ROTE: Rollback Protection for Trusted Execution

Sinisa Matetic*, Mansoor Ahmed*, Kari Kostiainen*, Aritra Dhar*, David Sommer*, Arthur Gervais*, Ari Juels** and Srdjan Capkun*

*Institute of Information Security, ETH Zurich, {firstname.lastname}@inf.ethz.ch
**Cornell Tech, juels@cornell.edu

***Abstract*—Intel SGX isolates the runtime memory of protected applications (enclaves) from the OS and allows enclaves to encrypt and authenticate (seal) data for persistent storage. Sealing prevents an untrusted OS from reading or arbitrarily modifying stored data. However, rollback attacks, where the adversary replays an old seal, remain possible. Data integrity violations through rollback can have severe consequences, especially for enclaves that operate on financial data. The SGX architecture was recently updated to support monotonic counters that may be used for rollback prevention, but we show that these counters have significant performance and security limitations.**

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 among participating enclaves. We construct a model that captures the ability of the adversary to schedule the execution of protected applications, and show that our solution achieves a strong security property that we call *all-or-nothing rollback*: the only way to violate data integrity is to reset all participating platforms to their initial state. We implement ROTE and demonstrate that such a distributed rollback protection mechanism can be very fast.

I. INTRODUCTION

Intel Software Guard Extensions (SGX) enables execution of security-critical application code, called *enclaves*, in isolation from the untrusted system software [10]. 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* [41], [39], [30], [10] 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 enclave state integrity is to leverage an SGX feature that allows multiple instances of the same enclave to exist on one platform. The OS can create two enclave instances and route update request to one instance and read requests to the other. To remote clients that perform attestation, the two 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 transaction 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 leveraging the use of non-volatile memory [19]. 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. 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 [38]. Similar limitations apply also to rollback protection techniques that leverage Trusted Platform Modules (TPMs) [41], [30], [39].

The second common approach is to maintain integrity information for protected applications in a separate trusted server [23], [44], [24]. The drawback of such solutions is that they require the setup of additional computing infrastructure and the server becomes an obvious target for attacks. Server replication using standard Byzantine consensus protocols [7] 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 among multiple enclaves. When an enclave updates its state, it stores a counter to a set of enclaves running on assisting processors and saves its state data on persistent storage. Later, when the enclave needs to recover its state, it obtains counter values from assisting enclaves to verify that the stored 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 assume that the adversary can break SGX protections on some of the assisting processors, e.g., through physical attacks. The adversary can also control all network communication between the platforms. Our adversary model combines commonly considered network control based on the standard Dolev-Yao model [11] and Byzantine faults [32], [25], but additionally captures the the ability of the adversary to restart the execution of trusted processes from a previously saved state, dictate the execution schedule of trusted processes, and run multiple instances of the same trusted process on all platforms. Such adversarial capabilities are crucial for the security of our system, and we believe that the model is of general interest.

Secure and practical realization of distributed rollback protection under such a strong adversarial model involves 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. The adversary can also create multiple instances of the same enclave on all assisting platforms and route counter writes and reads to separate instances.

In this paper, we design and implement a rollback protection system called ROTE (Rollback Protection for Trusted Execution). Our system distributes counter values to a set of enclaves that we call the *protection group*. The size of the group depends on desired level of security (the number of tolerated compromised enclaves) and robustness (the number of platforms that may be simultaneously unreachable). The main components of our solution are a state update mechanism that is an optimized version of *consistent broadcast* protocols [33], [6], and a recovery mechanism that obtains lost counters from the rest of the protection group upon enclave restart. We design a session key update mechanism to address attacks based on multiple enclave instances.

We show that our solution achieves a strong security property that we call *all-or-nothing rollback* in the presence of a strong adversary. 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 individually any enclave to their respective previous states. The only way to violate data integrity is to reset the entire group to its initial state. Similar to [30], [41], our approach can 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 up to 20 platforms using a simulated implementation over a local network and geographically distributed protection groups. 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. Enclave developers can use our system through a simple API. The TCB size increment of ROTE 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.
- *SGX counter experiments.* We show that SGX counters have severe performance limitations.
- *Novel approach.* We propose a novel way to protect SGX enclaves. Our key idea is to realize rollback protection by storing enclave-specific counters in a distributed system encompassing a collaborative set of 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 small performance overhead. When deployed over a low-latency network, the state update overhead is 1-2 ms.

The rest of this paper is organized as follows. In Section II we provide background information on SGX. Section III defines our adversarial model and explains rollbacks attacks. Section IV describes our distributed rollback protection approach. Section V describes the ROTE system and Section VI provides security analysis. Section VII provides performance evaluation and Section VIII further discussion. We review related work in Section IX. Section X concludes the paper.

II. SGX BACKGROUND

The SGX architecture consists of new instructions, protective mechanisms in the processor, and new processor-specific cryptographic keys. The main goal of the architecture is to prevent other applications and even the operating system from subverting the control-flow integrity of an application or observing its runtime state. The SGX architecture provides such isolation of security-critical application execution, but also gives rise to a new type of adversary, one that can terminate, restart and dictate the schedule of enclave execution (see Section III). Here we briefly describe the main protection mechanisms of SGX. For a more elaborate explanation of the architecture, we refer interested readers to [10].

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) [28]. 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 [10].

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 [10]). 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 [2], [10].

Attestation. Attestation is the process of authenticating that a particular piece of enclave code has been properly instantiated. In local attestation both the prover and the challenger reside on the same platform. This is done using the EREPORT mechanism which uses a report key, shared with the challenger, to sign security critical attributes of the prover enclave (e.g., its code hash and issuer key).

Remote attestation refers to a procedure where a remote challenger gains confidence that a particular piece of enclave code has been instantiated on a trusted platform. To accomplish this, the SGX architecture uses an Intel-provided *Quoting Enclave* which can access a processor-specific attestation key to sign security critical enclave data and measurements. This signed report is returned to the challenger who uses an online *attestation verification service* (operated by Intel) to verify that the key being used is a valid Intel SGX attestation key. The attestation key is a part of the group signature scheme called EPID (Enhanced Privacy ID) used to sign objects without uniquely identifying the platform (processor), thus protecting end-user privacy [21], [10]. The challenger can verify the responses from the Quoting Enclave and the online attestation service, and confirm the indicated enclave code is running on a genuine SGX processor. Once an enclave has been authenticated using attestation, an external entity or another enclave on the same platform can establish a secure channel to it using an authenticated key exchange.

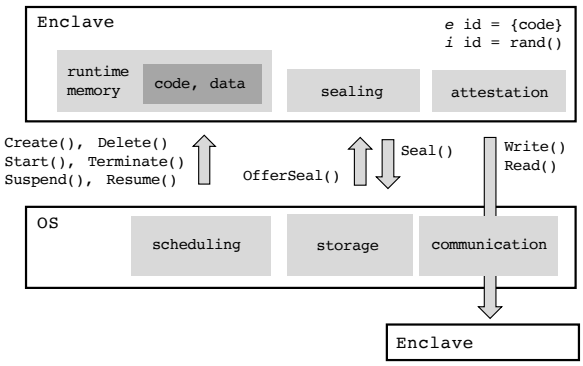


Fig. 1: SGX model. We model enclaves and the operating system, their main functionality, and the operations (scheduling, storage, communication) through which they interact.

Sealing. Enclaves can save confidential data across executions. Sealing is the process to encrypt and authenticate enclave data for persistent storage [1]. Typically enclaves perform sealing when their state has changed in such a way that the new state should be recoverable on the next execution of the enclave (possibly after a reboot). All local persistent storage, such as the 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.

III. 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.

A. 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 (isolation, attestation, sealing) 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 instances 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 identity e is rendered unusable.

Storage operations. The second set of operations that we model are 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 and 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. An external client can communicate with an enclave through read and write operations. Due to attestation, the client can write data such that only a particular enclave identity can read it. Similarly, the client can read data from an enclave and verify which enclave identity wrote it. We model these primitives as single operations, although the attestation is an interactive protocol between the enclave and the client (see Section II).

- $\texttt{Write}(m_e, i)$. 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.

We do not model platform reboots, as those have the same effect as enclave restarts. Our model assumes that each enclave instance has its own runtime memory that is perfectly isolated from the untrusted OS and other enclaves. In reality, enclave code and data are processed in shared processor caches. Additionally, all enclave code and data may not reside in the enclave runtime memory, but encrypted memory pages can be evicted to the untrusted memory. The OS-controlled paging mechanism has been demonstrated to leak information about the control flow of the enclave [46]. Cache-based side-channels are discussed in [10], [35]. We consider information leakage through side-channel attacks a realistic threat, but an orthogonal problem to rollback attacks, and thus outside of our model.

