartHome), the collected IoT data is often sent to a remote cloud server for further processing. One application of such a framework is remote patient monitoring. Personal vitals of a patient are collected by several IoT devices' sensors and aggregated by a trusted broker (TB) gateway on the user side; the data is sent to a cloud server for analysis and processing by health care providers (HCP). TB maintains a user-defined policy specifying which HCP services can access the patient data and provisions secret keys and the encrypted data to the HCP cloud application accordingly. SGX is leveraged on the HCP side to protect user data from unauthorized access.

To allow the user to manage the access control to the her uploaded data, a heartbeat protocol is introduced between the TB and HCP enclave. After establishing a secure RA session with the HCP application, the TB program periodically sends encrypted heartbeat signals to the HCP cloud. Each signal consists of two parameters: 1) an activeness flag (*i.e.*, *is_revoked*) indicating whether the uploaded data can be accessed and 2) a monotonically increasing counter for indexing the heartbeat signals. As long as the user allows her uploaded data to be accessed within the HCP service, the heartbeat signal is sent with an *active* state. Once a user decides to revoke access to her uploaded data, the last heartbeat signal is sent with an *inactive* state. On the HCP side, heartbeat signals are processed within an SGX enclave through an ecall `ecall_heartbeat_process`. The enclave decrypts the message (with the key derived from the remote attestation) and retrieves the counter value and the activeness flag. The enclave maintains two global variables to track the latest counter and to ensure the maximum allowed duration between heartbeat signals. These two global variables serve to prevent replays of the heartbeat signals and packet delays.

5.2.1 Tamarin Model

We model multiple users (represented by TBs) communicating with the HCP application. Since the ISV and platform entities were not required in enclave operations, the association network consists of only one layer—the enclave-process.

As shown in Figure 4, we modeled the ecall *heartbeat_process* as a sequence of four rules covering steps 1-2, 3, 4, 5, respectively. Additional two rules are introduced to cover branching at step 3 and 5 of the enclave-thread instructions. The thread decrypts the received heartbeat signal, performs various checks, and updates the global variable accordingly. The events of global variable update are recorded by designating a specific action-label `E_update(p_id, t_id, k, ptrG, valG, ptrG_new, valG_new, is_revoked_rcvd)` to the ecall rule, which updates the global variable. Here `p_id` and `t_id` are the process identity

