
# ROTE: Rollback Protection for Trusted Execution

**Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, and Arthur Gervais, *ETH Zurich;* Ari Juels, *Cornell Tech;* Srdjan Capkun, *ETH Zurich***

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

**This paper is included in the Proceedings of the 26th USENIX Security Symposium**

**August 16–18, 2017 • Vancouver, BC, Canada**

ISBN 978-1-931971-40-9

**Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX**

# ROTE: Rollback Protection for Trusted Execution

Sinisa Matetic, *ETH Zurich*
Mansoor Ahmed, *ETH Zurich*
Kari Kostiainen, *ETH Zurich*
Aritra Dhar, *ETH Zurich*
David Sommer, *ETH Zurich*
Arthur Gervais, *ETH Zurich*
Ari Juels, *Cornell Tech*
Srdjan Capkun, *ETH Zurich*

## Abstract

Security architectures such as Intel SGX need protection against rollback attacks, where the adversary violates the integrity of a protected application state by replaying old persistently stored data or by starting multiple application instances. Successful rollback attacks have serious consequences on applications such as financial services. In this paper, we propose a new approach for rollback protection on SGX. The intuition behind our approach is simple. A single platform cannot efficiently prevent rollback, but in many practical scenarios, multiple processors can be enrolled to assist each other. We design and implement a rollback protection system called ROTE that realizes integrity protection as a distributed system. We construct a model that captures adversarial ability to schedule enclave execution and show that our solution achieves a strong security property: the only way to violate integrity is to reset all participating platforms to their initial state. We implement ROTE and demonstrate that distributed rollback protection can provide significantly better performance than previously known solutions based on local non-volatile memory.

## 1 Introduction

Intel Software Guard Extensions (SGX) enables execution of security-critical application code, called *enclaves*, in isolation from the untrusted system software [1]. Protections in the processor ensure that a malicious OS cannot read or modify enclave memory at runtime. To protect enclave data across executions, SGX provides a security mechanism called *sealing* that allows each enclave to encrypt and authenticate data for persistent storage. SGX-enabled processors are equipped with certified cryptographic keys that can issue remotely verifiable *attestation* statements on the software configuration of enclaves. Through these security mechanisms (isolation, sealing, attestation) SGX enables development of various applications and online services with hardened security.

The architecture has also its limitations. While sealing prevents a malicious OS from reading or arbitrarily modifying persistently stored enclave data, *rollback attacks* [2, 3, 4, 1] remain a threat. In a rollback attack a malicious OS replaces the latest sealed data with an older encrypted and authenticated version. Enclaves cannot easily detect this replay, because the processor is unable to maintain persistent state across enclave executions that may include platform reboots. Another way to violate state integrity is to create two instances of the same enclave and route update requests to one instance and read requests to the other. To remote clients that perform attestation, the instances are indistinguishable.

Data integrity violation through rollback attacks can have severe implications. Consider, for example, a financial application implemented as an enclave. The enclave repeatedly processes incoming transactions at high speed and maintains an account balance for each user or a history of all transactions in the system. If the adversary manages to revert the enclave to its previous state, the maintained account balance or the queried transaction history does not match the executed transactions.

To address rollback attacks, two basic approaches are known. The first is to store the persistent state of enclaves in a non-volatile memory element on the same platform. The SGX architecture was recently updated to support monotonic counters that leverage non-volatile memory [5]. However, the security guarantees and the performance limits of this mechanism are not precisely documented. Our experiments show that writes of counter values to this memory are slow (80-250 ms), which limits its use in high-throughput applications. More importantly, this memory allows only a limited number of write operations. We show that this limit is reached within just few days of continuous system use after which the memory becomes unusable. Similar limitations also apply to rollback protection techniques that leverage Trusted Platform Modules (TPMs) [2, 4, 3].

The second common approach is to maintain integrity information for protected applications in a sep-

arate trusted server [6, 7, 8]. The drawback of such solutions is that the server becomes an obvious target for attacks. Server replication using standard Byzantine consensus protocols [9] avoids a single point of failure, but requires high communication overhead and multiple replicas for each faulty node.

In this paper we propose a new approach to protect SGX enclaves from rollback attacks. The intuition behind our solution is simple. A single SGX platform cannot prevent rollback attacks efficiently, but in many practical scenarios the owner or the owners of processors can assign multiple processors to assist each other. Our approach realizes rollback protection as a distributed system. When an enclave updates its state, it stores a counter to a set of enclaves running on assisting processors. Later, when the enclave needs to recover its state, it obtains counter values from assisting enclaves to verify that the recovered state data is of the latest version.

We consider a powerful adversary that controls the OS on the target platform and on *any* of the assisting platforms. Additionally, we even assume that the adversary can break SGX protections on some of the assisting processors and control all network communication between the platforms. Our adversary model combines commonly considered network control based on the standard Dolev-Yao model [10] and Byzantine faults [11, 12], but additionally captures the the ability of the adversary to restart trusted processes from a previously saved state and to run multiple instances of the same trusted process. Such adversarial capabilities are crucial for the security analysis of our system, and we believe that the model is of general interest. In fact, using our model we found potential vulnerabilities in recent SGX systems [3, 13, 14].

Secure and practical realization of distributed rollback protection under such a strong adversarial model involves several challenges. One of the main challenges is that when an assisting enclave receives a counter, its own state changes, which implies a set of new state updates that would in turn propagate. To prevent endless update propagation, the counter value must be stored in the volatile runtime memory of enclaves. However, the assisting enclaves may be restarted at any time. Moreover, the adversary can also create multiple instances of the same enclave on all assisting platforms and route counter writes and reads to separate instances.

We design and implement a rollback protection system called ROTE (Rollback Protection for Trusted Execution). The main components of our solution are a state update mechanism that is an optimized version of consistent broadcast protocols [15, 16], and a recovery mechanism that obtains lost counters from the rest of the protection group upon enclave restart. We also design a session key update mechanism to address attacks based on multiple enclave instances.

Our solution achieves a strong security property that we call *all-or-nothing rollback*. Although the attacker can restart enclaves freely, and thus implement subtle attacks where enclave state updates and recovery are interleaved, the adversary cannot roll back any single enclave to its previous state. The only way to violate data integrity is to reset the entire group to its initial state. If desired, similar to [4, 2], our approach can also provide crash resilience, assuming deterministic enclaves and a slightly weaker notion of rollback prevention (the latest input can be executed twice).

We implemented ROTE on SGX and evaluated its performance on four SGX machines. We tested larger groups of up to 20 platforms using a simulated implementation over a local network and geographically distributed enclaves. Our evaluation shows that state updates in ROTE can be very fast (1-2 ms). The number of counter increments is unlimited. This is in contrast to solutions based on SGX counters and TPMs, where state updates are approximately 100 times slower and limited. Compared to Byzantine consensus protocols, our approach requires significantly fewer replicas ($f+1$ instead of the standard $3f+1$). Enclave developers can use our system through a simple API. The ROTE TCB increment is moderate (1100 LoC).

**Contributions.** We make the following contributions.

- *New security model.* We introduce a new security model for reasoning about the integrity and freshness of SGX applications. Using the model we identified potential security weaknesses in existing SGX systems.
- *SGX counter experiments.* We show that SGX counters have severe performance limitations.
- *Novel approach.* We propose a novel way to protect SGX enclaves. Our main idea is to realize rollback protection by storing enclave-specific counters in a distributed system of collaborative enclaves on distinct nodes.
- ROTE. We propose and implement a system called ROTE that effectively protects against rollback attacks. ROTE ensures integrity and freshness of application data in a powerful adversarial model.
- *Experimental evaluation.* We demonstrate that distributed rollback protection incurs only a small performance overhead. When deployed over a low-latency network, the state update overhead is only 1-2 ms.

The rest of this paper is organized as follows. Section 2 explains models and rollbacks attacks. Section 3 describes our approach. Section 4 describes the ROTE system and Section 5 provides security analysis. Section 6 provides performance evaluation and Section 7 further discussion. We review related work in Section 8. Section 9 concludes the paper.

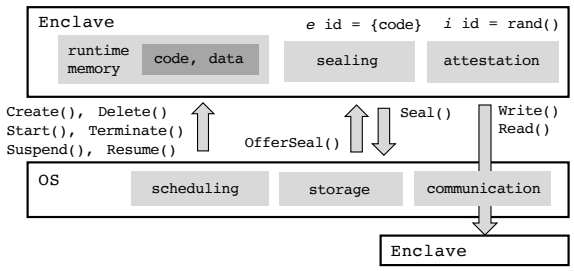



Figure 1: Modeled SGX operations.

## 2 Problem Statement

In this section we define models for the SGX architecture and the adversary. After that, we explain rollback attacks, limitations of known solutions, and our requirements. Appendix A provides a summary of the SGX architecture for readers that are not familiar with it.

### 2.1 SGX Model

Figure 1 illustrates our SGX model. We model enclaves and the operating system, their main functionality, and the operations through which they interact. Our model captures the main SGX functionalities that are available on all SGX platforms.

**Scheduling operations.** Enclave execution is scheduled by the OS.

• $e \leftarrow \texttt{Create}(code)$. The system software running on the OS can create an enclave by providing its code. The SGX architecture creates a unique enclave identifier $e$ that is defined by the code measurement.

• $i \leftarrow \texttt{Start}(e)$. The system software can start a created enclave using its enclave identifier $e$. The enclave generates a random and unique instance identifier $i$ for the enclave instance that executes the code that was assigned to it during creation. While an enclave instance is running, the OS and other enclaves are isolated from its runtime memory. Each enclave instance has its own program counter and runtime memory.

• $\texttt{Suspend}(i)$ and $\texttt{Resume}(i)$. The OS can suspend the execution of an enclave. When an enclave is suspended, its program counter and runtime memory retain their values. The OS can resume suspended enclave execution.

• $\texttt{Terminate}(i)$. The OS can terminate the enclave execution. At termination, the enclave runtime memory is erased by the SGX architecture and the enclave instance $i$ is rendered unusable.

**Storage operations.** The second set of operations is related to sealing data for local persistent storage.

• $s \leftarrow \texttt{Seal}(data)$. An enclave can save data for local persistent storage. This operation creates an encrypted, authenticated data structure $s$ that is passed to the OS.