B. 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. As specified above, the adversary can schedule enclaves and start multiple instances of the same enclave. The adversary can 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 (i.e., no leakage through side-channels). 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 (attestation, sealing). The enclaves may implement additional cryptography, such as key exchange and message signing, and the adversary cannot break those operations.

C. 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.

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

D. Limitations of SGX Counters

Intel has recently added support for monotonic counters [19] 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.

Below we describe the counter functionality and explain our findings on experimenting with them.

SGX counter service. An enclave can query the availability of counters from the Platform Service Enclave (PSE). If supported, the enclave can create up to 256 counters. The default owner policy is that only the enclaves with the same signing key may access the counter. The 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 the same counter later, as there is no API call to list the existing counters. After a successful counter creation, the enclave can increment, read, and delete the counter.

According to the SGX API documentation [19], the 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 [17], the counter service is provided by the Management Engine on the Platform Control Hub (PCH).

Experiments. We tested the 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 the 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. [41] 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 of the 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.

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 the 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 counter service is unavailable.

Performance limitations. An enclave developer could attempt to use the SGX counters as a rollback 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 the integrity of the stored data based on the 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, the counters 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 and update their state accordingly are not possible.

Weaker security model. According to Intel developer forums [17], the 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, the actual location of the non-volatile memory used to store the counters is not publicly stated. Based on Intel specifications [18], [16], 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 the Management Engine is an active component, the 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 the FBI iPhone unlocking debate) [38]. Although the precise storage location of the SGX counters remains unknown at the time of writing, it is clear that if the integrity of the 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 [2, p. 30].

Other concerns. The current design of the 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. Every time the state is updated, the enclave increments the counter value. 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 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 could be probably fixed with better design of the 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.

E. Limitation of SGX Trusted Time

Another recently introduced and optional SGX feature is the trusted time service [20]. 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 [20].

We tested the time service and noticed that the provided nonce remained the same across platform reboots. Reinstalling the 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 the latest one.

F. Limitations of Other Known Solutions

TPM solutions. TPMs provide monotonic counters and NVRAM that can be used to prevent rollback attacks [30], [39], [41]. 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) [41]. Thus, similar to SGX counters, TPM based solutions are unsuitable for applications that require fast and continuous state updates.

Integrity servers. Another approach is to leverage a trusted server to maintain state for protected applications [23], [44], [24]. The drawback of this approach is that it requires setup and maintenance of a new infrastructure and 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 [7], 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 the architecture 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 architecture changes are costly and the current NVRAM technologies have the performance limitations discussed above.

G. 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 without weakening the enclave protection model. We seek to find a solution that does not require expensive hardware modifications or setup of new computing infrastructure, allows easy data migration for flexible resource use, and does not have a single point of failure for high level of security.

[1]The TPM 2.0 specifications introduce *high-endurance non-volatile memory* that enables rapidly incremented counters [14]. 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.

IV. Our Approach

The intuition behind our approach is that a single SGX platform cannot 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. Instead of using a standard state replication protocol, our distributed system is customized for the task of rollback protection to reduce the number of required replicas and the communication overhead.

To realize rollback protection, the distributed system should provide, for each participating platform, an abstraction 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.
- $value/empty \leftarrow$ `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 monotonic counter 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 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.

A. Distributed Model

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 III-A. 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.

B. Distributed Adversary Model

On each platform, the adversary has the capabilities listed in Section III-B. 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 [11] with adversarial behaviour (Byzantine faults) on a subset of participating platforms [32], [25]. Additionally, the adversary can schedule the execution of trusted processes, replay old versions of their persistently stored data, and start multiple instances of the same trusted process on the same platform. In Section VI we explain subtle attacks enabled by such additional adversarial capabilities.

C. 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 practical 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 message, 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 a counter is saved to the runtime memory of n assisting enclaves. The adversary that controls the OS on all assisting platforms starts second instances of the same enclave on n 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.

Counter technique	Security property	Enclave type	Crash resilience
inc-then-store	no any rollback	any	no
store-then-inc	no arbitrary rollback, last input twice	deterministic	yes

TABLE I: Comparison of counter increment techniques. *inc-then-store* provides strong security, but no crash resilience. *store-then-inc* supports crash resilience, but enables the latest input to be executed twice.

V. 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 start by explaining the rationale behind the chosen counter increment technique, our system architecture, group assignment and system initialization. After that, we describe the rollback protection protocols.

A. Counter Increment Technique

Two common techniques for counter-based rollback protection exist (see Table I). 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 [30], [41]. 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 [41].

The stronger security guarantee is needed, for example, in a financial enclave that withdraws randomized digital coins and maintains account balances. If the enclave executes the same input twice on the same state, the adversary can withdraw double the coins accounted. 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 (e.g., appending a transaction to a transaction history).

Our distributed rollback protection approach is agnostic to the choice of the counter increment technique. We design and implement ROTE using inc-then-store, because of its stronger security guarantees. However, our system could be easily adapted to provide crash resilience using store-then-inc.

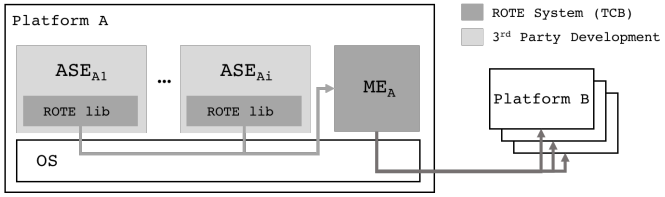


Fig. 2: The ROTE system architecture. Application-Specific Enclaves (ASEs) verify the integrity of their sealed data using the Master Enclave (ME) that distributes its own state to assisting platforms.

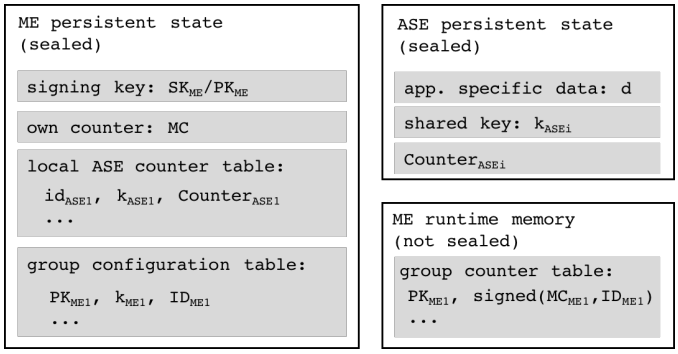


Fig. 3: ROTE system state structures. The ME maintains a local ASE counter table in its persistent state and a group counter table in its runtime memory. The group configuration table is set up during system initialization.

B. 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 Master Enclave (ME) and a 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 ME 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).

To provide state integrity for ASEs, the ME needs to protect its own state. The ME maintains a Monotonic Counter (MC), increases it for every ASE update, distributes it to MEs running on assisting platforms, and includes the counter value to its own sealed data. When the ME needs to verify the freshness of its own state, it obtains the latest counter value from the assisting nodes. The ME realizes the secure counter storage functionality (`WriteCounter` and `ReadCounter`).

The design choice of introducing a dedicated system service (ME) hides the distributed counter maintenance from the applications. Having a separate ME 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 the same time for the system to proceed.

These parameters have a dependency $n = f + 2u + 1$ (see Section VI). 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.

C. Group Assignment and 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 in scenarios where all assisting platforms belong to a single administrative domain (e.g., multiple servers in the same data center). In Appendix C we discuss group updates and group assignment alternatives.

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. We assume that the operating systems on these platforms are trusted at the time of system initialization (e.g., freshly installed OS version). During its first execution, the ME on each platform generates an asymmetric key pair SK_{ME}/PK_{ME}, and exports the public key. The public keys are delivered to the group owner securely, and the group owner issues a certificate by signing all group member keys. The group certificate can be verified by the ME on each selected platform by hard-coding the public key of the group owner to the ME implementation.

The ME is started a second time with the certified list of public keys and an *initialization key* as input parameters. The purpose of the initialization key 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. The initialization key is hard coded to the ME implementation in hashed format and the ME verifies the correctness of the provided key by hashing it. If the correct key is not provided, the ME aborts initialization. The ME proceeds by generating a random *instance ID*. A fresh instance ID is generated every time an ME is started and the purpose of the instance ID is to enable other nodes to distinguish specific ME instances and invalidate old ones. The ME saves the list of certified public keys PK_{MEi} to a *group configuration table* and runs an authenticated key agreement protocol to establish pair-wise session keys k_{MEi} with other MEs, and adds them and their respective instance IDs to the group configuration table. Finally, the ME creates a monotonic counter (MC), sets it to zero, and seals its state.