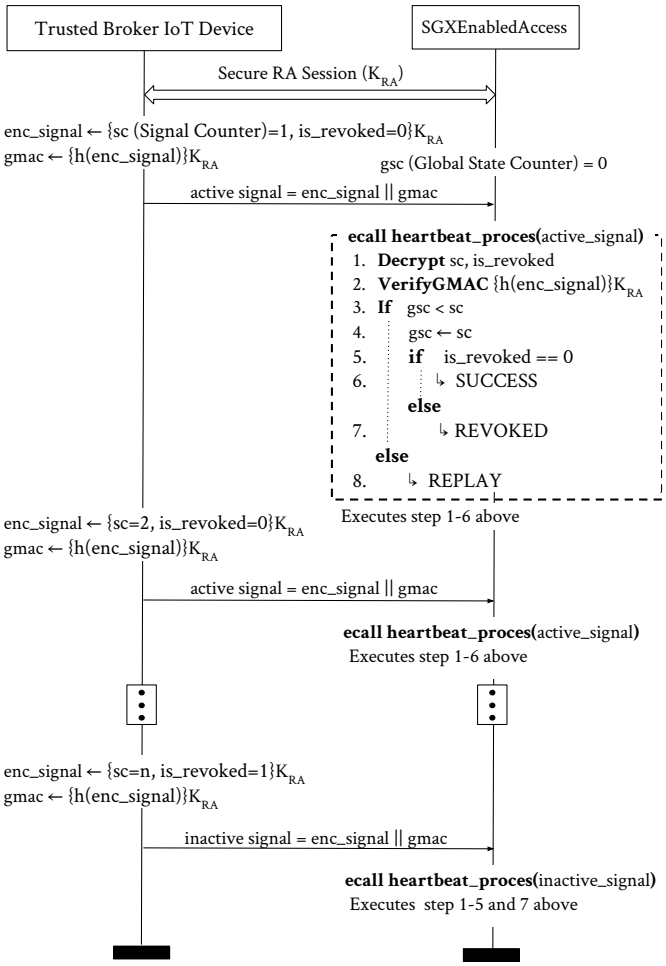

Figure 4: Interaction workflow of one TB device with ecall *heartbeat_process*

and thread identity; `k` is the RA key; `ptrG` and `valG` are the pointer and value of the counter values before the update while `ptrG_new` and `valG_new` are the updated pointer and value; and `is_revoked_rcvd` denotes the status of the accessibility.

It is enough to consider two distinct inputs available to adversary to model replay attacks. Therefore, we modeled two active signals followed by one inactive signals. The global variable is shared among multiple enclave-threads. RA procedure is abstracted with TB and HCP enclave thread starting with a pre-knowledge of RA session key; the communication channel is modeled with fact; GMAC tag is model as $h(enc\_signal)K_{RA}$.

To resolve the non-termination issues, we introduced five helper lemmas to ensure that (1) the RA session keys are never leaked to the adversary, (2) the thread rules of the same ecall strictly follow the specified execution order, (3) each rule instance for reading or writing global variables must have an antecedent rule instance for creating the same global variable,

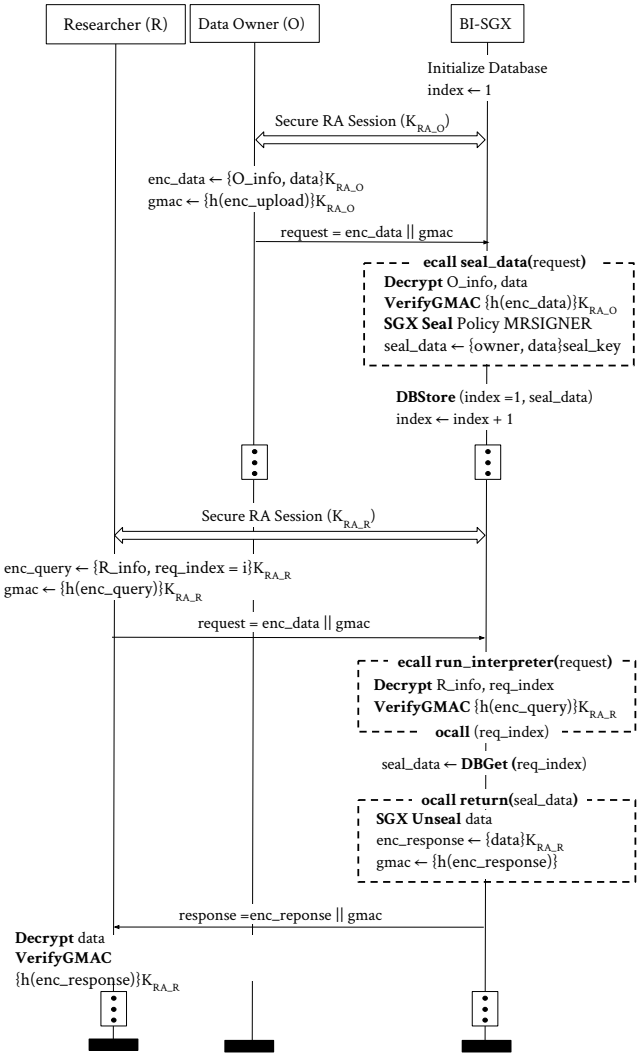

Figure 5: Protocol workflow of BI-SGX.

and (4) & (5) global variables cannot be read or written in parallel by concurrent threads in a critical section.