• $\texttt{OfferSeal}(i,s)$. The OS can offer sealed data $s$. The enclave can verify that it previously created the seal, but the enclave cannot distinguish which seal is the latest. Every enclave instance $i$ can unseal data previously sealed by an instance of the same enclave identity $e$.

**Communication operations.** Due to attestation, a client can write data such that only a particular enclave can read it. The client can read data from an enclave and verify which enclave wrote it. We model these primitives as single operations that can be called from the same or remote platform, although attestation is an interactive protocol between the enclave and client.

• $\texttt{Write}(m_e, i)$. The OS can write message $m_e$ to an enclave instance $i$. Only an enclave with enclave identity $e$ can read the written message $m_e$.

• $m_e \leftarrow \texttt{Read}(i)$. The OS can read message $m_e$ from an enclave instance $i$. The read message $m_e$ identifies the enclave identity $e$ that wrote the data.

Note that remote attestation identifies the enclave identity, but not the platform identity, because the attestation protocol is either anonymous or returns client-specific pseudonyms (see Appendix A for details). In local attestation the platform is implicitly known.

We do not model platform reboots, as those have the same effect as enclave restarts. Our model assumes that the runtime memory of each enclave instance is perfectly isolated from the untrusted OS and other enclaves. We consider information leakage from side-channel attacks a realistic threat [17, 18, 19], but an orthogonal problem to rollback attacks, and thus outside of our model.

### 2.2 Local Adversary Model

We consider a powerful adversary who, after an initial trusted setup phase, controls all system software on the target platform, including the OS. Based on the SGX model, the adversary can schedule enclaves and start multiple instances of the same enclave, offer the latest and previous versions of sealed data, and block, delay, read and modify all messages sent by the enclaves.

The adversary cannot read or modify the enclave runtime memory or learn any information about the secrets held in enclave data. The adversary has no access to processor-specific keys, such as the sealing key or the attestation key, and the adversary cannot break cryptographic primitives provided by the SGX architecture. The enclaves may also implement additional cryptographic operations that the adversary cannot break.

The adversarial capabilities that we identified as part of the model can be critical for many SGX systems. The ability to schedule, restart and create multiple enclave instances, enables subtle attacks that we address in this paper. We analyzed SGX systems using this model and found vulnerabilities that can be addressed through the techniques developed in this paper. These findings are reported in an extended version of this paper [20].

## 2.3 Rollback Attacks

The goal of the adversary is to violate the integrity of the enclave's state. This is possible with a simple rollback attack. After an enclave has sealed at least two data elements $s_1 \leftarrow \texttt{Seal}(d_1)$ and $s_2 \leftarrow \texttt{Seal}(d_2)$, the adversary performs `Terminate()` and `Start()` to erase the runtime memory of the enclave. When the enclave requests for the latest sealed data $d_2$, the adversary performs $\texttt{OfferSeal}(i, s_1)$ and the enclave accepts $d_1$ as $d_2$. When the sealed data captures the state of the enclave at the time of sealing, we say that the rollback attack reverts the enclave back to its previous state.

Another approach is a *forking attack*, where the adversary leverages two concurrently running enclave instances. The adversary starts two instances $i_1 \leftarrow \texttt{Start}(e)$ and $i_2 \leftarrow \texttt{Start}(e)$ of the same enclave $e$. The OS receives a request from a remote client to write data $m_e$ to enclave $e$. The OS writes the data to the first enclave instance $\texttt{Write}(m_e, i_1)$ which causes a state change. Another remote client sends a request to read data from the enclave $e$. The OS reads data from the second instance $m_e \leftarrow \texttt{Read}(i_2)$ which has an outdated state and returns $m_e$ to the client. The SGX architecture does not enable one enclave instance to check if another instance of the same enclave code is already running [21].

Such attacks can have severe implications, especially for applications that maintain financial data, such as account balances or transaction histories.

## 2.4 Limitations of Known Solutions

**SGX counters.** Intel has recently added support for monotonic counters [5] as an optional SGX feature that an enclave developer may use for rollback attack protection, when available. However, the security and performance properties of this mechanism are not precisely documented. We performed a detailed analysis of SGX counters and report our findings in Appendix B.

To summarize, we found out that counter updates take 80-250 ms and reads 60-140 ms. The non-volatile memory used to implement the counter wears out after approximately one million writes, making the counter functionality unusable after a couple of days of continuous use. Thus, SGX counters are unsuitable for systems where state updates are frequent and continuous. Additionally, since the non-volatile memory used to store the counters resides outside the processor package, the mechanism is likely vulnerable to bus tapping and flash mirroring attacks [22] (see Appendix B for details).

**TPM solutions.** TPMs provide monotonic counters and NVRAM that can be used to prevent rollback attacks [4, 3, 2]. The TPM counter interface is rate-limited (typically one increment every 5 seconds) to prevent memory wear out.[1] Writing to NVRAM takes approximately 100 ms and the memory becomes unusable after 300K to 1.4M writes (few days of continuous use) [2]. Thus, also TPM based solutions are unsuitable for applications that require fast and continuous updates.

**Integrity servers.** Another approach is to leverage a trusted server to maintain state for protected applications [6, 7, 8]. The drawback of this approach is that the centralized integrity server becomes an obvious target for attacks. To eliminate a single point of failure, the integrity server could be replicated using a Byzantine consensus mechanism. However, standard consensus protocols, such as PBFT [9], require several rounds of communication, have high message complexity, and require at least three replicas for each faulty node.

**Architecture modifications.** Finally, the SGX architecture could be modified such that the untrusted OS cannot erase the enclave runtime memory. However, this approach would prevent the OS from performing resource management and would not scale to many enclaves. Additionally, rollback attacks through forced reboots and multiple enclave instances would remain possible. Another approach would be to enhance the processor with a non-volatile memory element. Such changes are costly and current NVRAM technologies have the performance limitations we discussed above.

## 2.5 Rollback Protection Requirements

The goal of our work is to design a rollback protection mechanism that overcomes the performance and security limitations of SGX counters and other known solutions. In particular, our solution should support unlimited and fast state updates, considering a strong adversary model without a single point of failure. When there is a trade-off between security and robustness, we favor security.

# 3 Our Approach

The intuition behind our approach is that a single SGX platform cannot efficiently prevent rollback attacks, but the owner or the owners of SGX platforms can enroll multiple processors to assist each other. Thus, our goal is to design rollback protection for SGX as a distributed system between multiple enclaves running on separate processors. Our distributed system is customized for the task of rollback protection to reduce the number of required replicas and communication.

To realize rollback protection, the distributed system should provide, for each participating platform, an ab-

[1] The TPM 2.0 specifications introduce *high-endurance non-volatile memory* that enables rapidly incremented counters [23]. The counter value is maintained in RAM and the value is flushed to non-volatile memory periodically (e.g., mod 100) and at controlled system shutdown. However, if the system is rebooted without calling TPM Shutdown, the counter value is lost and at start-up the TPM assumes the next periodic value. Therefore, such counters do not prevent attacks where the adversary reboots the system.

straction of a *secure counter storage* that consists of two operations:

- `WriteCounter`(*value*). An enclave can use this operation to write a counter value to the secure storage.[2]
- *value*/*empty* ← `ReadCounter()`. An enclave can use this operation to read a counter value from the secure storage. The operation returns the last written value or an empty value if no counter was previously written.

When an enclave performs a security-critical state update operation (e.g., modifies an account balance or extends a transaction history), it distributes a monotonically increasing counter value over the network to a set of enclaves running on assisting processors (`WriteCounter`), stores the counter value to its runtime memory and seals its state together with the counter value for local persistent storage. When the enclave is restarted, it can recover its latest state by unsealing the saved data, obtaining the counter values from enclaves on the assisting processors (`ReadCounter`) and verifying that the sealed state is of the latest version. The same technique allows potentially concurrently running instances of the same enclave identity to determine that they have the latest state. When an enclave needs to verify its state freshness (e.g., upon receiving a request to return the current account balance or transaction history to a remote client), it obtains the counter value from the network (`ReadCounter`) and compares it to the one in its runtime memory. By using enclaves on the assisting platforms, we reduce the required trust assumptions on the assisting platforms.

## 3.1 Distributed Model

We use the term *target platform* to refer to the node which performs state updates that require rollback protection. We assume $n$ SGX platforms that assist the target platform in rollback protection The platforms can belong to a single administrative domain or they could be owned by private individuals who all benefit from collaborative rollback protection. We model each platform using the SGX model described in Section 2.1. The distributed system can be seen as a composition of $n+1$ SGX instances (target platform included) that are connected over a network. We make no assumptions about the reliability of the communication network, messages may be delayed or lost completely. We assume that while participating in collaborative rollback protection, some platforms may be temporarily down or unreachable.

**Distributed adversary model.** On each platform, the adversary has the capabilities listed in Section 2.2. Additionally, we assume that the adversary can compromise the SGX protections on $f < n$ participating nodes, excluding the target platform. Such compromise is possible, e.g., through physical attacks. On the compromised SGX nodes the adversary can freely modify the runtime memory (code and data) of any enclave, and read all enclave secrets and the SGX processor keys.

This adversarial model combines a standard Dolev-Yao network adversary [10] with adversarial behaviour (Byzantine faults) on a subset of participating platforms [11, 12]. In addition, the adversary can schedule the execution of trusted processes, replay old versions of persistently stored data, and start multiple instances of the same trusted process on the same platform. In Section 5 we explain subtle attacks enabled by such additional adversarial capabilities.

## 3.2 Challenges

Secure and practical realization of our approach under a strong adversarial model involves challenges.

**Network partitioning.** A simple solution would be to store a counter with all the assisting enclaves, and at the time of unsealing require that the counter value is obtained from all assisting enclaves. However, if one of the platforms is unreachable at the time of unsealing (e.g., due to network error, maintenance or reboot), the operation would fail. Our goal is to design a system that can proceed even if some of the participating enclaves are unreachable. In such a system, some of the assisting enclaves may have outdated counter values, and the system must ensure that only the latest counter value is ever recovered, assuming an adversary that can block messages, and partition the network by choosing which nodes are reachable at any given time.