When an ASE wants to use the ROTE system for the

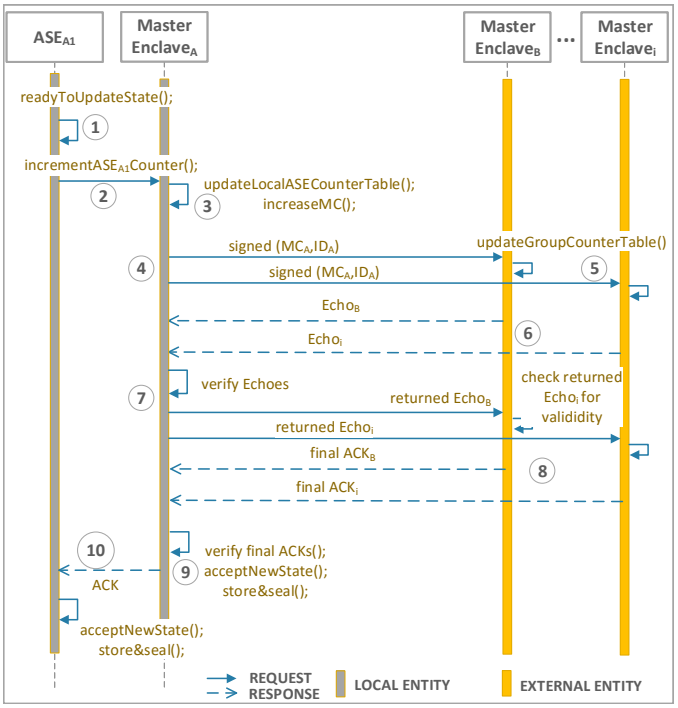


Fig. 4: ASE state update protocol. The ASE sends a counter increment request to the ME that increments the ASE specific counter in its table and increments its MC to distribute it to the assisting MEs. If the ME receives sufficient number of echoes and ACKs, both the ME and ASE seal their respective states.

first time, it performs an attestation of the ME. The code measurement of the ME can be hard-coded to the ASE implementation or provisioned by the ASE developer. The ASE runs an authenticated key establishment protocol with the ME. The ME 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.

D. 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 maintained account balances), it starts the state update protocol shown in Figure 4. This protocol can be seen as an customized version of the Echo broadcast [33]. We discuss the differences between our state update mechanism and Byzantine broadcast in Section IX. In all of our protocols the communication between the enclaves is encrypted and authenticated using the shared session keys. We add nonces and end point identifiers to each message to prevent message replay. The protocol proceeds as follows:

1) The ASE is ready to perform a security-critical state update that needs to be protected.
2) The ASE triggers a counter increment using the ME.
3) The ME increments a counter for the ASE, increases its own MC, and signs the MC and its instance ID using SK_{ME}. The counter is signed to preserve its integrity in the case of compromised assisting MEs.
4) The ME sends the signed counter and the instance ID to all MEs in the protection group.
5) Upon receiving the signed MC and the instance ID, each ME updates its group counter table. The table is kept in the runtime memory, and not sealed after every update, to avoid endless propagation.
6) The MEs that received the counter send an *echo* message that contains the received signed MC and the instance ID. The MEs also save the *echo* in runtime memory for later comparison.
7) After receiving a quorum $q = u+f+1 = \frac{n+f+1}{2}$ *echos*, the ME returns the *echos* to their senders. The second round of communication is needed to prevent attacks based on ME restarts during the update protocol.
8) Upon receiving back the *echo*, each ME 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 ME. If this is the case, the ME replies a final ACK message.
9) After receiving q final ACKs, the ME seals its own state together with the MC value to the disk.
10) The ME 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.

E. ME Restart Protocol

Figure 5 shows the protocol that the ME runs after a restart. The protocol performs three main functions. First, the ME joins an existing protection group with a new instance ID. Second, the ME retrieves its counter value from the group. And third, the restarted ME obtain the lost MCs of other nodes.

In restart the ME loses all previously established session keys. The ME generates a new, random instance ID for itself and establishes new pairwise session keys with other platforms in the protection group. In order to preserve our security guarantee, the target ME waits until it establishes new session keys with all other MEs residing in the protection group. All assisting MEs update their group configuration tables with the new session key and instance ID. The session key refreshing mechanism prevent nodes from communicating with multiple ME instances on one platform (see Section VI). Another condition for successfully joining the protection group is that sufficiently many nodes return non-zero counter values (step 6 below). This check prevents simultaneously restarted MEs from establishing a second, parallel protection group. Additionally, after obtaining its counter value, the ME increments its own counter (step 7). This allows other nodes to determine which instance ID is the latest in cases where the protection group has multiple instance IDs for the same node.

The protocol proceeds as follows:

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

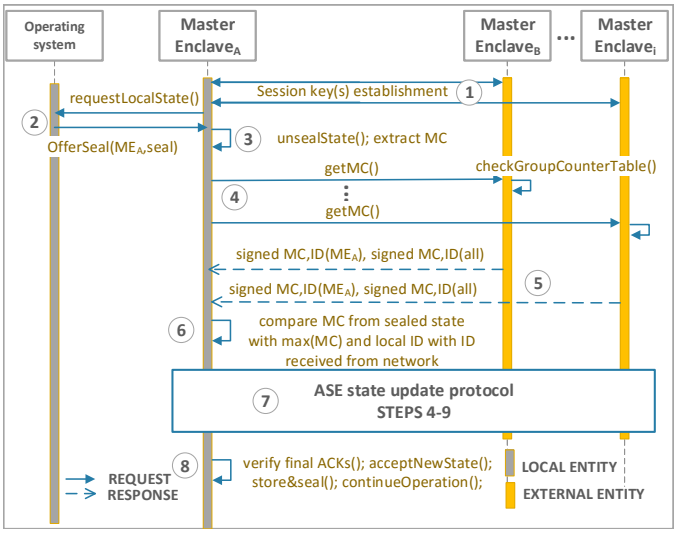


Fig. 5: ME restart protocol. The ME obtains the latest MC value from the assisting MEs and verifies the integrity of the unsealed data using it.

2) The ME queries the OS for the sealed state.
3) The ME unseals the state (if received) and extracts the MC.
4) The ME sends a request to all other MEs in the protection group to retrieve its MC.
5) The assisting MEs check their group counter table. If the MC is found, the enclaves reply the signed MC and the associated instance ID. Additionally, the complete table of other signed MCs and instance IDs that the responding node has in its memory.
6) When the ME receives q responses from the group it selects the maximum value and verifies the signature over it. We select the maximum value because some MEs might have an old counter value or they may have purposefully sent one. Additionally, the target ME verifies signatures and compares all the group counter table entries with received values for other nodes. For each assisting ME, the target ME picks the highest MC and updates its own group counter table with the value and the associated instance ID. The ME 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 ME initiates a MC update using steps 4-9 from the *ASE state update protocol*, Figure 4.
8) The ME stores and seals the updated state together with the incremented counter value. The ME will also save the counter value to its runtime memory.

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

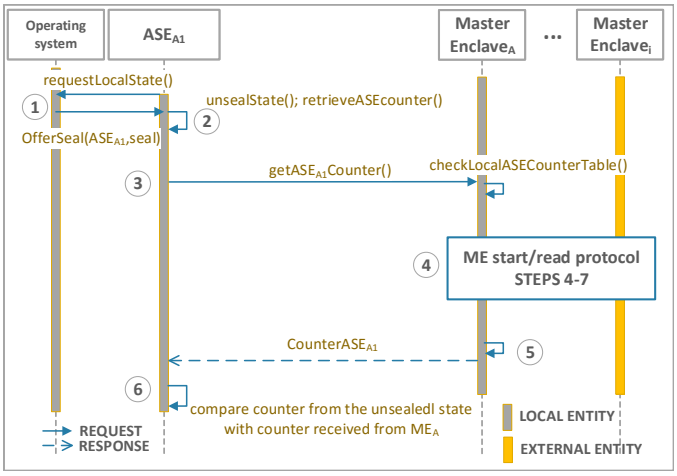


Fig. 6: ASE start/read protocol. An ASE obtains the latest data state counter from the ME and verifies the integrity of the unsealed data using it.

F. 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 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 (e.g., updated account balance).