5.2.2 Security Property

In this system, the heartbeat signals sent by the TB contain monotonically increasing counter values as index, with larger index indicating more recent status of the accessibility of the uploaded data. Hence, it is required that the HCP enclave keeps track of the most recent status with the received signals. That is, the HCP enclave updates the global variable only when receiving a heartbeat signal with a larger counter value. This security property is shown below.

```
All E_update(process_id, thread_id, K_RA, ptrG,
     valG_old, ptrG_new, valG_new, 'active') @t
==>  (Ex z. valG_new = valG_old + z)
```

5.2.3 Analysis Results

Tamarin generates an attack path as follows. The attacker replays the same signal to two thread instances; the execution of the two threads are manipulated and synchronized until the check instruction (step 3 in Figure 4); they both update the global state variable with the same value at step 4. The global state variable is used to track the most recent signal counter value and prevent replay. However, the attacker could use the method to extend the subscription period of a user by using stale state values.

We have verified the attack on the original version of the code, confirming the effectiveness of the attack. To fix the vulnerability, we introduce a per process lock for global state variable check-and-set instruction. We patched Tamarin model accordingly, and the security property was proven.

5.3 State Continuity with Sealed Data

BI-SGX [42] is an open-source project that aims to provide a confidential cloud platform with Intel SGX. BI-SGX supports two types of users: 1) *data owners* (*e.g.*, patients) who own the data and upload it to the cloud; 2) *data users* (*e.g.*, researchers or medical practitioners), who utilize, analyze, and perform computations on the data.

The protocol of BI-SGX is described in Figure 5. The data owner sends data to the SGX application via the ecall *seal_data*, which decrypts the messages from the secure channel, extracts the data and owner credentials, and wraps the data and its ownership into a sealed chunk. The ecall returns the sealed data, which is then stored in a database with a monotonically increasing counter value as the index. In this way, each uploaded data from the same data owner ends up in the database with a unique index value.

To perform computation over the data, a researcher sends an encrypted query to the SGX application. Some of these queries may include an index to specify the target data. In this case, the request is processed inside the ecall *run_interpreter*, which issues an ocall with the requested index as the input. The ocall queries the database with the index and locates the sealed blob. The enclave unseals the sealed blob and performs customized operations over the data and returns the results to the users through the secure channel.

5.3.1 Tamarin Models

The association network consists of three layers: *platform*, *ISV/enclave-binary*, and *enclave-process*. We model two ecalls, one for the data owner's upload request and the other for the user's data query. The ecall for upload request is modeled as single rule and the ecall for data query is modeled as a sequence of two thread rules. For the user's data query, we model only the data retrieval and ignore the concrete operations the user is interested.

Two types of events need to be labeled: (1) the user's request of data is marked by an action-label `RCHR_rcv (RA_session_k, index_req)` with `RA_session_k` representing the RA session key and `index_req` indicating the index of the requested data; (2) the enclave's response is marked by another action-label `E_reply(RA_session_k, index_req,seal)` at the ecall *run_interpreter* with `seal` representing the sealed data obtained by the BI-SGX enclave when processing the user's query.

RA session keys and GMAC tag are abstracted in the same way as described in §5.2.1. Database is modeled as described in §4.8 with authentication. Integrity is abstracted by using a dedicated database fact. The communication with the database occurs over public channel as it is handled by untrusted code.

We introduce five helper lemmas: Two for preserving the MC properties as described in §5.1.2; three others for proving that each user and BI-SGX enclave communication uses a unique RA session.

5.3.2 Security Property

The key challenge is to properly model state continuity in this case. A replay occurs if the same data is retrieved and processed by the BI-SGX enclave when the user sends queries with different indexes. Hence, the security property considered is that with queries containing different indexes, different data is retrieved and processed. The property, as shown below, indicates that when two users' queries containing different indexes are processed, the sealed data involved in the processing must be different.

```
All RCHR_rcv(RA_session_x, index_x) @t1
  & RCHR_rcv(RA_session_y, index_y) @t2
  & not(index_x = index_y)
==>
  Ex E_reply(RA_session_x, index_x, seal_a) @t3
     & (All E_reply(RA_session_x, index_t, seal_t) @t4
          ==> #t3 = #t4)
   & E_reply(RA_session_y, index_y, seal_b) @t5
     & (All E_reply(RA_session_y, index_t, seal_t) @t6
          ==> #t5 = #t6)
   & not(seal_a = seal_b)
```