**Coordinated enclave restarts.** When an enclave seals data, it sends a counter value to a set of enclaves running on assisting platforms and each enclave must store the received counter. However, sealing the received counter for persistent storage would cause a new state update that would propagate endlessly. Therefore, the enclaves must maintain the received counters in their runtime memory. The participating enclaves may be restarted at any time, which causes them to lose their runtime memory. Thus, the rollback protection system must provide a recovery mechanism that allows the assisting enclaves to restore the lost counters from the other assisting enclaves. Such a recovery mechanism opens up a new attack vector. The adversary can launch coordinated attacks where he restarts assisting enclaves to trigger recovery while the target platform is distributing its current counter value.

**Multiple enclave instances.** Simple approaches that store a counter to a number of assisting enclaves and later read the counter from sufficiently many of the same enclaves are vulnerable to attacks where the adversary creates multiple instances of the same enclave. Assume that

[2] We use counter *write* abstraction instead of counter *increment*, because our distributed secure storage implementation allows writing of any counter value to the storage. However, the ROTE system only performs monotonic counter increments using this functionality.

a counter is saved to the runtime memory of all assisting enclaves. The adversary that controls the OS on all assisting platforms starts second instances of the same enclave on all platforms. The target enclave updates its state and sends an incremented counter to the second instances. Later, the target enclave obtains an old counter value from the first instances and recovers a previous state from the persistent storage.

# 4 ROTE System

In this section we describe ROTE (Rollback Protection for Trusted Execution), a distributed system for state integrity and rollback protection on SGX. We explain the counter increment technique, our system architecture, group assignment and system initialization. After that, we describe the rollback protection protocols.

## 4.1 Counter Increment Technique

Two common techniques for counter-based rollback protection exist. The first technique is *inc-then-store*, where the enclave first increments the trusted counter and after that updates its internal state and stores the sealed state together with the counter value on disk. This approach provides a strong security property (no rollback to any previous state), but if the enclave crashes between the increment and store operations, the system cannot recover from the crash.

The second technique is *store-then-inc*, where the enclave first saves its state on the disk together with the latest input value, after that increments the trusted counter, and finally performs the state update [4, 2]. If the system crashes, it can recover from the previous state using the saved input. This technique requires a deterministic enclave and provides a slightly weaker security property: arbitrary rollback is not possible, but the last input may be executed twice on the same enclave state [2].

The stronger security guarantee is needed, for example, in enclaves that generate random numbers, communicate with external parties or create timestamps. Consider a financial enclave that receives a request message from an external party and for each request it should create only one signed response that is randomized or includes a timestamp (`sgx_get_trusted_time` [24]). If store-then-inc is used, the adversary can create multiple different signed responses for the same request.[3]

The weaker security guarantee is sufficient in applications where the execution of the same input on the same state provides no advantage for the adversary.

[3]While some enclaves that require random numbers can be made deterministic by using a stateful PRNG and including its state to the saved enclave state, this may be difficult for enclaves that reuse code from existing libraries not designed for this. Similarly, some replay issues can be addressed on the protocol level, but enclave developers do not always have the freedom to change (standardized) protocols.

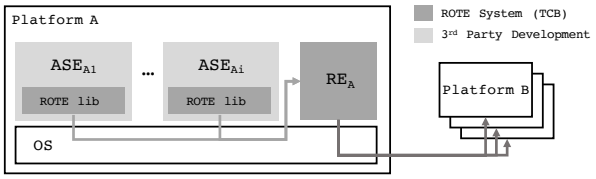



Figure 2: The ROTE system architecture.

In this paper we instantiate ROTE using inc-then-store, because of its strong security guarantee for any enclave. Our goal is to build a generic platform service that can protect various applications. We emphasize that if crash tolerance is required, then store-then-inc should be used. A rollback protection system could even support both counter increment techniques and allow developers to choose the protection style based on their application.

## 4.2 System Architecture

Figure 2 shows our system architecture. Each platform may run multiple user applications that have a matching Application-Specific Enclave (ASE). The ROTE system consists of a system service that we call the Rollback Enclave (RE) and a ROTE library that ASEs can use for rollback protection.

When an ASE needs to update its state, it calls a counter increment function from the ROTE library. Once the RE returns a counter value, the ASE can safely update its state, save the counter value to its memory and seal any data together with the counter value. When an ASE needs to verify the freshness of its state, it can again call a function from the ROTE library to obtain the latest counter value to verify the freshness of unsealed seal data (or state in its runtime memory).

The RE maintains a Monotonic Counter (MC), increases it for every ASE update, distributes it to REs running on assisting platforms, and includes the counter value to its own sealed data. When the RE needs to verify the freshness of its own state, it obtains the latest counter value from the assisting nodes. The RE realizes the secure counter storage functionality (`WriteCounter` and `ReadCounter`) described in Section 3.

The design choice of introducing a dedicated system service (RE) hides the distributed counter maintenance from the applications. Having a separate RE increases the TCB of our system slightly, but we consider easier application development more important.

The ROTE system has three configurable parameters:

- $n$ is the number of assisting platforms,
- $f$ is the number of compromised processors, and
- $u$ is the maximum number of assisting platforms that can be unreachable or non-responsive at time of state update or read for the system to proceed. Platform restarts are typically less frequent events and during them we require all the assisting platforms to be responsive.

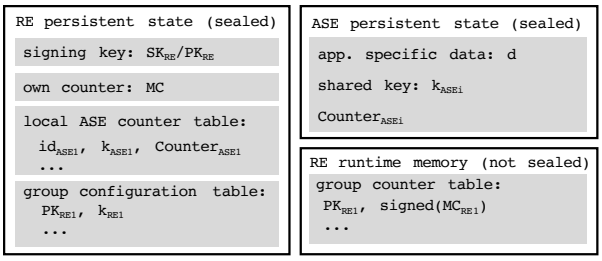



Figure 3: The ROTE system state structures.

These parameters have a dependency $n = f + 2u + 1$ (see Section 5). As an example, a system administrator can select the desired level of security $f$ and robustness $u$ which together determine the required number of assisting platforms $n$. Alternatively, given $n$ assisting platforms, the administrator can pick $f$ and $u$. Recall that standard Byzantine consensus protocols require always at least $3f + 1$ replicas.

To avoid *shared-fate* scenarios due to power outages or communication blockades, the participating platforms would ideally have independent or redundant power supply, battery backup, networking and OS maintenance.

## 4.3 System Initialization

Our system is agnostic to the way the $n$ assisting SGX platforms are chosen. Here we explain an example approach based on a trusted offline authority. Such group assignment is practical when all assisting platforms belong to a single administrative domain (e.g., multiple servers in the same data center). We call the trusted authority that selects the assisting nodes the *group owner*. The group owner can be a fully offline entity to reduce its attack surface. To establish a *protection group*, the group owner selects $n$ platforms.

In this section, we assume that the operating systems on these platforms are trusted at the time of system initialization (e.g., freshly installed OS). Note that although SGX supports remote attestation, this assumption is required, if the group needs to be established among *predefined* platforms. The SGX attestation is anonymous (or pseudonymous) and therefore it does not identify the attested platform. If the application scenario allows that the protection group can be established among *any* SGX platforms, then system initialization is possible without initially trusted operating systems using remote attestation. We discuss such group setup alternatives in Section 4.7.

During its first execution, the RE on each platform generates an asymmetric key pair $SK_{RE}/PK_{RE}$, and exports the public key. The public keys are delivered to the group owner securely, and the owner issues a certificate by signing all group member keys. The group certificate can be verified by the RE on each selected platform by hard-coding the public key of the group owner to the RE implementation.

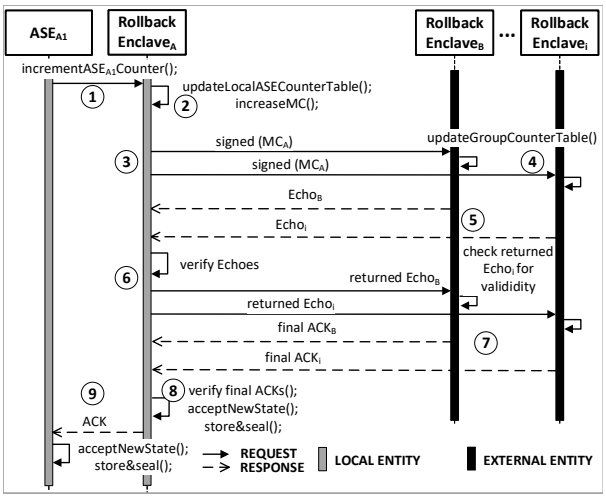



Figure 4: The ASE state update protocol.

The RE is started a second time with the certified list of public keys and a secret *initialization key* as input parameters. The purpose of this secret key for initialization is to indicate a legitimate group establishment operation and to prevent a later, parallel group creation by compromised operating systems on the same certified platforms (see Section 5). The initialization key is hard coded to the RE implementation in hashed format and the RE verifies the correctness of the provided key by hashing it. Without the correct key, the RE aborts initialization. The RE saves the list of certified public keys $PK_{REi}$ to a *group configuration table* and runs an authenticated key agreement protocol to establish pair-wise session keys $k_{REi}$ with all REs, and adds them to the group configuration table. Finally, the RE creates a monotonic counter (MC), sets it to zero, and seals its state.

When an ASE wants to use the ROTE system for the first time, it performs *local* attestation on the RE. The code measurement of the RE can be hard-coded to the ASE implementation or provisioned by the ASE developer. The ASE runs an authenticated key establishment protocol with the RE. The RE adds the established shared key $k_{ASEi}$ to a *local ASE counter table* together with a locally unique enclave identifier $id_{ASEi}$ and adds the same key to its own state. The used state structures are shown in Figure 3.

## 4.4 ASE State Update Protocol

When an ASE is ready to update its state (e.g., a financial application has received a new transaction and is ready to process it and update the maintained account balances), it starts the state update protocol shown in Figure 4. This protocol can be seen as a customized version of the Echo broadcast [15], as discussed in Section 8. The communication between the enclaves is encrypted and authenticated using the shared session keys in all of our protocols. We add nonces and end point identifiers

to each message to prevent message replay. The protocol proceeds as follows:

(**1**) The ASE triggers a counter increment using the RE.

(**2**) The RE increments a counter for the ASE, increases its own MC, and signs the MC using $SK_{RE}$. The counter is signed to preserve its integrity in the case of compromised assisting REs.

(**3**) The RE sends the signed counter to all REs in the protection group.