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 ME to retrieve its latest ASE counter value.
4) To verify the freshness of its runtime state, the ME performs the steps 4-7 from the ME Restart protocol, with the following exceptions. When receiving counter values from the assisting nodes, the ME additionally checks if the highest counter value is associated with a different instance ID than what is stored in the group configuration table. This indicates that another node is running a newer ME instance. The old session key must be deleted and a new one established. If the obtained MC does not match the MC residing in the memory of the ME, the state of the ME is not the latest, and the ME must abort and be restarted. If the values match, the current data is fresh and ME can continue normal operation. Step 7 from the ME Restart protocol only occurs if the data in the enclave memory was not fresh.
5) If all verification checks are successful, the ME 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 counter values match, the ASE successfully loads the previously sealed state or completes the security-critical client request. Otherwise, the ASE refuses to load the unsealed state

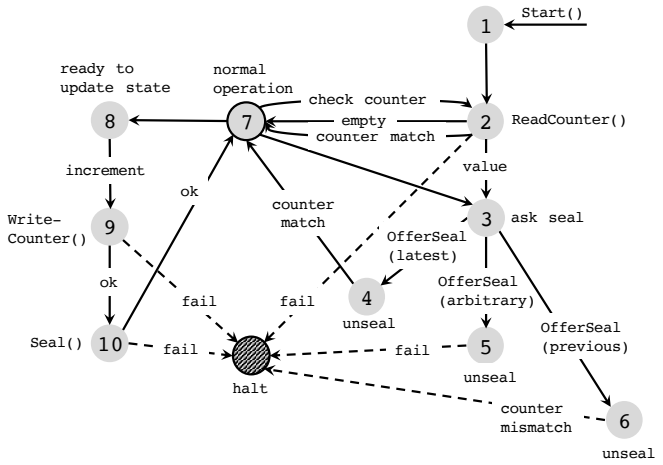


Fig. 7: State transition diagram showing enclave *execution states* using an ideal secure counter storage functionality. We show that the enclave can only accept the latest state or halt its execution.

or perform the client request.

VI. Security Analysis

Our system is designed to provide the following security property: an ASE cannot be rolled back to a previous state. In this section we first show that given a secure storage functionality (Section IV), the ME can verify that its state is the latest. After that, we show that the MEs realize the secure counter storage functionality as a distributed system. Finally, we show that ASEs cannot be rolled back if the ME cannot be rolled back. Our system achieves a security guarantee that we call *all-or-nothing rollback*. Enclave data integrity violation may only occur if the entire protection group is reset to its initial state.

A. Rollback Protection with Secure Storage

Given the ideal secure counter storage functionality defined in Section IV, rollback attacks can be prevented using the inc-then-store technique. In Figure 7 we illustrate a state transition diagram that represents ME states during sealing, unsealing and memory reading of security-critical data using the ideal 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 make the master enclave accept a previous version of sealed data.

First start. After creating an enclave, $e \leftarrow$ `Create(code)` the OS can start it using $i \leftarrow$ `Start(e)`, and the ME execution begins from State 1. The counter MC is set to zero in the runtime memory and the ME proceeds to State 2 to determine if it has previously saved sealed state. The ME reads the counter value from the secure storage using `ReadCounter()`. On the first execution the operation returns *empty* and the ME continues to State 7, where it continues normal operation. If the `ReadCounter()` operation fails, the ME halts.

Sealing. When the ME needs to seal data for local persistent storage (save its current state), it proceeds to State 8. The ME increments the monotonic counter MC, and performs the `WriteCounter()` operation to write it to the secure storage in State 9. If the writing operation succeeds, the ME continues to State 10. If counter writing fails, the ME halts. In State 10, the ME seals the data ($s \leftarrow$ `Seal(data)`) that represents its current state together with the counter value. OS confirmation moves the enclave back to normal operation in State 7.

Unsealing. When the ME needs to unseal data (recover its state) and the counter MC in the runtime memory has a non-zero value, the ME proceeds from State 7 to State 3. The ME asks the OS for the latest seal. 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, and the ME proceeds back to State 7. The adversary can also offer a previously sealed state (`OfferSeal(`$previous$`)`) which moves the execution to State 6. The unsealing is successful, but the counter value in the unsealed data does not match the MC value in the runtime memory and the ME halts. Finally, the adevrsary can offer any other data (`OfferSeal(`$arbitrary$`)`) which moves the ME to State 5. In this case the unsealing fails and the ME halts.

Security-critical memory read. The adversary can start more than one ME instance The SGX architecture does not enable one enclave instance to check if another instance of the same enclave code is already running [12].

When the ME needs to read its runtime state (e.g., to complete a client request), the ME proceeds from State 7 to State 2. This check is needed to guarantee that another instance of the same enclave does not have a newer state (forking attack). The ME reads the counter from the secure storage and compares the counter value to the one residing in its memory. If the counters are the same the ME concludes that it has the latest internal memory state and proceeds back to State 7. However, if the counter obtained from the secure storage is higher than the local monotonic counter in the memory, ME knows the freshness is violated and continues to State 3 in order to obtain the latest seal (see above).

Restart. After an ME restart, the execution proceeds to State 2. If the `ReadCounter()` operation returns a non-empty value, the ME proceeds to State 3, and if it returns empty we proceed to State 7, from where we follow the same steps as above.

If in any of these states the ME is terminated or restarted, its execution continues from State 1. Deleting 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 ideal secure storage functionality, the adversary cannot rollback the state of the ME.

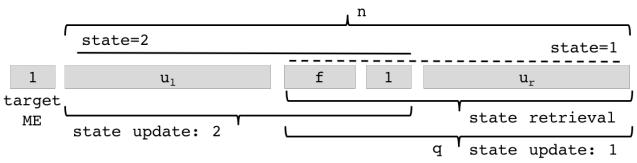


Fig. 8: Network partitioning example. On first state update the adversary blocks the left side of the group. On second state update she blocks the right side. To obtain the latest state, write to and read from a quorum of $q = u + f + 1 = \frac{n+f+1}{2}$ platforms.

B. Distributed Secure Storage Realization

Next, we show that ROTE realizes the secure counter storage functionality as a distributed system. When obtaining a counter value from the distributed protection group (`ReadCounter`), the enclave receives the latest value that was successfully 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. As defined in Section V, the ROTE system has three parameters: the number of assisting nodes n, the number of compromised nodes f, and the number of 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 the 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. 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 $f < n$. If we set $u = 0$, and therefore $q = n$, we can tolerate $f = n - 1$ compromised processors. However, with such parameters the system does not proceed if even one platform is not reachable at the time of sealing or master enclave restart.

Platform resets. If the counter is successfully written to q nodes, then at least $u + 1$ honest assisting MEs have the latest MC value in their runtime memory. If an assisting ME is restarted, it attempts to recover the lost MC values (its own counter and counters for other nodes) from the rest of the protection group.

Three distinct cases are possible. The first case is that the number of restarted MEs is at most u. Since the number of not restarted nodes is $u+f+1 = q$ there is sufficient available platforms with the correct MC for the counter retrieval, concerning network partitioning. All restarted platforms will recover the correct MC. The second case is that more than u platforms, but not the entire protection group, are rebooted at the same time. The number of remaining platforms is not sufficient for ME recovery and the distributed system no longer provides successful MC reading or writing, but no rollback is possible. The third case is all $n + 1$ nodes are restarted at the same time, in which case new system configuration has to be deployed again by the group owner to re-initialize the system.

Two-phase counter writing. The above analysis showed that if the counter is successfully written to at least q nodes, then at least $u + 1$ honest nodes have the latest counter value, and during reading at least one correct value is always received. We also showed that the system can maintain the counter, if the adversary resets assisting MEs *after* successful counter writing. What remains to be shown is that our two-phase status update protocol successfully writes the counter to q nodes, despite possible ME resets *during* the protocol.

We illustrate the challenges of counter writing in this particular setting through an example attack on a single-phase variant of the status update protocol. This variant completes after the ME has received q echos. In this attack, during state update, the adversary blocks all communication and performs sequential passing of the messages. First, the attacker allows the message delivery to only one node that saves the counter and returns an echo. After that, the attacker restarts the ME on that node, which initiates the lost counter recovery procedure from the rest of the protection group. The adversary blocks the communication to the target platform, and the restarted ME recovers the previous counter value, because all the reachable MEs have not yet received the new value. The adversary repeats the same process for all other 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 phase of our state update protocol prevents such attacks. Below we show that no combination of ME restarts during the state update protocol allows the target ME to complete the protocol, unless the counter was written to q nodes and at least $u + 1$ honest nodes saved the counter value. There are four distinct cases to consider. (Below we assume that the adversary restarts at most u platforms simultaneously. If more than u platforms are restarted, recovery is not possible, as explained above.)