5.3.3 Analysis Results

By running the prover, Tamarin shows a replay attack of sealed data. The root cause of this attack is that the association between the index and data is maintained in the untrusted storage, *i.e.*, the database. Hence, the adversary could alter the mapping and replay the sealed data. We have also confirmed the effectiveness of the attack in practice. To fix this vulnerability, we implement the mapping of the index and the data within the enclave using MC, preventing the adversary from modifying such mapping. In particular, we add MC value inside the sealed data which can act as an index of the user query. Since, the adversary cannot modify the index stored inside sealed data, she cannot replay a sealed data for any index other than the one stored inside. This index is checked in the ecall *run_interpreter* to match with user's requested index. The property of state continuity was then proven using the updated Tamarin model.

5.4 Summary of Case Studies

In the three case studies, Sawtooth tries to preserve the states of PoET certificate generation using monotonic counters; Heartbeat tries to maintain the recently received active heartbeat signals, and BI-SGX tries to preserve the one-to-one mapping between the index and the sealed data. With Tamarin, we are able to capture these vulnerabilities by carefully modeling adversary behaviour and enclave operations.

Responsible disclosure. We have disclosed the vulnerabilities to developers of these three projects. The Sawtooth team have acknowledged our findings and patched the vulnerabilities we discovered [2]. Developers of BI-SGX have planned to address the discovered issue by altering the design of BI-SGX.

We run the Tamarin prover (v1.7.0) on a machine with a quad-core 1.80GHz Intel© Core™ i7-8550U CPU and 16 GB RAM, and Ubuntu Linux 18.04. We introduced helper lemmas— two for Sawtooth, five for Heartbeat, and five for BI-SGX—to help prove the target properties. We can see that with vulnerable models studied in this paper, Tamarin could discover attack traces within a couple of minutes, with a longest case (Sawtooth) being 78 seconds. While for patched versions, the proofs take a couple of hours to finish, with a longest case (Heartbeat) of 2 hours and 4 minutes.

Table 2: Verification time and size of the Tamarin models.

App	Attack Discovery Time	Verification Time	# Rules	Model LOC
Sawtooth [1]	1m 18s	25s	11	300
Heartbeat [19]	7s	2h 4m 7s	11	250
BI-SGX [42]	36s	37s	18	450

6 Discussion and Limitations

There are two major limitations of our approach. First, the verification process is not completely automated. It requires the users to manually translate the source code or design logic of the enclave program into the Tamarin model. Such manual efforts typically include modeling the program logic in Tamarin, encoding the property of state continuity as a lemma expressed in first-order logic, ensuring correct syntax and protocol behavior using executability lemmas [5], validating the results from Tamarin's output, and so on. As Tamarin is a semi-automated tool, the users are also expected to interact with Tamarin and refine the proof with several iterations.

Second, Tamarin may encounter non-termination problems. When Tamarin models become complicated, the verification

process may take very long time and sometimes never terminate. Reasons of non-termination include *partial deconstructions*, *looped construction*, and *undecidability*. Details about partial deconstructions and solution have been discussed in prior studies [20, 31]. Looped construction and unbounded instantiation of terms force Tamarin to resolve similar constraints repeatedly without converging the state spaces. After careful observation of the recursive constraint structures in the Tamarin interactive GUI, users can build helper lemmas (§2.2.5) that prevent the proof branch from entering into repeated loop, and thus allowing Tamarin to terminate in many cases. Additionally, a looped construction could be partially mitigated by constructing induction lemmas to discard recursive dependencies, and by using restriction axioms to minimize loops construction. Nevertheless, proving a property for a given model is undecidable. Therefore, it is impossible to ensure a termination in all cases. We plan to contribute to the Tamarin community and improve the tool in future work.

Admittedly, our work is only the first step towards automated verification of state continuity for SGX enclave programs. While our approach in theory can be applied to large, complex programs, the manual efforts involved remains a major obstacle for developers to apply this approach in practice. Future work will aim to fully automate the verification process for developers with minimal expertise in Tamarin. For instance, we will extend our approach with LLVM to automate the extraction of SGX primitives and integrate our solution with a learning-based approach to resolve non-termination problems.