(**4**) Upon receiving the signed MC, each RE updates its group counter table. The table is kept in the runtime memory, and not sealed after every update, to avoid endless propagation.

(**5**) The REs that received the counter send an *echo* message that contains the received signed MC. The REs also save the *echo* in runtime memory for later comparison.

(**6**) After receiving a quorum $q = u + f + 1 = \frac{n+f+1}{2}$ *echos*, the RE returns the *echos* to their senders.[4] The second round of communication is needed to prevent attacks based on RE restarts during the update protocol.

(**7**) Upon receiving back the *echo*, each RE finds the self-sent *echo* in its memory and checks if the MC value from it matches the one in the group counter table and the one received from the target RE. If this is the case, the RE replies with a final ACK message.

(**8**) After receiving $q$ final ACKs, the RE seals its own state together with the MC value to the disk.

(**9**) The RE returns the incremented ASE counter value. The ASE can now safely perform the state update (e.g., update account balance), save the counter value to its runtime memory for later comparison, and seal its state with the counter.

## 4.5 RE Restart Protocol

Figure 5 shows the protocol that the RE runs after a restart. The goal of the protocol is to allow the RE to join the existing protection group, retrieve its counter value and the MC values of the other nodes.

At restart the RE loses all previously established session keys and has to establish new session keys. In order to preserve our security guarantees, the target RE waits until it establishes new session keys with all other REs residing in the protection group. All assisting REs update their group configuration tables accordingly. The session key refresh mechanism prevents nodes from communicating with multiple RE instances on one platform (see Section 5). Another condition for successfully joining

[4]It might seem that waiting for more than $q$ responses, and therefore allowing more than $q$ nodes to complete the protocol, would increase system robustness. However, the quorum is designed such that writing the latest counter to more than $q$ nodes does not help the system to proceed in case of node unavailability or restarts (see Section 5).

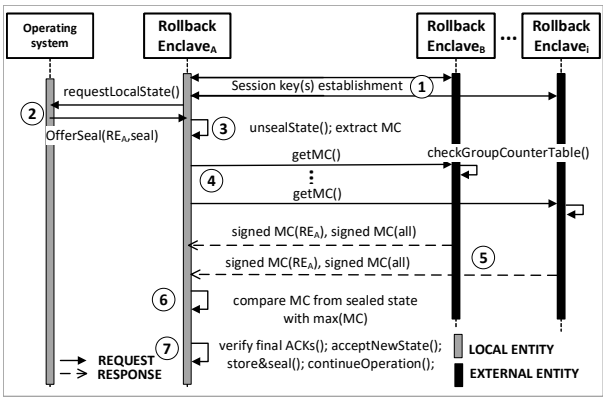



Figure 5: The RE restart protocol.

the protection group is that sufficiently many nodes return non-zero counter values (step 6 below). This check prevents simultaneously restarted REs from establishing a second, parallel protection group. This guarantee can be maintained when at most $u$ nodes restart simultaneously. The protocol proceeds as follows:

(**1**) Session key establishment with other nodes and update of the group configuration table.

(**2**) The RE queries the OS for the sealed state.

(**3**) The RE unseals the state (if received) and extracts the MC.

(**4**) The RE sends a request to all other REs in the protection group to retrieve its MC.

(**5**) The assisting REs check their group counter table. If the MC is found, the enclaves reply with the signed MC. Additionally, the complete table of other signed MCs that the responding node has in its memory is sent to the target RE.

(**6**) When the RE receives $q$ responses from the group (recall that $q = u + f + 1$ and $q \geq n/2$), it selects the maximum value and verifies the signature. We select the maximum value because some REs might have an old counter value or they may have purposefully sent one. The target RE verifies signatures and compares all the group counter table entries with received values for other nodes. For each assisting RE, the target RE picks the highest MC and updates its own group counter table with the value. The RE also verifies that at least $f + 1$ of the received counter values are not zero to prevent creation of the parallel network. If the obtained counter value matches the one in the unsealed data, the unsealed state can be accepted.

(**7**) The RE stores and seals the updated state. The RE will also save the counter value to its runtime memory.

The RE now has an updated group counter table that reflects the latest counters for each node in the group.

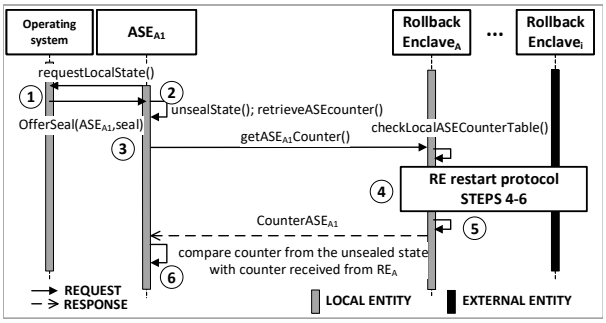



Figure 6: The ASE start/read protocol.

## 4.6 ASE Start/Read Protocol

When an ASE needs to verify the freshness of its state, it performs the protocol shown in Figure 6. This is needed to verify the freshness of unsealed state after an ASE restart or when an ASE replies to a client request asking its current state (e.g., account balance). The ASE must verify that another ASE instance does not have a newer state. The protocol proceeds as follows:

(**1**) The ASE queries the OS for the sealed data.

(**2**) The ASE unseals the state (if received) and obtains a counter value from it.

(**3**) The ASE issues a request to the local RE to retrieve its latest ASE counter value.

(**4**) To verify the freshness of its runtime state, the RE performs the steps 4-6 from the RE Restart protocol, to obtain the latest MC from the network. This is needed to prevent forking attacks with multiple RE instances. If the obtained MC does not match the MC residing in the memory, the state of the RE is not the latest, so, the RE must abort and be restarted. This is an indication that another instance of the same RE was running and updated the state in the meantime. If the values match, the current data is fresh and the RE can continue normal operation.

(**5**) If all verification checks are successful, the RE returns a value from the local ASE counter table.

(**6**) The ASE compares the received counter value to the one obtained from the sealed data.

If the counters match, ASE loads the previously sealed state or completes a security-critical client request.

## 4.7 Group Management

**Group updates.** The group owner issues a signed list of public parts of the public-private key pairs generated by each Rollback Enclave that define the protection group. Assume that later one or more processors in the group are found compromised or need replacement. The group owner should be able to update the previously established group (i.e., exclude or add new nodes) without interrupting the system operation.

During system initialization, the RE verifies the signed list of group member keys and seals the group configuration. When a group update is needed, the group owner issues an updated list that will be processed and sealed by the RE. This approach does not require the entry of the secret initialization key such as in first group establishment. However, the adversary should not be able to revert the group to its previous configuration (e.g., one including compromised nodes) by re-playing the previous group configuration. Since group updates are typically infrequent, they can be protected using SGX or TPM counters.

At system initialization, the RE creates a monotonic counter using SGX counter service or on a local TPM. If this is done using TPM, establishing a shared secret with the TPM (see session authorization in [23]) is necessary. The group owner includes a version number to every issued group configuration. When the RE processes the signed list, it increments the SGX or TPM counter to match the group version, and includes the version number in the sealed data. For every group update, the RE increments either of these counters. When the RE is restarted, it verifies that the version number in the unsealed group configuration matches the counter. The NVRAM memory in TPMs is expected to support approximately 100K write cycles, while with SGX counters support approximately 1M cycles, sufficient for most group management needs. For example, if group updates are issued once a week, the NVRAM would last 2000 years using TPMs and 20000 year using SGX counters.

**Group setup with attestation.** In Section 4.3 we described group setup for pre-defined platforms. The drawback of this approach is that it requires trusted operating systems at initialization. If the application scenario allows group establishment among *any* SGX platforms, similar trust assumption is not needed. The group owner can attest $n+1$ group members using the attestation mode that returns a pseudonym for each attested platform, establish secure channels to all group members, and distribute keys that group members use to authenticate each other. Because each platform reports a different pseudonym, this process guarantees that the protection group consists of $n+1$ separate platforms in contrast to multiple instances on one compromised CPU.

# 5 Security Analysis

Our system is designed to provide the following security property: an ASE cannot be rolled back to a previous state. In Section 5.1 we first show that given a secure storage functionality, as defined in Section 3, an RE can verify that its state is the latest. After that, in Section 5.2, we show that the participating REs realize the secure counter storage as a distributed system. Finally, by putting these two together, we show that ASEs cannot be rolled back if the RE cannot be rolled back.

Our system achieves a security guarantee that we call

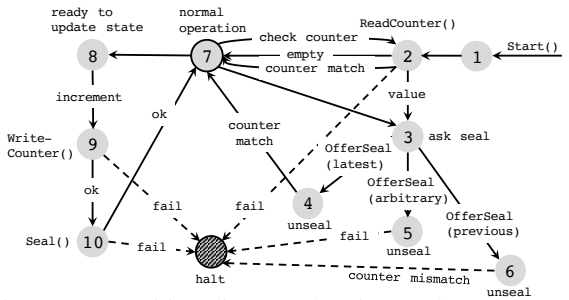



Figure 7: Transition diagram showing enclave *execution states* using an ideal secure counter storage functionality.

*all-or-nothing rollback*. The only way to violate enclave data integrity is to reset all nodes which brings the entire group to its initial state. In many application scenarios such integrity violation is easy to detect, and we do not consider it an attack on ROTE.

In the event of crashes, restarts or node unavailability, the system may fail to proceed temporarily or permanently. We distinguish three such cases: *Halt-1* where the system may be able to proceed automatically by simply trying again later (e.g., temporary network issue); *Halt-2* where manual intervention from the system administrator is needed (e.g., faulty node that needs to be fixed); and *Halt-X* where the complete system has to be re-initialized and the latest state of enclaves will be lost (e.g., simultaneous crash of all nodes). Recall that as the adversary controls the OS on all nodes, denial of service is always possible.

## 5.1 Protection with Secure Storage

Given the secure counter storage functionality (see Section 3) rollback can be prevented using the inc-then-store technique. In Figure 7 we illustrate a state transition diagram that represents RE states during sealing, unsealing and memory reading using the secure storage functionality. The notion of state in this section is an *execution state*, in contrast to enclave *data states* created and stored using sealing. We show that any combination of adversary operations, in any of the enclave execution states, cannot force the RE to accept a previous version of sealed data. We also show that in spite of multiple local RE instances, the read enclave state is always the latest. Note that this state transition diagram does not capture system initialization.