- *Case 1: Echo blocking.* If the attacker blocks communication or restarts assisting MEs so that q nodes cannot send the echo, the target ME does not receive enough responses and the protocol does not complete.
- *Case 2: No echo blocking.* If the attacker allows at least q echoes to pass, the target ME can start returning these echoes to the original senders. Depending on the attacker actions on the assisting MEs after they have sent the echoes, the following two cases are possible.
- *Case 2a: No restarts during first round.* If none of the assisting MEs were rebooted during the first protocol round, then at least $u+1$ nodes have the updated MC of the target ME in their runtime memory. If the adversary restarts assisting MEs before they sent the final ACK

and after they received the self-sent *echo* returned from the target ME, the protocol will not complete, because less than q final ACKs will be received. The adversary can also restart assisting MEs after they have sent the final ACK which will result in successful state update, and successful state recovery of the restarted ME since sufficient number of the assisting nodes already have the updated counter value.

- *Case 2b: Restarts during first round.* If the adversary restarts assisting MEs during the first round, the update protocol will either successfully complete (q final ACKs received) or halt execution 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 ME sends each of the received echoes to the original sender from the assisting nodes. Because of sequential ME 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 MEs will deliver the final ACK, and the protocol will not complete.

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 saved the counter value.

Forking attacks. Finally, we consider forking attacks in our distributed setting. We prevent the above forking attack by requiring that the ASE start/read and ME restart protocol include ME connecting to the assisting nodes and verifying its latest counter and seal. This adds a small delay to every ASE start and read operations. If the latest counter is correct along with the associated instance ID, ME can be certain that it has been the one that made the last update. If the comparison fails, or the sessions keys are outdated thereby preventing communication with other nodes, ME knows another version has been running and proceeds with new key establishment, unsealing and retrieving its latest state, outlined in the ME restart protocol. This ensures that performing any execution in the system is done on the latest and fresh data that was sealed and protected. Additionally, in order to uniquely identify the latest running instance of the ME, every read of the state initiates a state update protocol, thereby increasing the value of the MC and making the forked ME instance outdated (only maximum values of the MC are accepted, and having an instance running on the old MC and its associated instance ID indicates to the protection group to remove sessions keys for it).

The session key refreshing mechanism allows us to uniquely identify the latest running instance and prevents parallel communication with two instance running on one platform. After every instantiation or restart of the ME, new instance ID is generated and keys have to be established with all nodes from the protection group. When the ME start the key establishment protocol with any of the other nodes, they delete the old session key that they shared with the previous instance residing on the same platform, rendering its communication unusable. If at least q nodes communicate with the latest instance of the ME on a platform, the previous instance does not have a quorum of other MEs available to communicate to in order to perform any operations (read or write) that would violate our security properties. If this previous instance starts establishing new keys with other nodes, it is forced to read and load the latest, fresh state along with increasing the MC thereby making it the latest instance (and at the same time making the other one to lose the quorum). With the addition of the MC increase outlined above, we always have only one instance with the latest state (and the highest MC) on each platform. The protocol requires ME to always verify freshness of its own memory when an ASE request a state retrieval, thereby preventing any other instance running on the platform to roll back the ASE.

Another attack is to form a parallel network by instantiating new MEs on all of the platforms in the protection group and make the ASEs to communicate with MEs that formed the rogue system, causing rollback. However, our system prevents it since upon first start, the ME will receive all zero MCs from other nodes in the group and prevent forming of the network if there is no password entered. Hence, a new network may only be created under supervision of the group owner, by distributing the initialization key.

Lastly, we require all nodes to be reachable when a restart of the ME occurs, while establishing new session keys, to prevent the attacker from instantiating new MEs on different platforms in a one-by-one manner while keeping some of the nodes disconnected. This would result with a node that has no notion of the newly created instances and thus connecting to the old ones, enabling rollback. However, if all nodes are available during session key establishment, everyone in the protection group has a notion of the latest ME instances and would refuse to communicate with deprecated instances that hold an old data state.

C. Security Analysis Summary

If the target ME 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 will only accept the latest sealed state they produced. The adversary may only cause a complete system reset if all the nodes are restarted at the same time.

VII. Performance Evaluation

In this section we describe our implementation and experimental evaluation.

A. Implementation

Our implementation consists of the following components. First, we implemented the ME and an accompanying *master relay* application. Second, we implemented the ROTE library, a simple test ASE, and a matching *test relay* application. The purpose of the relays is to mediate enclave-to-enclave

component	LoC
test relay	100
master relay	1600
test ASE	100
ROTE library	150
Master Enclave (ME)	950
TCB size	**1100**

TABLE II: ROTE prototype implementation size.

communication (enclaves cannot communicate directly). We implemented all components in C++, the relays were implemented for the Windows platform. Table II shows the sizes of implemented components.

Communication. The communication between the relay applications and enclaves takes place using function calls from the Intel SGX API. The local communication between the relay applications was implemented using Windows named pipes. The communication across platforms was implemented using the standard Windows networking stack. When an ME sends a message to another ME, it first passes the data to the local master relay that sends it to the remote master relay over the network where it is passed to the remote ME.

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

B. State Update and Read Delay

Experiments. The main performance metric that we measure is the ASE state update and state retrieval *delay* at the time of sealing and unsealing. In this case we disregard disk I/O, which we consider later (Section VII-C). This operation includes counter distribution/retrieval to and from the protection group over the network, and thus the delay depends on the size of the protection group and the network. Depending on the ASE, such updates may be needed frequently. The ASE unsealing is a local operation, and thus constant time and fast (e.g., 10 μs depending on the size of the sealed data). The ME start would be normally performed only once per platform boot, and thus the operation is not time-critical.

In our implementation, all the messages passed between the nodes were 224 bytes, where 200 bytes indicate the encrypted payload and the remaining 24 bytes is the header containing routing information and sent in plaintext. We report state average update delays over 100,000 executions for the first and second experimental setup, and 100 executions for the third experimental setup (due to larger networking times).

Our first experimental setup consisted of four SGX-enabled laptops connected via local network (1-GBit Ethernet, off-the-shelf router). The laptops had Windows 10 OS, Intel i7-6500U processor and 8GB RAM. The round-trip time between the laptops was measured at $\leq 1ms$. For this setup, we used the ROTE prototype implementation.

nodes	locations
2	US (West), Europe
3	US (West), Europe, Asia
4	US (West), Europe, Asia, South America,
5	US (West), Europe, Asia, South America, Australia
6	US (West), Europe, Asia, South America, Australia, US (East)

TABLE III: Geographically distributed protection groups.

nodes	state update delay	state read delay
2	2.00 ms ($\pm$1.5 μs)	1.34 ms ($\pm$1.0 μs)
3	2.02 ms ($\pm$1.3 μs)	1.35 ms ($\pm$1.1 μs)
4	2.03 ms ($\pm$1.4 μs)	1.37 ms ($\pm$1.0 μs)

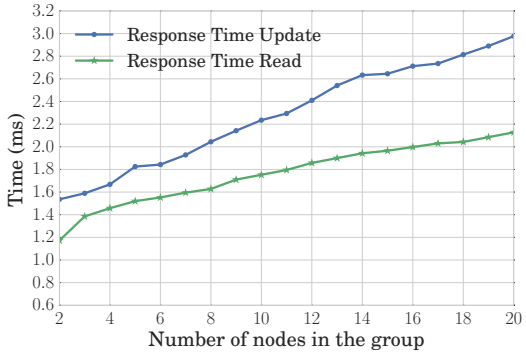
TABLE IV: State update delay on local network.

Our second experimental setup consisted 20 identical desktop computers connected via local network (1-Gbit Ethernet, university network). All desktops had Windows 10 OS, Intel i7-6700T processor and 8GB RAM. The round-trip time between the computers was measured at $\leq 1ms$. For this setup, we used a simulated implementation that is otherwise identical but does not run the protocol as enclaves.

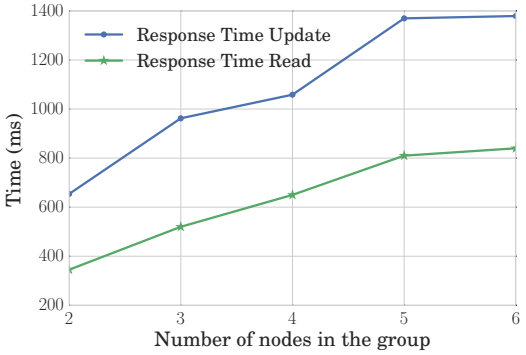
Our third experimental setup was a geographically distributed protection group of sizes from two to six nodes that we tested on Amazon AWS EC2 instances with Windows Server 2016, Intel Xeon-E5-2686 processor, 16GB RAM and dedicated 750 Mbit Internet connections. We distributed the platforms across different continents as shown in Table III. Also for this setup we used the simulated implementation.