7 Related Works

Various solutions to provide state continuity have been proposed. Memoir [41], ICE [49] and Ariadne [50] implement libraries to interact with non-volatile memory protected by TPM chips to provide freshness and integrity protection upon each usage within untrusted code. These libraries act as intermediary between TPM chips and untrusted code. To overcome the limitation of slow speed of non-volatile memory writes, these works suggest to reduce the number of writes by accessing the TPM only at boot time [41,49] or flipping only a single bit per write using gray-code [50]. While these centralized solutions require TPM chips, ROTE [38] and LCM [16] provide distributed state-continuity solutions for state continuity.

Another line of research focuses on formally modeling and proving the security of state continuity provided by these libraries and frameworks, which is also the focus of our paper. In particular, Ahman *et al.* provide assertion based constructs in F* verification tool for state preservation [6]. It introduces monotonic state interfaces and stable predicates for efficient modeling of states. RollSec [22] is a prototype framework for extracting variable based program states using program syntax, control flow and data flow information. It requires monitoring, recording and compensation modules to identify and fix state related rollback issues. However, these works do not cover state rollback in TEE applications studied in this paper.

Moat [46] uses Boogie verifier [11] and Z3 SMT solver [24] to provide assertion based formal framework to verify confidentiality of enclave programs. Xu *et al.* provides Tamarin based formal framework for modeling to prove confidentiality, authentication and privacy for ARM TrustZone's chain of trust and attestation protocols of TEE based applications [54]. Jacomme *et al.* extend SAPIC tool for Tamarin by providing encodings for report functionality for TEE based applications [33]. The report functionality is introduced to extend Tamarin modeling to use direct reporting construct in SAPIC pi calculus language to prove authentication of TEE applications to remote clients. Our work focuses modeling and proving properties different from these work, *i.e.*, state continuity of SGX applications using Tamarin.

For state-continuity solutions based on TPM chips, TPM/TPM2.0 related interaction and applications are formally verified [9,23,25,44,47]. These works model TPM specific interfaces (API or TPM commands), configuration registers and secure key management unit and prove confidentiality, remote or local attestation (direct anonymous attestation and root of trust for measurement) [9, 23, 47] and authorization [25, 44] of the interacting applications. Our work covers a broader range of state-continuity scenarios. For applications that requires TPM chips, our focus is to verify whether the SGX applications use these TPM chips correctly.

8 Conclusion

In this paper, we make the first attempt towards symbolic verification of state continuity properties for enclave programs. We show that SGX-specific semantics and operations can be modeled as multiset re-writing rules and the state continuity property can be reasoned using the Tamarin Prover. We have shown the effectiveness of the method on three types of state continuity flaws in three open-source projects. Our study shows the great potential of symbolic verification tools, such as Tamarin Prover, in more diverse and complex scenarios.

Acknowledgments

We are grateful to Cas Cremers for valuable feedback in shepherding our paper, as well as other reviewers for their insightful comments. We also would like to thank the Tamarin community particularly Jannik Dreier, Jonathan Hoyland, Benjamin Kiesl, and Kevin Milner at Google forum for providing insight and active support of Tamarin. The authors at The Ohio State University were partially supported by NSF grant 1834213.

References

[1] Hyperledger Sawtooth. Retrieved January 20, 2021 from https://www.hyperledger.org/use/sawtooth.

[2] Hyperledger Sawtooth-PoET patch. Retrieved January 12, 2021 from https://github.com/hyperledger/sawtooth-poet/commit/6f9db4998a11b427c6a24ea42f9891cb9ff0101e.

[3] Hyperledger Sawtooth-PoET vulnerable. Retrieved January 12, 2021 from https://github.com/hyperledger/sawtooth-core/releases/tag/v1.0.5, Filepath /consensus/poet/sgx/sawtooth_poet_sgx/libpoet_enclave/poet_enclave.cpp.

[4] Intel SGX Software Development Kit (SDK). Retrieved January 27, 2021 from https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html.