**First start.** After creating and starting the enclave using $e \leftarrow$ `Create(code)` and $i \leftarrow$ `Start(e)`, the RE execution begins from State 1. The MC is set to zero in the runtime memory and RE proceeds to State 2. The RE reads the counter value from the secure storage using `ReadCounter()`. If the `ReadCounter()` operation fails, the RE halts (Halt-1). On the first execution the operation returns *empty* and the RE continues to State 7 to continue normal operation. From State 7 the execution moves to State 2 for verifying freshness if a `Read()` request is received, while the `Write()` request moves execution to State 8.

**Sealing.** When the RE needs to seal data for local persistent storage, it proceeds to State 8. The RE increments the MC, and performs the `WriteCounter()` operation to the secure storage in State 9. The RE continues to State 10 if the operation succeeds, otherwise it halts (Halt-1). In State 10, the RE seals data ($s \leftarrow$ `Seal(data)`) of its current state along with the counter value. OS confirmation moves the enclave to normal operation in State 7. If sealing fails, the node can try again (Halt-1). If that does not help, the node loses its latest state and becomes unavailable, and a group update is needed (Halt-2).

**Unsealing.** When the RE needs to unseal data (recover its state), the RE proceeds from State 7 to State 3. The adversary can offer the correct sealed data (`OfferSeal(`$latest \equiv s$`)`) which moves the execution to State 4. Unsealing is successful and the counter value in the seal matches the MC value in the runtime memory, bringing the RE back to State 7. The adversary can offer a previously sealed state (`OfferSeal(`*previous*`)`) which moves the execution to State 6. Unsealing is successful, but counter values do not match and the RE halts (Halt-1 or Halt-2).[5] Finally, the adversary can offer any other data (`OfferSeal(`*arbitrary*`)`) which moves the RE to State 5 where unsealing fails and RE halts (Halt-1 or Halt-2).

**Forking.** If a new instance of the RE is started, the execution for it moves to State 1 following *First start*. Other instances remain in their original states. If for every `Write()` and `Read()` operation a counter is incremented or respectively retrieved from the secure counter storage to verify freshness, no rollback is possible. When the RE needs to read its runtime state (e.g., to complete a client request), the RE proceeds from State 7 to State 2. The RE reads the MC from the secure counter storage (if this fails, Halt-1) and compares the value to the one residing in its memory. This check is needed to guarantee that another instance of the same enclave does not have a newer state. If comparison succeeds, RE has the latest internal memory state and proceeds back to State 7. If the comparison fails (retrieved MC is higher), the RE moves to State 3 to obtain the latest seal (see above).

**Restart.** After an RE restart, the execution proceeds to State 2. If the `ReadCounter()` operation returns a non-empty value, the RE proceeds to State 3, otherwise to State 7, from where we follow the same steps as above. If the counter read operation fails, RE enters Halt-1.

If in any of these states the RE is terminated or restarted, its execution continues from State 1. Deleting

[5] If the OS provides an incorrect sealed data, most likely it is faulty an needs to be fixed. From some OS errors it may be possible to recover by simply trying again.

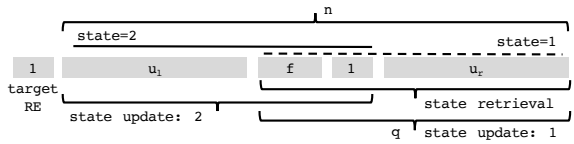



Figure 8: Network partitioning example where the adversary intentionally blocks a part of the nodes.

and creating the same enclave again has the same effect. `Suspend()` and `Resume()` have no effect, i.e., the enclave remains in the same execution state. We conclude that, assuming the secure storage functionality, the adversary cannot rollback the state of the RE.

## 5.2 Distributed Secure Storage Realization

Next, we show how ROTE realizes the secure counter storage functionality as a distributed system. When obtaining a counter from the distributed protection group (`ReadCounter`), RE receives the latest value that was sent to the protection group (`WriteCounter`). We divide the analysis into four parts: quorum size, platform resets, two-phase counter writing, and forking attacks.

**Quorum size.** The ROTE system has three parameters: the number of assisting nodes $n$, compromised nodes $f$, and unresponsive nodes $u$. The required quorum for responses at the time of counter writing and reading is $q = f + u + 1 = \frac{n+f+1}{2}$. Figure 8 illustrates that this is an optimal quorum size. We consider an example where the adversary performs network partitioning by blocking messages during writing and reading.

On the first write, the attacker allows the counter value 1 to reach the right side of the group by blocking the messages sent to the left side. On the second write, the adversary allows the counter value 2 to reach the left side of the group by blocking the right side. Finally, on counter read, the adversary blocks the left side again. If the counter is successfully written to $q = f + u + 1$ nodes, there always exists at least $u + 1$ honest platforms in the group that have the latest counter value in the memory. Because counter reading requires the same number of responses, at least one correct counter value is obtained upon reading. The maximum number of tolerated compromised platforms is $f = n - 1$, if $u = 0$ and $q = n$. If the quorum cannot be satisfied in either the state update protocol or any counter retrieval, the system enters Halt-1 and can try to perform the same operation again.

**Platform restarts.** If an assisting RE is restarted, it needs to first establish session keys and then recover the lost MC values from the protection group. Session key establishment procedure is explained below under *Forking attacks*; the main take-away is that up to $u$ nodes may restart simultaneously and after the nodes are online again the RE needs to establish session keys with *every* node in the group before proceeding with MC recovery.[6] Once the keys are established, some assisting nodes can be inactive or restarted. Three distinct cases are possible. First, the number of inactive/restarted REs is at most $u$. Since the number of running nodes is $u + f + 1 = q$ there are sufficient available platforms with the correct MC for the counter retrieval. Second, more than $u$ platforms, but not the entire protection group, are restarted. The number of remaining platforms is insufficient for RE recovery and the distributed system no longer provides successful MC access, but no rollback is possible (Halt-X, since there is no guarantee that the non-restarted nodes have the latest counter, thereby risking a rollback. However, before re-initializing the system, the latest states from the non-restarted nodes can be manually saved.) Third, all $n + 1$ nodes are restarted at the same time, in which case a new system configuration has to be deployed again by the group owner to re-initialize the system (Halt-X).

**Two-round counter writing.** Additionally, it remains to be shown how our update protocol successfully writes the counter to $q$ nodes, despite possible RE restarts *during* the protocol. We illustrate the challenges of counter writing through an example attack on a single-round variant of the update protocol that completes after the RE has received $q$ echoes. During state update the adversary blocks all communication and performs sequential message passing. First, the attacker allows message delivery to only one node that saves the counter and returns an echo. After that, the attacker restarts the RE on that node, which initiates the recovery procedure from the rest of the protection group. The adversary blocks the communication to the target platform, and the restarted RE recovers the previous counter value, because other reachable REs have not yet received the new value. The adversary repeats the same process for all platforms. As a result, the target node has received $q$ echos and accepts the state update, but all the assisting nodes have the previous counter value. Rollback is possible.

The second communication round in our protocol prevents such attacks. No combination of RE restarts during the state update protocol allows the target RE to complete it, unless the counter was written to $q$ nodes. There are four distinct cases to consider. Below, we assume that the adversary restarts at most $u$ platforms simultaneously. If more are restarted, recovery is not possible (Halt-X).

- *Case 1: Echo blocking.* If the attacker blocks communication or restarts assisting REs so that $q$ nodes cannot send the echo, the protocol does not complete (Halt-1).

[6] Consider an example, where two nodes are restarted at the same time. The first node wakes up and attempts to establish new session keys with all assisting nodes. This node has to wait, until the second restarted node wakes up and can communicate. After this point, both of the restarted nodes can establish session keys (with all nodes) and proceed with the RE Restart protocol.

• *Case 2: No echo blocking.* If the attacker allows at least $q$ echoes to pass, RE starts returning them and we have two cases to observe:

• *Case 2a: No restarts during first round.* If none of the assisting REs were restarted during the first protocol round, then at least $u+1$ nodes have the updated MC. If the adversary restarts assisting REs before they sent the final ACK and after they received the self-sent *echo* back from the target RE, the protocol will not complete (Halt-1), because fewer than $q$ final ACKs will be received. The protocol run may be repeated again. The adversary can also restart assisting REs after they have sent the final ACK which will result in successful state update, and successful state recovery of the restarted REs since a sufficient number of the assisting nodes already have the updated counter value.

• *Case 2b: Restarts during first round.* If the adversary restarts assisting REs during the first round, the update protocol will either successfully complete ($q$ final ACKs received) or halt execution (Halt-1) depending on the number of simultaneously restarted nodes. Sequential node restarts, as discussed in the example attack above, are detected. Upon receiving $q$ echoes, the RE sends each of the received echoes to the original sender. Because of sequential RE restarts, all assisting nodes have the previous MC value in their runtime memory, and thus the protocol will fail upon comparison of the echoes and the MC values. None of the assisting REs will deliver the final ACK, and the protocol will not complete (Halt-1).

We conclude that the successful completion of the two-phase state update protocol guarantees that at least $q$ nodes received and at least $u+1$ honest nodes have (i.e., correctly stored) the correct MC.

**Forking attacks.** Our system prevents attacks based on multiple enclave instances by requiring that the ASE start/read and RE restart protocols contact the assisting nodes and verify the latest counter from the protection group. If the latest counter is correct, RE can be certain that it made the last update. If the session's keys are outdated, communication with other nodes is disabled and RE knows another instance has run in parallel.

The session key refresh mechanism allows us to uniquely identify the latest running instance and prevents parallel communication with two instances running on one platform. After every RE start, keys have to be established *with all nodes* from the protection group to prevent the attacker from instantiating new REs on different platforms in a one-by-one manner while keeping some of the nodes disconnected. Other nodes delete the old session key that they shared with the previous instance residing on the same platform, rendering its communication unusable. The protection group only allows keys for one running instance on each platform. Also, by forcing state retrieval and freshness verification after each instantiation and for all ASE requests, the running instance on each platform will always have the latest state and highest MC, thus preventing rollback.

Our system also ensures that the adversary cannot establish a parallel protection group on the same platforms and re-direct ASEs to the rogue system causing a rollback. If no initialization key is provided and the RE receives all zero MC values from others in the group during setup, it will abort execution. A new network may only be created under the supervision of the group owner with the correct initialization key.