Results. The state update delay consists of two components: networking overhead and SGX processing. In Appendix A we provide measurements on the cryptographic operations on SGX, measured on an average laptop (Intel i7-6500U processor and 8GB RAM). 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 the state update protocol requires is asymmetric signatures (0.46 ms per signing operation). The ASE state update protocol has one signature creation, the signature is verified later in the ME start/read and ASE start/read protocol. The required processing time of the state update protocol is less than 0.6 ms on an average laptop, where the creation of the first protocol message takes 0.51 ms (due to signing operation) and the creation of the second, third and forth messages is fast (approximately 5 μs). The state reading protocol requires the master enclave to perform 1 round trip to retrieve the latest counter and the verification of the signature which takes 0.84 ms.

Table IV shows the results from our first experimental setup using the ROTE prototype. 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. (The round-trip time using a 1-Gbit router took $\leq 1ms$ and with 4 messages passed between the nodes it equals 2 ms with a maximum deviation of 3.2 μs measured across 100,000 repetitions).



(a) State update delay (local group)



(b) State update delay (global group)

Fig. 9: Experimental results. The first figure shows ROTE performance (update and read delay) for protection groups that are connected over a local network. The second figure shows performance for geographically distributed protection groups.

Figure 9 shows the results from our second and third experiments using simulated implementation for larger groups. Figure 9a illustrates the status update delay for group sizes up to 20 platforms in the local network (round-trip time less than a millisecond). We see 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 a group size of five nodes, the update delay is 1.82 ms, read delay is 1.52 ms and for group size of 20 nodes, the delay is 2.98 ms and 2.13 ms, respectively.

Figure 9b illustrates the status update time for geographically distributed group sizes up to six platforms (see Table III). The observed increases in delay is less systematic, 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 distributing it over five locations gives us the update time of 1.37 seconds. The read delay equals to 342 ms and 810 ms, respectively to the number of different location.

We draw two conclusions from these experiments. First, the performance overhead imposed by our rollback protection mechanism is defined largely by the network connections between the platforms. Second, if the nodes are connected over a low-delay network, ROTE can support applications that require frequent state updates (e.g., over 380 updates per second). For applications that can tolerate larger sealing delays (e.g., more than 600 ms per state update), the ROTE system can be run on geographically distant protection groups.

C. Example Application Performance

In addition to state update delay, we measured the throughput of an example financial enclave with no rollback protection, using ROTE, and using SGX counters (see Table V).

Our example application is a financial enclave that processes incoming transactions from a buffer that is never empty. The experimental setup was a protection group of four nodes. We used the Intel NUC for disk operations due to its fast NVMe SSD; we used the two Dell laptops and the Lenovo laptop as the assisting nodes; lastly, we used the Lenovo laptop to test the SGX counters. For every processed 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 transaction processing and local state of the app can differ based on the exact use case scenario. 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). Additionally we tested the ROTE system with two, three and four nodes as protection group members and the difference in the results varies between 1-3% covered by the standard deviation of the results itself.

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

TABLE V: Performance comparison for the example application w/o ROTE, using ROTE and using SGX monotonic counters.

VIII. DISCUSSION

In this section we discuss data migration, performance optimizations, and information leakage.

A. Data Migration

Although sealing binds encrypted enclave data to a specific processor, our solution enables easy 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 ME. The ME sends the enclave data to another master within the same protection group together with the measurement of the ASE. The communication channel between the master enclaves is encrypted and authenticated. On the receiving processor, the ME passes the enclave data to an instance of the same ASE (based on attestation using the received measurement) which can seal it. Combined with group updates (see Appendix C), such enclave data migration enables flexible management of available computing resources. Similar data migration is discussed in [40].

B. Information Leakage

As explained in Section III, our model excludes execution side-channels, such as page faults and measurements on shared caches. Here we briefly discuss additional information leakage that our solution may add. Each local enclave state update 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 minor threat. If the exact timing of such events should be hidden, the master enclave can generate network traffic that mimics state updates and master enclave restarts at random intervals. Communication between the enclaves is encrypted and authenticated to prevent eavesdropping.

C. Performance Optimizations

The main performance characteristic of our solution, the state update overhead, is dominated by the networking time and the asymmetric signature operation required for the first message of the state update protocol. In case of a local 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 counter 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 application scenario.

Alternatively, because symmetric MAC operations are significantly faster (approximately 1 μs), the protocol could be optimized by replacing the asymmetric signature with n MACs that are calculated using the pair-wise keys established during system initialization. As malicious nodes can create fake MACs, it is no longer sufficient to receive q responses from any of the n nodes and pick the maximum value during counter read (as explained in Section VI-B). Instead at least q responses must be received from the same nodes as written previously and at least q responses must have the same value for the read operation to succeed. Such optimization can make state updates approximately 0.5 ms faster, but it can prevent the system from proceeding in environments with high node unavailability or network partitioning.

D. Consensus Applications

In the specific case that all the participating enclaves implement a distributed application whose purpose is to maintain a consensus (e.g., a permissioned blockchain), our rollback protection mechanism 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 needs to determine its latest state), it queries its latest state from the participating enclaves similar to our ME start protocol. We leave a detailed design as future work.

E. Forking prevention

Along with the novel approach that we implement, a second possible approach might be used by leveraging a TPM. After system boot, the started ME instance could extend a PCR that has a known value at boot. If a second ME instance is started, it can check if the PCR value differs from its known initial value [41]. This approach might be more practical than involving the whole system to perform verification, however it increases the system security perimeter to be outside of the processor.

IX. Related work

SGX counters and TPMs. Ariadne [41] uses TPM NVRAM or SGX counters for enclave rollback protection. The counter is incremented using store-then-inc technique that provides crash resilience, but allows two executions of the latest input on the same state. Ariadne minimizes the TPM NVRAM wear out using counter increments that only require a single bit flip. This work has three main differences to our solution. The first is service availability. SGX counters are an optional feature, not supported by all platforms, while our solution leverages functionality available on all SGX platforms. The second is performance. SGX counter increments take 80-250 ms and make the non-volatile memory unusable after few days of continuous use. Similar performance limitations apply to TPM NVRAM as well. Our solution performs counter increments in 1-2 ms (local network) and imposes no limits on the number of state updates. The third difference is the security model. Solutions that rely on SGX counters are likely vulnerable to bus tapping and flash mirroring attacks [38]. In our solution the trust perimeter is the processor package.

Memoir [30] leverages TPM NVRAM for rollback protection of isolated applications. The main drawback is as above: NVRAM updates are slow for high-throughput applications and continuous use will wear out the memory in few days. 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.

The ICE system [39] enhances the CPU with protected volatile memory, a power supply and a capacitor. At system

	ROTE state update	consistent broadcast Echo broadcast [33]	Byzantine agreement PBFT [7]
stored value	counters	arbitrary	arbitrary
sender	trusted	untrusted	untrusted
provides total order	no	no	yes
replicas	$f+1$	$3f+1$	$3f+1$
message complexity	$O(n)$	$O(n)$	$O(n^2)$

TABLE VI: Comparison between ROTE state update protocol and common Byzantine broadcast and agreement primitives.

boot, a base value is written to NVRAM in the TPM. After that, a non-reversible update history (hash chain) is recorded in the processor protected memory and at system shutdown the capacitor flushes the latest history version. After reboot or crash, the unsealed data is only accepted if it matches the latest history version and the base value in the TPM. Both the optimized Memoir and ICE require hardware changes.

Integrity servers. Another common approach is to store data integrity information on a separate server. For example, the Verena system [23] maintains authenticated data structures for web applications and stores integrity information (hashes) for these stuctures 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 [24]. In contrast to our system, such solutions requires setup and maintenance of new infrastructure and the integrity server becomes an obvious attack target.

Byzantine broadcast and agreement. Our state update protocol resembles well-known Byzantine broadcast primitives [6]. Echo broadcast [33] is an example of a *consistent broadcast* that ensures that non-malicious nodes do not receive different values. Bracha broadcast [4] is an example of a *reliable broadcast* that ensures that all non-malicious nodes receive the same value. Practical BFT [7] is an example of a Byzantine agreement (or *total-order broadcast*) protocols that additionally ensures that all the nodes receive the broadcasted values in the same order [6].

Our state update mechanism requires that the target platform only accept the update and creates the seal if a sufficient number of assisting nodes have received the counter value. Thus, our state update protocol follows the approach of Echo broadcast [33] with an additional confirmation message in the end. Like broadcast primitives, our state update protocol has $O(n)$ message complexity. Byzantine agreement protocols typically require $O(n^2)$ messages. Another significant difference is the number of required replicas. 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 sender (i.e., the target platform) is trusted and the distributed data is a signed counter value. Thus $f+1$ replicas are sufficient. Table VI summarizes the comparison.