[5] Tamarin Manual. Retrieved January 18, 2021 from https://tamarin-prover.github.io/manual/book/001_introduction.html.

[6] Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: Foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2, December 2017.

[7] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for cpu based attestation and sealing. In *2nd HASP*, 2013.

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel SGX. In *12th USENIX OSDI*, 2016.

[9] Guangdong Bai, Jianan Hao, Jianliang Wu, Yang Liu, Zhenkai Liang, and Andrew Martin. Trustfound: Towards a formal foundation for model checking trusted computing platforms. In *International Symposium on Formal Methods*, pages 110–126. Springer, 2014.

[10] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. Cryptology ePrint Archive, Report 2019/1393, 2019. https://eprint.iacr.org/2019/1393.

[11] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[12] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, page 1383–1396. ACM, 2018.

[13] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems*, 33(3), August 2015.

[14] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy*, pages 483–502, Los Alamitos, CA, USA, may 2017. IEEE Computer Society.

[15] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[16] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 157–168, 2017.

[17] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Stealing intel secrets from sgx enclaves via speculative execution. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, June 2019.

[18] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2008.

[19] Y. Chen, W. Sun, N. Zhang, Q. Zheng, W. Lou, and Y. T. Hou. Towards efficient fine-grained access control and trustworthy data processing for remote monitoring services in iot. *IEEE Transactions on Information Forensics and Security*, 14(7):1830–1842, 2019. Github: https://github.com/fishermano/SGXEnabledAccess.

[20] Véronique Cortier, Stéphanie Delaune, and Jannik Dreier. Automatic generation of sources lemmas in tamarin: towards automatic proofs of security protocols. In *European Symposium on Research in Computer Security*, pages 3–22. Springer, 2020.

[21] Cas Cremers, Marko Horvat, Jonathan Hoyland, Samuel Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *ACM SIGSAC Conference on Computer and Commuincations Security*, pages 1773–1788. ACM, October 2017.

[22] Weiqi Dai, Yukun Du, Hai Jin, Weizhong Qiang, Deqing Zou, Shouhuai Xu, and Zhongze Liu. Rollsec: Automatically secure software states against general rollback. *International Journal of Parallel Programming*, 46:788–805, 2017.

[23] Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kaynar. A logic of secure systems and its application to trusted computing. In *2009 30th IEEE Symposium on Security and Privacy*, pages 221–236. IEEE, 2009.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[25] Stéphanie Delaune, Steve Kremer, Mark D Ryan, and Graham Steel. A formal analysis of authentication in the tpm. In *International Workshop on Formal Aspects in Security and Trust*, pages 111–125. Springer, 2010.

[26] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy*, pages 463–482, 2017.

[27] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[28] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, page 193–206. ACM, 2003.

[29] Guillaume Girol, Lucca Hirschi, R. Sasse, D. Jackson, C. Cremers, and David Basin. A spectral analysis of noise: A comprehensive, automated, formal analysis of diffie-hellman protocols. In *USENIX Security Symposium*, 2020.

[30] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11(10.1145):2487726–2488370, 2013.

[31] Jonathan Hoyland. *An Analysis of TLS 1.3 and its use in Composite Protocols*. PhD thesis, 2018. Royal Holloway, University of London.

[32] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX OSDI*, 2016.

[33] C. Jacomme, S. Kremer, and G. Scerri. Symbolic models for isolated execution environments. In *2017 IEEE European Symposium on Security and Privacy-*, pages 530–545, 2017.

[34] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[35] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symposium on Security and Privacy*, pages 435–450, 2017.

[36] N. Kobeissi, G. Nicolas, and K. Bhargavan. Noise explorer: Fully automated modeling and verification for arbitrary noise protocols. In *2019 IEEE European Symposium on Security and Privacy*, pages 356–370, 2019.

[37] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *Journal of Computer Security*, 24(5):583–616, 2016.

[38] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium*, pages 1289–1306, Vancouver, BC, August 2017. USENIX Association.

[39] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isolated execution. In *2nd HASP*, 2013.

[40] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. pages 696–701, 2013.