**Summary.** If the target RE has the latest MC that it sent, it is able to distinguish its latest sealed state, and if the latest sealed state is loaded, all the ASEs state counters kept within are fresh. Upon retrieval, the ASE always receives the latest counter, and thus each ASEs can verify that it has the latest state data. If the target RE is not able to recover the latest MC, the system end ups in either Halt-1, Halt-2 or Halt-X.

## 6 Performance Evaluation

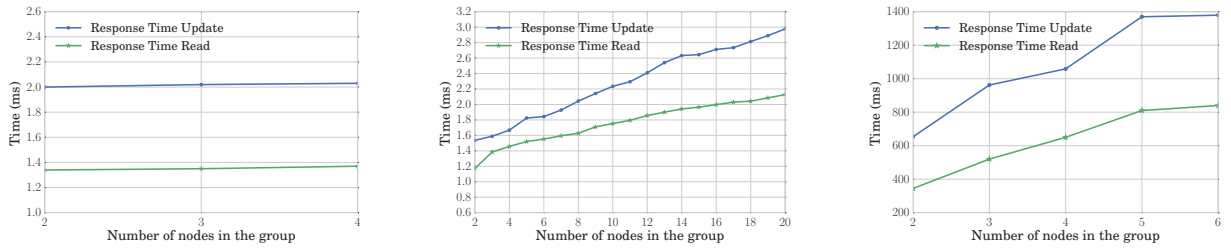
In this section we describe our performance evaluation. First, we describe our implementation that consists of the following components. We implemented the RE (950 LoC), an accompanying *rollback relay* application (1600 LoC), ROTE library (150 LoC), a simple test ASE (100 LoC), and a matching *test relay* application (100 LoC). The purpose of the relays is to mediate enclave-to-enclave communication. We implemented all components in C++, the relays were implemented for the Windows platform. The local communication between the relay applications was implemented using Windows named pipes. The total TCB accounts for 1100 LoC.

The enclaves use asymmetric cryptography for signing (ECDSA) and encryption (256-bit ECC). Our implementation establishes shared keys using authenticated Diffie-Hellman key exchange. For symmetric message encryption and authentication we use 128-bit AES-GCM in encrypt-then-MAC mode. All used cryptographic primitives are provided by the standard Intel SGX libraries.

### 6.1 State Update and Read Delay

The main performance metrics that we measure are the ASE state update and state read delays that include the counter writing to and reading from the protection group. The delays depends on the network characteristics and the size of the protection group ($n+1$). The RE restart operation is typically performed once per platform boot, and thus the operation is not similarly time-critical so we do not measure it. In all test cases we set $u = f = 0$, as their values do not affect state update and read delays.[7]

[7]The state update protocol proceeds immediately after receiving $q$ responses, and therefore node unavailability does not affect update de-




(a) 1st exp. setup (ROTE implementation) (b) 2nd exp. setup (local group, simulated) (c) 3rd exp. setup (global group, simulated)

Figure 9: Experimental results, state update/read delay. The first figure shows ROTE performance for protection groups that are connected over a local network, the second figure shows the simulated performance for a larger group also over a local network, while the third figure is for geographically distributed protection groups.

**Experimental setup.** Our first experimental setup consisted of four SGX laptops and our second experimental setup consisted of 20 identical desktop computers, both connected via local network (1Gbps, ping $\leq 1ms$). Our third experimental setup was a geographically distributed (in order, US (West), Europe, Asia, S. America, Australia, US (East)) protection group of sizes from two to six nodes that we tested on Amazon AWS EC2. For the first setup we used the real ROTE implementation while the latter two we used a simulated implementation (the same protocol, but no enclaves).

**Results.** The state update delay consists of two components: networking and processing overhead. Context switching to enclave execution is fast (few microseconds). Symmetric encryption used in the protocol is also fast (less than a microsecond). The only computationally expensive operation that we use is asymmetric signatures (0.46 ms per signing operation). We provide more performance benchmarks in [20].

The ASE state update protocol has one signature creation which is verified later in the RE and ASE start/read protocols. The required processing time of the state update protocol is less than 0.6 ms, where the creation of the first protocol message takes 0.51 ms (signing). The state read protocol requires one round trip, while the state update protocol needs two. All messages passed between the nodes are 224 bytes (200 payload + 24 header).

Figure 9 shows the results from our three experimental setups. Figure 9a shows that the state update delay was approximately 2 ms, while the state read delay was approximately 1.3 ms for group sizes from two to four nodes using the ROTE implementation. Figure 9b illustrates an increase in the delay as the group size grows. This is as expected, since the target platform needs to communicate with more platforms. For group size of 20 nodes, the delay is 2.98 ms and 2.13 ms, respectively. Lastly, Figure 9c illustrates a less systematic increase in delay, due to the dependency on network connections between various geographic locations in the protection group. The update time between two locations takes 654 ms while between five the update time is 1.37 seconds. The read delay is respectively 342 ms and 810 ms.

We draw two conclusions from these experiments. First, the performance overhead imposed by ROTE is defined largely by the network connections between the nodes. Second, if the nodes are connected over a low-delay network, ROTE supports applications requiring very fast state updates (1-2 ms). For applications tolerating larger delays (e.g., more than 600 ms per state update), ROTE can be run on geographically distant groups.

| Request type | State size (KB) | no rollback protection (ms) | ROTE system (ms) | SGX counter protection (ms) |
|---|---|---|---|---|
| Write state | 1 | 3.85 ($\pm$ 0.06) | 5.17 ($\pm$ 0.03) | 160.7 ($\pm$ 0.7) |
| | 10 | 4.65 ($\pm$ 0.05) | 6.03 ($\pm$ 0.03) | 162.7 ($\pm$ 1.6) |
| | 100 | 6.49 ($\pm$ 0.04) | 7.83 ($\pm$ 0.05) | 169.1 ($\pm$ 2.1) |
| Read state | 1 | 0.06 ($\pm$ 0.00) | 1.41 ($\pm$ 0.02) | 61.04 ($\pm$ 3.1) |
| | 10 | 0.19 ($\pm$ 0.00) | 1.53 ($\pm$ 0.01) | 61.17 ($\pm$ 3.1) |
| | 100 | 1.76 ($\pm$ 0.05) | 3.1 ($\pm$ 0.02) | 62.74 ($\pm$ 3.2) |

Table 1: Example application throughput without rollback protection, using ROTE and using SGX counters.

## 6.2 Example Application Throughput

Additionally, we measured the throughput of an example financial enclave that processes incoming transactions repeatedly (the transaction buffer is never empty). We tested the enclave using (a) no rollback protection, (b) the ROTE implementation, and (c) SGX counter based rollback protection. The experimental setup was a protection group of four nodes. For every update transaction, the enclave updates its state, creates a new seal, and writes it to the disk, while the read transaction includes reading from the disk, unsealing and retrieving the counter for comparison. In case of ROTE and SGX counter variants, the enclave also performs a counter increment. We tested three different enclave state sizes (1 KB, 10 KB, 100 KB) since the state size for transactions can differ based on the exact use case.

lay. Similarly, up to $f$ compromised nodes can discard counter values or return fake values, but that does not affect the protocol delay.

**Results.** Table 1 shows our results. In all three cases the ROTE system provides significantly better state update performance than using SGX counters (e.g., 190 over 6 tx/s for 1KB) while suffering a 20-25% performance drop in comparison to systems which have no rollback protection (e.g., 260 over 190 tx/s for 1KB). We conclude that our system provides significantly faster rollback protection than methods based on local non-volatile memory. Compared to systems with no rollback protection, our solution imposes a moderate overhead.

## 7 Discussion

**Data migration.** Although sealing binds encrypted enclave data to a specific processor, our solution enables data migration within the protection group. Migration is especially useful before planned hardware replacements and group updates (e.g., node removal). In a migration operation, an ASE first unseals its persistent data and passes it to the RE. The RE sends the enclave data to another Rollback Enclave within the same protection group together with the measurement of the ASE. The communication channel between the REs is encrypted and authenticated. On the receiving processor, the RE passes the enclave data to an instance of the same ASE (based on attestation using the received measurement) which can seal it. Note that the RE is agnostic to the internal state of ASEs and just re-encrypts data it receives from an ASE without the need to understand its semantics. Combined with group updates (Section 4.7), such enclave data migration enables flexible management of available computing resources. Similar data migration is discussed in [25].

**Information leakage.** Our model excludes execution side-channels. Here we briefly discuss additional information leakage that our solution may add. Each enclave state update and read causes network communication. An adversary that can observe the network, but does not have access to the local persistent storage, can use the information leakage to determine the timing of sealing and unsealing events. Also the reboot of the target platform causes an observable network pattern. We consider such information leakage a practical concern but developing countermeasures is outside the scope of this paper.

**Performance.** The main performance characteristic of our solution, the state update delay, is dominated by the networking and the asymmetric signature operation required for the first message of the state update protocol. In case of a local, 1Gpbs, network and an average laptop, the networking takes approximately 1 ms and the signature operation 0.5 ms. A possible optimization is to pre-compute the asymmetric signatures. Since the signed data is predictable MC values, we can pre-compute and store them. This pre-computation may be done at times when the expected load is low or at system initialization depending on the specific scenario.

For communication between the enclaves we use symmetric keys derived from the key agreement protocol for performance reasons, since it is computationally much less expensive. However, depending on the application scenario we could use asymmetric keys which would enable, for example, post-incident forensics. This design choice is dependent on the use case and performance requirements. ROTE can accommodate both approaches.

**Consensus applications.** In the specific case where all participating enclaves implement a distributed application with the purpose to maintain a consensus (e.g., permissioned blockchain), our rollback protection can be optimized further. In such an application, all participating enclaves have a shared, global state and the state update protocol can be replaced with a suitable Byzantine agreement protocol. When an enclave is restarted (or determines its latest state), it queries its latest state from the participating enclaves similar to our RE restart protocol. We leave a detailed design as future work.

**Forking prevention.** The current SGX architecture does not provide the ability for one enclave instance to check if another instance of the same enclave is already running. The implementation of this feature would simplify rollback protection significantly.