Byzantine adversary models. Classical works on Byzantine agreement [25], [32] consider a model where f processes behave arbitrarily and the other processes behave as expected. The main difference to our model is that the attacker can control the execution schedule of the trusted processes as well. Researchers have also considered agreement under models where the Byzantine faulty nodes have some trusted functionality (e.g., an unmodifiable hardware primitive) available. Such approaches reduce the number of required replicas to $2f+1$ [8], [26], [9] or $f+1$ [22]. In our model we have no trust assumptions on the compromised nodes. Byzantine agreement has also been considered with so called dual failure models [29], [13], [37] where the adversary can fully control the faulty processes and can read the secrets of other processes (but not affect their control-flow integrity). In our case, the adversary cannot read secrets from trusted enclaves but can restart them. Lastly, recent development showed a novel message aggregation technique that couples hardware-based TEE with lightweight secret sharing resulting in FastBFT protocol, which has better scalability and performance than the previously developed BFT protocols [27].

SGX adversary models. Several recent research papers consider adversaries against SGX-based systems [42], [47], [45], [34], [15], [5]. These works typically consider individual adversarial capabilities, such as platform restarts, but to the best of our knowledge none of them defines all operations through which the OS or the adversary can control enclave execution schedule. Additionally, Pass et al. present a formal abstraction of the attestation verification model enabling secure and composable two-party computation [31], while Sinha et al. introduce MOAT, a tool for verifying confidentiality properties of applications running in SGX covering several distinct SGX attacker models, yet different from the one presented in this paper [36].

X. CONCLUSION

In this paper we have proposed a new approach for rollback protection on Intel SGX. Our key 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 constructed a model that captures the adversarial capabilities to schedule enclaves and show that our system provides a strong security guarantee that we call *all-or-nothing rollback*. Our experiments demonstrate that distributed rollback protection is practical even if applications require fast or frequent state updates.

REFERENCES

[1] Introduction to intel sgx sealing, 2016. https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing.

[2] Intel software guard extensions, reference number: 332680-002, 2015. https://software.intel.com/sites/default/files/332680-002.pdf.

[3] Trusted platform module library part 1: Architecture, familiy 2.0, level 00 revision 01.16 october 30, 2014, 2015. https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.16.pdf.

[4] G. Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2), Nov. 1987.

[5] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. Securekeeper: Confidential zookeeper using intel sgx. In *Middleware 2016: 17th International Middleware Conference Proceedings*. ACM, 2016.

[6] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[7] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[8] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *ACM SIGOPS Operating Systems Review*, volume 41-6, 2007.

[9] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *International Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183, 2004.

[10] V. Costan and S. Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 20 16. http://eprint. iacr. org, 2016.

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

[12] I. S. S. Forum. Ensuring only a single instance of enclave, 2017. https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/709552.

[13] J. A. Garay and K. J. Perry. A continuum of failure models for distributed computing. In *International Workshop on Distributed Algorithms*, pages 153–165. Springer, 1992.

[14] T. C. Group. Trusted platform module library, part 1: Architecture, family 2.0, 2014.

[15] D. Gupta, B. Mood, J. Feigenbaum, K. Butler, and P. Traynor. Using intel software guard extensions for efficient two-party secure function evaluation. In *Proceedings of the 2016 FC Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, 2016.

[16] Intel. 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.

[17] Intel. Developer zone forums, 2016. https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/607330.

[18] Intel. 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.

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

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

[21] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. Intel software guard extensions: Epid provisioning and attestation services. Technical report, Intel Corporation, Whitepaper 2016. https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services, 2016.

[22] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *European conference on Computer Systems (EuroSys)*, pages 295–308, 2012.

[23] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for web applications. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

[24] K. Kostiainen, N. Asokan, and J.-E. Ekberg. Credential disabling from trusted execution environments. In *Nordic Conference on Secure IT Systems (Nordsec)*, 2010.

[25] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3), July 1982.

[26] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Network Systems Design and Implementation (NSDI)*, volume 9, pages 1–14, 2009.

[27] J. Liu, W. Li, G. O. Karame, and N. Asokan. Scalable byzantine consensus via hardware-assisted secret sharing. *arXiv preprint arXiv:1612.04997*, 2016.

[28] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[29] F. J. Meyer and D. K. Pradhan. Consensus with dual failure modes. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):214–222, 1991.

[30] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *IEEE Symposium on Security and Privacy (SP)*, 2011.

[31] R. Pass, E. Shi, and F. Tramer. Formal abstractions for attested execution secure processors. Technical report, Cryptology ePrint Archive, Report 2016/1027, 2016.

[32] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[33] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80. ACM, 1994.

[34] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 45–48. ACM, 2016.

[35] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[36] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[37] H.-S. Siu, Y.-H. Chin, and W.-P. Yang. A note on consensus on dual failure modes. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):225–230, 1996.

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

[39] R. Strackx, B. Jacobs, and F. Piessens. Ice: A passive, high-speed, state-continuity scheme. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[40] R. Strackx and N. Lambrigts. Idea: State-continuous transfer of state in protected-module architectures. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2015.

[41] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security Symposium*, 2016.

[42] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. Cryptology ePrint Archive, Report 2016/635, 2016. http://eprint.iacr.org/2016/635.

[43] Trustonic. Trusted execution environment (tee), 2016. https://www.trustonic.com/technology/trusted-execution-environment.

[44] M. van Dijk, J. Rhodes, L. F. G. Sarmenta, and S. Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *ACM Workshop on Scalable Trusted Computing (STC)*, 2007.

[45] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security*, pages 440–457. Springer, 2016.

[46] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[47] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *Conference on Computer and Communications Security (CCS)*, 2016.

APPENDIX A
ADDITIONAL PERFORMANCE MEASUREMENTS

Operation	**Time**
enclave switching time	2.6 ($\pm$ 0.0) μs
SHA256	2.4 ($\pm$ 0.0) μs
Opening and closing ECC context	2.4 ($\pm$ 0.0) μs
ECDSA signing (0.5KB)	457.5 ($\pm$ 0.3) μs
ECDSA verfication (0.5KB)	843.6 ($\pm$ 0.9) μs
Sealing (1KB)	9.6 ($\pm$ 0.1) μs
Unsealing (1KB)	4.5 ($\pm$ 0.1) μs
Rijndael AES-CTR$_{128}$ encryption (0.5KB)	0.63 ($\pm$ 0.0) μs
Rijndael AES-CTR$_{128}$ decryption (0.5KB)	0.62 ($\pm$ 0.0) μs
Rijndael AES-GCM encryption + MAC (1KB)	1.05 ($\pm$ 0.0) μs
Rijndael AES-GCM decryption + verification (1KB)	1.07 ($\pm$ 0.0) μs

TABLE VII: Cryptographic operations on SGX.

Table VII provides measurements of cryptographic operations on SGX. We report average time over 1M repetitions. All enclave operations are reported switching time excluded. The test platform was running Windows 10 OS, Intel i7-6500U processors, 8GB RAM and a 256GB SSD.

Along with the state update and read delay we measure the theoretical system *throughput*, measured as the number of state update operations that one platform or an entire protection group can process (encompassing only the system protocol overhead).

The throughput per node in the local network is shown in Figure 10a. For a group size of 10 nodes, we measure over 435 updates and 570 read operations per second. The throughput for the entire group is shown in Figure 10b. For a group size of 20 platforms, we measure more than 6,700 updates and over 9400 read operations per second. We emphasize that these numbers represents the best-case scenario, as there is no significant other traffic in the network and no significant other computation on the platforms during our measurements. The observed increases in delay, and decreases in throughput, is less systematic, due to the dependency on network connections between various geographic locations in the protection group.

The throughput per node and the throughput of the entire group in the geographically distributed setting are shown in Figures 10c and 10d. For a group size of six platforms, we measure 0.73 updates and 1.2 read operations per second for a single node, while the entire group consequently achieves 4.4 updates and 7.1 read operations per second.

APPENDIX B
ADDITIONAL DISCUSSIONS

A. Deployment Scenarios

Here we discuss briefly two types of deployment settings. The first type is a shared computing resource such as a cloud platform. The computing resource consists of multiple platforms that belong under the same administrative domain (e.g., cloud platform provider), and thus the domain owner can assign multiple processors to participate in collaborative rollback protection. The ASEs may originate from different third-party developers. The second type is a deployment where the participating platforms are owned by private individuals. Each user benefits from rollback protection for their own enclaves and platforms, and is therefore willing to provide resources for other users in a collaborative system.

Depending on the deployment setting, the adversary could be a malicious administrator that can compromise operating systems, control the networking between the nodes, and physically tamper with a small number of processors. If the system is deployed among private individuals, the adversary could be a malicious user that has compromised his own processor locally through physical attacks and operating systems of other users remotely.