[41] Bryan Parno, Jay Lorch, John (JD) Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2011.

[42] Aoi Sakurai. BI-SGX: Secure Cloud Computation. *58th SIGBIO Bioinformatics Study Group, Japan*, 2019. Website: https://bi-sgx.net/, Github: https://github.com/hello31337/BI-SGX.

[43] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *36th IEEE Symposium on Security and Privacy*, 2015.

[44] Jianxiong Shao, Yu Qin, Dengguo Feng, and Weijin Wang. Formal analysis of enhanced authorization in the tpm 2.0. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 273–284, 2015.

[45] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. A practical encrypted data analytic framework with trusted processors. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, Dallas, TX, November 2017.

[46] Rohit Sinha, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *The ACM Conference on Computer and Communications Security*. ACM, October 2015.

[47] Ben Smyth, Mark Ryan, and Liqun Chen. Formal analysis of anonymity in ecc-based direct anonymous attestation schemes. In *International Workshop on Formal Aspects in Security and Trust*, pages 245–262. Springer, 2011.

[48] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: a novel approach to efficient automatic security protocol analysis 1. *Journal of Computer Security*, 9(1-2):47–74, 2001.

[49] Raoul Strackx, Bart Jacobs, and Frank Piessens. Ice: A passive, high-speed, state-continuity scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference*, page 106–115. ACM, 2014.

[50] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium*, 2016.

[51] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. Cryptology ePrint Archive, Report 2016/635, 2016.

[52] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC*, 2017.

[53] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *Computer Security – ESORICS 2016*, pages 440–457, 2016.

[54] Shiwei Xu, Yizhi Zhao, Zhengwei Ren, Lingjuan Wu, Yan Tong, and Huanguo Zhang. A symbolic model for systematically analyzing tee-based protocols. In *International Conference on Information and Communications Security*, pages 126–144. Springer, 2020.

[55] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[56] F. Zhang, E. Cecchetti, K. Croman, A. Juels, , and E. Shi. Town crier: An authenticated data feed for smart contracts. In *23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[57] J. Zhu, X. Yan, and W. Huang. A formal framework for state continuity of protected modules. In *2018 4th International Conference on Big Data Computing and Communications*, pages 114–119, 2018.

A Rule Execution Criteria

In the Tamarin proof process, only certain rules are considered to be part of the trace. For a given target lemma, a candidate rule is considered executable only if it satisfies the following criteria:

1. The premise facts (except the built-in facts Fr and In) of the candidate rule should be produced by other rules and can be consumed from the current system state.
2. The variables of action-labels specified in the candidate rule's action part should comply with the model's restriction axioms and the target lemma's variable constraints.
3. The execution of the rule should respect the *type restriction* (§2.2.4) constraint for all the variables prefixed with '~' symbol. However, this restriction is nullified if the variables are part of a persistent fact.
4. Variables with the same name across all received facts of a rule should receive the same value (*pattern matching*).
5. The order of the candidate rule's execution, for timepoints of all action-labels of the candidate rule, should satisfy the timepoint constraints specified in model's restriction axioms and the target property.
6. If the candidate rule execution is part of the target lemma, which is influenced by a helper lemma (§2.2.5), the rule execution should satisfy the helper lemma's constraints.

The rules satisfying above conditions can be executed in parallel. Upon execution of a rule, the consumed linear facts are removed from the system state and the produced facts are added to the system state. During the verification process, the backward search algorithm ensures that a valid rule execution trace satisfy the above mentioned criteria and the trace maintains a consistent system state when looking at the top-down execution of the model.

B Tamarin Sawtooth attack trace

Figure 6 shows a Sawtooth attack (§5.1.3) produced by Tamarin in interactive GUI mode. In the trace, ovals denote adversary actions; rectangle boxes denote model rules; bold and gray arrows denote fact dependency for linear and persistent respectively; dotted, red and black arrows show adversary message reuse, message deductions and public channel interaction.

The attack can be seen at the last two CWC rule instances of enclave-process instances `p_id` and `p_id.1`. In these rules, the certificate is generated with the same `MC_ref` value (symbolically denoted as '1'+'1') in the same platform with identity instance `platform`.