Forking prevention could be implemented using a TPM. After system boot, the RE instance could extend a PCR that has a known value at boot. If a second RE instance is started, it can check if the PCR value differs from its known initial value [2]. The drawback of this approach is the increase of the system security perimeter outside of the processor.

**Periodic check-pointing.** For increased robustness, our rollback protection can be complemented with periodic check-pointing. An example approach would be to increment a counter on local NVRAM on selected updates (e.g., mod 100). If all nodes crash at the same time, the administrator has an option to recover from the latest saved checkpoint with the risk of possible rollback.

## 8 Related work

**SGX-counter and TPM solutions.** Ariadne [2] uses TPM NVRAM or SGX counters for enclave rollback protection. The counter is incremented using store-then-inc that provides crash resilience, but allows two executions of the latest input. Ariadne minimizes the TPM NVRAM wear using counter increments that flip only a single bit. Compared to our solution, SGX counters are an optional feature, increments are slow and make the non-volatile memory unusable after few days of continuous use. Similar performance limitations apply to TPM NVRAM. SGX counters are also likely vulnerable to bus tapping and flash mirroring attacks [22], while in our solution the trust perimeter is the processor package.

Memoir [4] also leverages TPM NVRAM for rollback protection, and therefore has similar performance limitations. An optimized variant of Memoir assumes the availability of an Uninterrupted Power Supply (UPS). This variant stores the state updates to volatile Platform Configuration Registers (PCRs) and at system shutdown writes the recorded update history to the NVRAM. ICE [3] enhances the CPU with protected volatile memory, a power supply and a capacitor that at system shutdown the flushes the latest state to non-volatile memory. Both the optimized Memoir and ICE require hardware changes. Additionally, reliably flushing data upon a crash or power outage can be challenging in practice.

**Client-side detection.** Brandenburger [26] proposes client-side rollback detection for SGX in the context of cloud computing. The main difference to our work is that this approach does not prevent a rollback directly on the server. Instead, it allows mutually trusting clients to remain synchronized, and given that certain connectivity requirements are met, detect consistency and integrity violations (including rollback) after the incident.

**Integrity servers.** Verena [6] maintains authenticated data structures for web applications and stores integrity information on a separate, trusted server. Another use case is to prevent the usage of disabled credentials on mobile devices by storing counters on an integrity-protected server [8]. In such solutions the integrity server becomes a single point of failure.

**Byzantine broadcast and agreement.** Our state update protocol follows the approach of Echo broadcast [15] with an additional confirmation message in the end. Like other byzantine broadcast primitives, our state update protocol requires $O(n)$ messages. Byzantine agreement typically require $O(n^2)$ messages. Byzantine broadcast and agreement protocol operate on arbitrary values and assume a potentially malicious sender. Thus, such protocols require $3f+1$ replicas. In our system the target enclave is trusted and the distributed data is a signed counter value. Thus, $f+1$ replicas are sufficient.

**Secure audit logs.** Secure audit log systems [27, 28, 29, 30] provide accountability and in particular prevent manipulation of previous log entries *after* the target platform becomes compromised. Most such audit log systems assume a trusted but infrequently accessible storage. Our goal is to design a system that has no single point of failure, and therefore in ROTE the trusted storage is realized as a distributed system amongst a set of assisting nodes (some of which can be compromised).

**Accountability for distributed systems.** PeerReview [31] provides accountability for distributed systems and in particular detect nodes that violate from expected behaviour. Instead of fault detection, our goal is to realize distributed secure storage, customized for rollback protection, in the presence of faulty nodes.

**Adversary models.** Agreement has been considered under models where the faulty nodes have some trusted functionality (e.g., an unmodifiable hardware primitive). Such approaches reduce the number of required replicas to $2f+1$ [32, 33, 34, 35] or $f+1$ [36]. We have no trust assumptions on the compromised nodes. Byzantine agreement has also been considered with dual failure models [37, 38, 39] where the adversary can fully control the faulty processes and can read the secrets of other processes. In our case, the adversary cannot read secrets from trusted enclaves, but it can extract keys from $f$ compromised nodes, and additionally schedule enclaves' execution on all nodes.

Several recently proposed SGX systems [40, 41, 42, 13, 43, 44, 45, 46] consider an adversary model with an untrusted OS. To the best of our knowledge, our work is the first to define a model with explicit adversarial capabilities that cover enclave restarts and multiple instances. These capabilities are critical for the security of our system and also other SGX systems (see the extended version of this paper for details [20]).

## 9 Conclusion

In this paper we have proposed a new approach for rollback protection on Intel SGX. Our main idea is to implement integrity protection as a distributed system across collaborative enclaves running on separate processors. We consider a powerful adversary that controls the OS on all participating platforms and has even compromised a subset of the assisting processors. We show that our system provides a strong security guarantee that we call *all-or-nothing rollback*. Our experiments demonstrate that distributed rollback protection provides significantly better performance compared to solutions based on local non-volatile memory.

## Acknowledgements

This work was partly supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme. We would like to thank Jonathan McCune for his insightful comments.

## References

[1] V. Costan *et al.*, "Intel SGX explained," in *Cryptology ePrint Archive*, 2016.

[2] R. Strackx *et al.*, "Ariadne: A minimal approach to state continuity," in *USENIX Security*, 2016.

[3] ——, "ICE: A passive, high-speed, state-continuity scheme," in *ACSAC*, 2014.

[4] B. Parno *et al.*, "Memoir: Practical state continuity for protected modules," in *IEEE S&P*, 2011.

[5] Intel, "SGX documentation: sgx_create_monotonic_counter," 2016, https://software.intel.com/en-us/node/696638.

[6] N. Karapanos *et al.*, "Verena: End-to-End Integrity Protection for Web Applications," in *IEEE S&P*, 2016.

[7] M. van Dijk *et al.*, "Offline Untrusted Storage with Immediate Detection of Forking and Replay Attacks," in *ACM STC*, 2007.

[8] K. Kostiainen *et al.*, "Credential Disabling from Trusted Execution Environments," in *Nordsec*, 2010.

[9] M. Castro *et al.*, "Practical Byzantine fault tolerance," in *OSDI*, 1999.

[10] D. Dolev *et al.*, "On the security of public key protocols," *IEEE Transactions on information theory*, 1983.

[11] M. Pease *et al.*, "Reaching agreement in the presence of faults," *Journal of the ACM*, 1980.

[12] L. Lamport *et al.*, "The Byzantine Generals Problem," *ACM TOPLAS*, 1982.

[13] M.-W. Shih *et al.*, "S-NFV: Securing NFV states by using SGX," in *ACM SDN-NFV*, 2016.

[14] F. Schuster *et al.*, "VC3: trustworthy Data Analytics in the Cloud Using SGX," in *IEEE S&P*, 2015.

[15] M. K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in Rampart," in *ACM CCS*, 1994.

[16] C. Cachin *et al.*, *Introduction to reliable and secure distributed programming*. Springer, 2011.

[17] Y. Xu *et al.*, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *IEEE S&P*, 2015.

[18] F. Brasser *et al.*, "Software Grand Exposure: SGX Cache Attacks are Practical," 2017, http://arxiv.org/abs/1702.07521.

[19] M. Schwarz *et al.*, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," 2017, http://arxiv.org/abs/1702.08719.

[20] S. Matetic *et al.*, "Rote: Rollback protection for trusted execution," 2017, https://eprint.iacr.org/2017/048.

[21] Intel Support Forum, "Ensuring only a single instance of Enclave," 2017, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/709552.

[22] S. Skorobogatov, "The bumpy road towards iPhone 5c NAND mirroring," 2016, http://arxiv.org/abs/1609.04327.

[23] Trusted Computing Group, "Trusted Platform Module Library, Part 1: Architecture, Family 2.0," 2014.

[24] Intel, "SGX documentation: sgx_get_trusted_time," 2016, https://software.intel.com/en-us/node/696636.

[25] R. Strackx *et al.*, "Idea: State-continuous transfer of state in protected-module architectures," in *ESSoS*, 2015.

[26] M. Brandenburger *et al.*, "Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory," 2017, http://arxiv.org/abs/1701.00981.

[27] B. Schneier *et al.*, "Secure audit logs to support computer forensics," *ACM TISSEC*, 1999.

[28] D. Ma *et al.*, "A new approach to secure logging," *ACM TOS*, 2008.

[29] S. A. Crosby *et al.*, "Efficient data structures for tamper-evident logging," in *USENIX Security*, 2009.

[30] A. Sinha *et al.*, "Continuous tamper-proof logging using tpm 2.0," in *TRUST*, 2014.

[31] A. Haeberlen *et al.*, "PeerReview: Practical Accountability for Distributed Systems," *ACM OSR*, 2007.

[32] B.-G. Chun *et al.*, "Attested append-only memory: Making adversaries stick to their word," in *ACM OSR*, 2007.

[33] D. Levin *et al.*, "TrInc: Small Trusted Hardware for Large Distributed Systems," in *NSDI*, 2009.

[34] M. Correia *et al.*, "How to tolerate half less one Byzantine nodes in practical distributed systems," in *DISC*, 2004.

[35] J. Liu *et al.*, "Scalable Byzantine Consensus via Hardware-assisted Secret Sharing," *arXiv preprint arXiv:1612.04997*, 2016.

[36] R. Kapitza *et al.*, "CheapBFT: resource-efficient byzantine fault tolerance," in *EuroSys*, 2012.

[37] F. J. Meyer *et al.*, "Consensus with dual failure modes," *IEEE TPDS*, 1991.

[38] J. A. Garay *et al.*, "A continuum of failure models for distributed computing," in *PDAA*, 1992.

[39] H.-S. Siu *et al.*, "A note on consensus on dual failure modes," *IEEE TPDS*, 1996.

[40] F. Tramer *et al.*, "Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge," 2016, http://eprint.iacr.org/2016/635.

[41] F. Zhang *et al.*, "Town Crier: An Authenticated Data Feed for Smart Contracts," in *CCS*, 2016.

[42] N. Weichbrodt *et al.*, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves," in *ESORICS*, 2016.

[43] D. Gupta *et al.*, "Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation," in *WAHC*, 2016.

[44] S. Brenner *et al.*, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Middleware*, 2016.

[45] R. Pass *et al.*, "Formal abstractions for attested execution secure processors," in *Cryptology ePrint Archive*, 2016.

[46] R. Sinha *et al.*, "Moat: Verifying Confidentiality of Enclave Programs," in *CCS*, 2015.

[47] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *HASP@ ISCA*, 2013.

[48] "Intel Software Guard Extensions, Reference Number: 332680-002," 2015, https://software.intel.com/sites/default/files/332680-002.pdf.

[49] S. Johnson *et al.*, "Intel SGX: EPID provisioning and attestation services," 2016, https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services.

[50] B. Alexander, "Introduction to Intel SGX Sealing," 2016, https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing.

[51] Intel, "Developer Zone Forums," 2016, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/607330.

[52] ——, "Intel 100 Series and Intel C230 Series Chipset Family Platform Controller Hub (PCH)," 2016, http://www.intel.com/content/www/us/en/chipsets/100-series-chipset-datasheet-vol-1.html.

[53] ——, "Intel 9 Series Chipset Family Platform Controller Hub (PCH)," 2015, http://www.intel.com/content/www/us/en/chipsets/9-series-chipset-pch-datasheet.html.

# A SGX Background

Here we briefly describe the main protection mechanisms of SGX. For a more elaborate explanation of the architecture, we refer interested readers to [1].

**Enclave creation.** An enclave is created by the system software. During enclave creation, the system software specifies the enclave code. Security mechanisms in the processors create a data structure called SGX Enclave Control Structure (SECS) that is stored in a protected memory area (see below). Because enclaves are created by the system software running on the OS, their code cannot contain sensitive data. The start of the enclave is recorded by the processor, reflecting the content

of the enclave code as well as the loading procedure (sequence of instructions). The recording of an enclave start is called *measurement* and it can be used for later attestation. Once an enclave is no longer needed, the OS can terminate it and thus erase its memory structure from the protected memory.

**Runtime isolation.** The SGX security architecture guarantees that enclaves are *isolated* from all software running outside of the enclave, including the OS, other enclaves, and peripherals. By isolation we mean that the control-flow integrity of the enclave is preserved and other software cannot observe its state. The isolation is achieved via protection mechanisms that are enforced by the processor. The code and data of an enclave are stored in a protected memory area called Enclave Page Cache (EPC) that resides in Processor Reserved Memory (PRM) [47]. PRM is a subset of DRAM that cannot be accessed by the OS, applications or direct memory accesses. The PRM protection is based on a series of memory access checks in the processor. Non-enclave software is only allowed to access memory regions outside the PRM range, while enclave code can access both non-PRM memory and the EPC pages owned by the enclave [1].

The untrusted OS can evict EPC pages into the untrusted DRAM and load these back at a later stage. While the evicted EPC pages are stored in the untrusted memory, SGX assures their confidentiality, integrity and freshness via cryptographic protections. The architecture includes the Memory Encryption Engine (MEE) which is a part of the processor uncore (microprocessor function close to but not integrated into the core [1]). The MEE encrypts and authenticates the enclave data that is evicted to the non-protected memory, and ensures enclave data freshness at runtime using counters and a Merkle-tree structure. The root of the tree structure is stored on the processor die. Additionally, the MEE is used to protect SGX's Enclave Page Cache against physical attacks and is connected to the Memory Controller [48, 1].

**Attestation.** Attestation is the process of verifying that certain enclave code has been properly initialized. In *local attestation* a prover enclave can request a statement that contains measurements of its initialization sequence, enclave code and the issuer key. Another enclave on the same platform can verify this statement using a shared key created by the processor. In *remote attestation* the verifier may reside on another platform. A system service called Quoting Enclave signs the local attestation statement for remote verification. The verifier checks the attestation signature with the help of an online attestation service that is run by Intel. Each verifier must obtain a key from Intel to authenticate to the attestation service. The signing key used by the Quoting Enclave is based on a group signature scheme called EPID (Enhanced Privacy ID) which supports two modes of attestation: fully anonymous and linkable attestation using pseudonyms [49, 1]. The pseudonyms remain invariant across reboot cycles (for the same verifier). Once an enclave has been attested, the verifier can establish a secure channel to it using an authenticated key exchange mechanism.

**Sealing.** Enclaves can save confidential data across executions. Sealing is the process to encrypt and authenticate enclave data for persistent storage [50]. All local persistent storage (e.g. disk) is controlled by the untrusted OS. For each enclave, the SGX architecture provides a sealing key that is private to the executing platform and the enclave. The sealing key is derived from a Fuse Key (unique to the platform, not known to Intel) and an Identity Key that can be either the Enclave Identity or Signing Identity. The Enclave Identity is a cryptographic hash of the enclave measurement and uniquely identifies the enclave. If data is sealed with Enclave Identity, it is only available to this particular enclave version. The Signing Identity is provided by an authority that signs the enclave prior to its distribution. Data sealed with Signing Identity can be shared among all enclave versions that have been signed with the same Signing Identity.

## B SGX Counter Analysis

Intel has recently added support for monotonic counters [5] as an optional SGX feature that an enclave developer may use for rollback attack protection. However, the security and performance properties of this mechanism are not well documented. Furthermore, they are not available on all platforms. In this Appendix we outline all executed experiments and evaluate the SGX counter and trusted time service.

**SGX counter service.** An enclave can query availability of counters from the Platform Service Enclave (PSE). If supported, the enclave can create up to 256 counters. The default owner policy encompasses that only enclaves with the same signing key may access the counter. Counter creation operation returns an identifier that is a combination of the Counter ID and a nonce to distinguish counters created by different entities. The enclave must store the counter identifier to access it later, as there is no API call to list existing counters. After a successful counter creation, an enclave can increment, read, and delete the counter.

According to the SGX API documentation [5], counter operations involve writing to a non-volatile memory. Repeated write operations can cause the memory to wear out, and thus the counter increment operations may be rate limited. Based on Intel developer forums [51], the counter service is provided by the Management Engine on the Platform Control Hub (PCH).

**Experiments.** We tested SGX counters on five different platforms: Dell Inspiron 13-7359, Dell Latitude

E5470, Lenovo P50, Intel NUC and Dell Optiplex 7040. The counter service was not available on Intel NUC. On Dell laptops a counter increment operation took approximately 250 ms, while on the Lenovo laptop and Dell Optiplex increment operations took approximately 140 ms and 80 ms, respectively. Strackx et al. [2] report 100 ms for counter updates. Counter read operations took 60-140 ms, depending on the platform. As expected, the counter values remained unchanged across enclave restarts and platform reboots. We tested the wear-out characteristics of the counters and found out that on both Dell laptops, after approximately 1.05 million writes, the tested counter became unusable and other counters on the same platform could not be created, incremented or read (all SGX counter operations return `SGX ERROR BUSY`).

Additionally, we observed that reinstalling the SGX Platform Software (PSW) or removing the BIOS battery deletes all counters. Finally, to our surprise, we noticed that after reinstalling the PSW, first usage of counter service triggered the platform software to connect to a server whose domain is registered to Intel. If Internet connection is not available, the counters are unavailable.

**Performance limitations.** An enclave developer could attempt to use SGX counters as a rollback protection mechanism. When an enclave needs to persistently store an updated state, it can increment a counter, include the counter value and identifier to the sealed data, and verify integrity of the stored data based on counter value at the time of unsealing. However, such approach may wear out the used non-volatile memory. Assuming a system that updates one of the enclaves on the same platform once every 250 ms, counters would become unusable in few days. Even with a modest update rate of one increment per minute, the counters are exhausted in two years. Services that need to process tens or hundreds of transactions per second are not possible.

**Weaker security model.** According to Intel developer forums [51], counter service is provided by the Management Engine on the PCH (known as "south bridge" in older architectures). However, to the best of our knowledge, actual location of the non-volatile memory used to store the counters is not publicly stated. Based on Intel specifications [52, 53], the PCH typically does not host non-volatile memory, but it is connected over an SPI bus to a flash memory that is also used by the BIOS. Since Management Engine is an active component, communication between the processor and the Management Engine can be replay protected. However, the SPI flash is a passive component, and therefore any counter stored there is likely to be vulnerable to bus tapping and flash mirroring attacks, as recently demonstrated in the case of mobile devices (inspired by FBI iPhone unlocking debate) [22]. Although the precise storage location of SGX counters remains unknown at the time of writing, it is clear that if the integrity of enclave data relies on the SGX counter feature, then additional hardware components besides the processor must be considered trusted. This is a significant shift from the enclave execution protection model, where the security perimeter is the processor package [48, p. 30].

**Other concerns.** The current design of SGX counter APIs makes safe programming difficult. To demonstrate this we outline a subtle rollback attack. Assume an enclave that at the beginning of its execution checks for the existence of sealed state, and if one is not provided by the OS, it creates a new state and counter, and stores the state sealed together with the counter value and identifier. The enclave increments the counter after every state update. Later, the OS no longer provides a sealed state to the restarted enclave. The enclave assumes that this is its first execution and creates a new (second) counter and new state. Recall that the SGX APIs do not allow checking existence of previous counter. The enclave updates its state again. Finally, the OS replays a previous sealed state associated with the first counter. A careful developer can detect such attacks by creating and deleting 256 counters (an operation that takes two minutes) to check if a previous counter, and thus sealed state, exists. A crash before counter deletion would render that particular enclave permanently unusable.

We have no good explanation why a connection to an Intel server is needed after the PSW reinstall. Similarly, we do not know why the SGX counters become unavailable after BIOS battery removal or PSW reinstall.

The above attack and availability issues probably could be fixed with better design of SGX APIs and system services, but the performance limitations and the weaker security model are hard to avoid in future versions of the SGX architecture.

**SGX trusted time.** Another recently introduced and optional SGX feature is the trusted time service [24]. As in the case of SGX counters, also the time service is provided by the Management Engine. The trusted time service allows an enclave developer to query a time stamp that is relative to a reference point. The function returns a nonce in addition to the timestamp, and according to the Intel documentation, the timestamp can be trusted as long as the nonce does not change [24].

We tested the time service and noticed that the provided nonce remained same across platform reboots. Reinstalling PSW resulted in a different nonce, but the provided time was still correct. The reference point is the standard Unix time. As a rollback protection mechanism the trusted time service is of limited use. Including a timestamp to each sealed data version allows an enclave to distinguish which out of two seals is more recent. However, the enclave cannot know if the sealed data provided by the OS is fresh and latest.