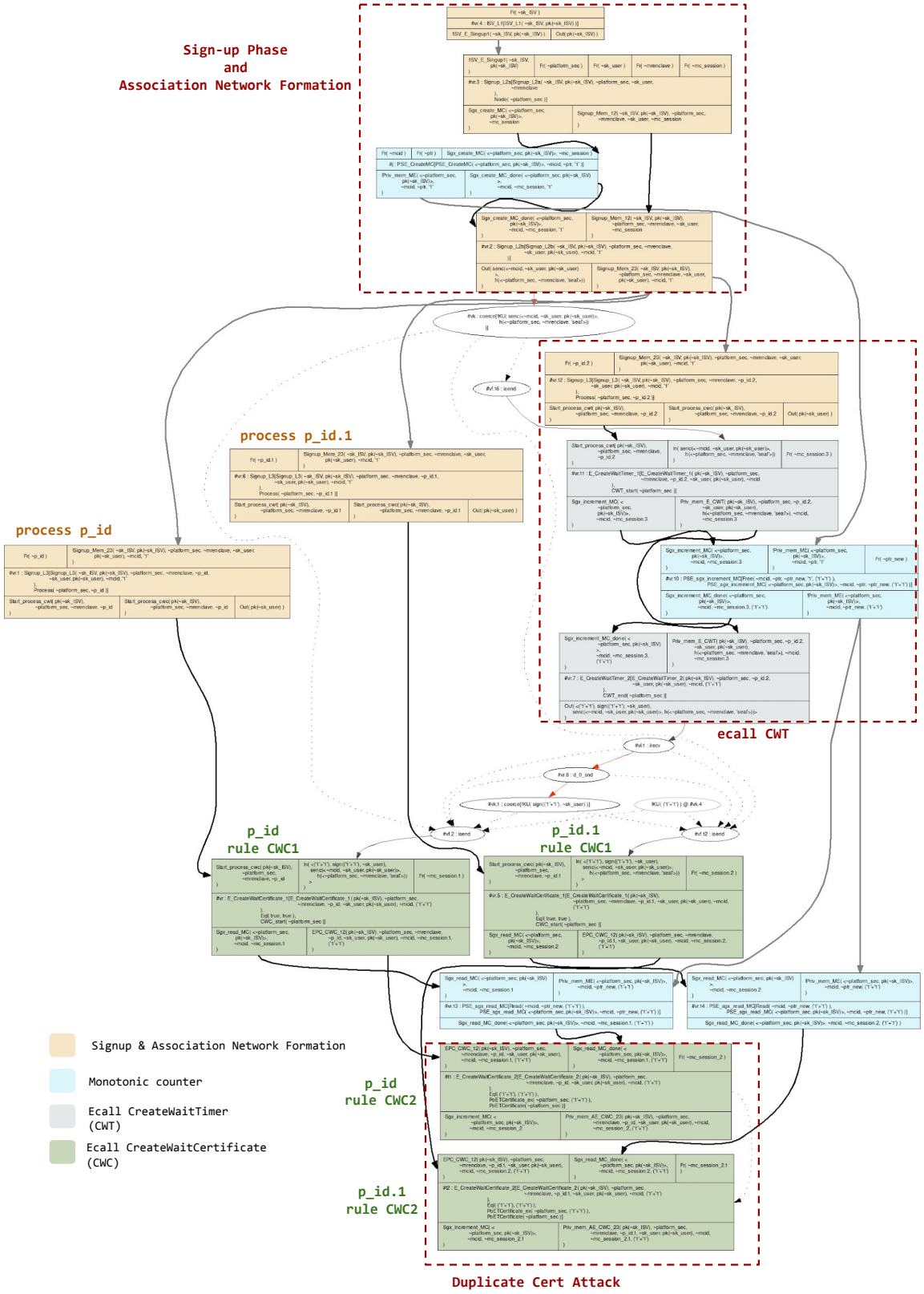

Figure 6: Tamarin produced attack trace for Sawtooth