B. System Robustness

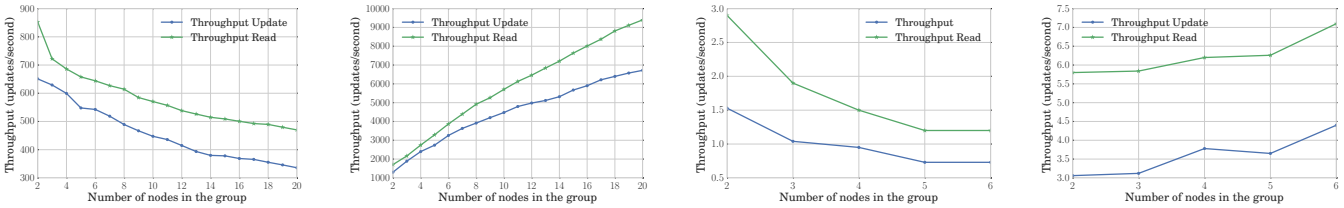
In this paper we consider a powerful adversary that controls the OS on all platforms, all network communication, and has compromised a subset of the processors. Such adversary can, obviously, cause a denial of service. Examples include deleting sealed data instead of placing it on the local persistent storage or by blocking all network communication during master enclave restart. While our system cannot prevent denial of service completely, it should be robust in the sense that benign program crashes or minor networking errors should not place the system in a state form which it cannot recover (without losing the previously sealed data).

Crash resilience. If the master enclave crashes after a sufficient number of assisting platforms have confirmed a state update (Step 8 in Figure 4), but before it has provided an updated sealed state to the OS (Step 9), the master enclave is left in a state from which it cannot recover (i.e., the distributed secure storage has a higher counter value than any existing seal). We consider such master enclave crashes unlikely. The master enclave is a system component that can be extensively analyzed (e.g., our prototype implementation is only 1000 LoC). Sudden power loss or OS bugs are other possible causes for ME crashes. Another critical time point is if an ASE crashes after an update request (Step 1 in Figure 4) and before sealing (Step 10). Although the crashing time window is very limited, assuming that all third-party enclaves are free of errors is more problematic.

To provide crash resilience we could adopt a technique from the Memoir system [30]. The intuition behind this technique is that a deterministic enclave can be safely rolled back to a previous state, if it will only execute the same input data after the recovery as during the crash. Because the input is the same, the enclave will not leak any new information to the adversary on the execution following the crash and rollback. The technique requires that a history of input values (a hash chain) is included to the sealed data and verified during possible claimed crash recovery.

To support crash resilience our system would require the following modifications. Each ASE would need to record all inputs to a hash chain: $h(h(h(i_1)\|i_2)...)$. Similarly, the ME would need to form an aggregated hash with every state hash received from the ASEs. The aggregated hashes should be sealed prior to MC sending, and the message with the signed MC should include the last aggregated hash. This would allow the enclaves to recover from system crashes at critical time points. For example, the ME could accept the previous version of sealed state and compare the provided input to match the aggregated hash. If the inputs match, it is safe to proceed. The same principle is valid for the ASEs. The limitation of this approach is that enclave applications must be deterministic.

Network errors. Similarly, if there is a network problem that allows echo lists to reach assisting nodes, but prevents q final ACKs from reaching the target node (Step 8 in Figure 4), the system is left in a state from which it cannot recover. To reduce the possibility of such events, the system parameter u can be increased and the assisting nodes can be chosen from different locations.



(a) Single platform throughput (local) (b) Group throughput (local) (c) Single platform throughput (global) (d) Group throughput (global)

Fig. 10: Experimental results. The first two graphics shows ROTE throughput performance for protection groups that are connected over a local network. The second row two graphics show performance for geographically distributed protection groups.

C. Other Security Architectures (TrustZone)

We have designed our rollback protection system for SGX. However, a similar approach could be adapted for other security architectures. ARM TrustZone realizes isolation through a dedicated processor mode and access control enforcements on the system hardware. The OS and the applications are executed in the Normal World mode, while security-critical code can be run in the isolated Secure World mode. In the TrustZone architecture, the hardware protections do not isolate applications running inside the Secure World from each other. Instead, a small security kernel within the Secure World can provide isolation between protected applications [43]. The security kernel can also measure the target application and issue signed statements over it (attestation). Like Intel SGX, TrustZone is vulnerable to rollback attacks. The runtime memory of Secure World does not typically provide non-volatile storage due to high manufacturing cost.

To implement our solution on TrustZone, the functionality of the master enclave could be implemented as part of the security kernel. The secure communication between application enclaves and master enclaves could be implemented as function calls within the Secure World. The attestation between master enclaves could be implemented by the security kernel. The security kernel could measure its own code and verify that another TrustZone platform is running the same security kernel configuration.

APPENDIX C
GROUP MANAGEMENT

In Section V we described the ROTE system using a trusted offline authority: the group owner. In this appendix we discuss how groups can be updated and established without trust on first use assumption or a trusted authority.

A. Group Updates

The group owner issues a signed list of public parts of the public-private key pairs generated by each master 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 old members and add new ones) without disabling access to the already rollback-protected sealed data.

During system initialization, the ME verifies the signed list of group member keys and seals the group configuration. When a group update is needed, the group owner can issue an updated list that will be processed and again sealed by the ME. Note that this approach does not need password entry such as in first group establishment. However, the adversary should not be able to revert the group to its previous configuration (e.g., one that includes compromised nodes) by replaying the previous sealed group configuration. Since group updates are typically infrequent, they can be protected using SGX or TPM counters.

At system initialization, the master enclave 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 [3]) is necessary. The group owner includes a version number to every issued group configuration. When the ME 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 ME increments either of these counters (depending which of those two solutions is implemented). Additionally, the ME establishes shared keys with the new group members and sends its MC to them. When the ME is restarted, it verifies that the version number in the unsealed group configuration matches the counter. The NVRAM memory available in TPMs is expected to support approximately 100K write cycles, while in the case of SGX counter we showed it supports 1M write cycles, a sufficient number 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.

Usage of TPM counters requires trust on first use, otherwise a malicious OS could redirect the TPM session establishment to another TPM (e.g., one in possession of the adversary). Usage of SGX counters does not have similar requirement.

B. Group Establishment Without Trust On First Use

The group establishment described in Section V-C requires trust on first use, because a malicious OS could modify the exported ME public key. This assumption may be removed,

if the group owner can perform remote attestation of the group member MEs such that the member platforms have no network connection besides to the group owner (i.e., the malicious OS cannot redirect the attestation session to another platform). However, in this case, the group owner has to deliver the password for establishing the system via an out-of-band channel to each node. Note that a compromised OS can always block the communication, re-direct the password to another enclave or read it from input. Thus, the group owner is responsible of tracking all the initial enclave and instance IDs and deliver them to securely to all participating nodes - enabling verification that the initial group establishment is secured.

C. Group Establishment Without Trusted Authority

Distributed rollback protection does not necessarily require a trusted authority. Alternatively, the participating platforms can decide the protection group configuration directly themselves. In the presence of potentially malicious nodes, such group establishment can be done using a Byzantine agreement protocol [25], [32] and remote attestation. Another alternative is to establish asymmetric groups where each platform distributes its state to a freely chosen and attested set of assisting platforms that in turn choose their assisting platforms independently.

Symmetric groups. An enclave that requires rollback protection chooses n SGX processors that should form the protection group. The enclave can discover the assisting nodes from a public directory, peer-to-peer manner etc. The enclave broadcasts the proposed group configuration to all chosen nodes using a Byzantine agreement protocol (total-order broadcast) that tolerates malicious nodes and guarantees that all non-malicious nodes agree on the same value. After the group is established, attestation is performed to verify the correctness of the code running and that it is running inside SGX. When an update to the group configuration is needed, the leader repeats the same process. Byzantine agreement protocols have high message complexity. Because byzantine agreement requires $3f + 1$ replicas, the rollback protection mechanism can no longer tolerate $f < n$ compromised nodes. Because Byzantine agreement protocols have higher message complexity $O(n^2)$, this group establishment is not applicable to very large groups.

Asymmetric groups. Another alternative is that each platform distributes its state to a freely chosen set of assisting platforms that in turn choose their assisting platforms independently. The set of platforms can be even selected for each state update separately. We call such protection groups *asymmetric*. Asymmetric groups offer flexible deployment, as every node is free to choose where it distributes its state at any given time. However, the different trust model requires minor changes to the rollback protection system, and it provides slightly different security and liveness properties. We discuss these differences briefly below.

The first main difference is the recovery of master enclave's own counter (MC) after restart. We illustrate this with an example. Assume that a node distributes its counter to one set of nodes s_1. At the next state update the node distributes its counter to another set of nodes s_2. The rollback protection system must prevent attacks, where the adversary causes counter recovery form s_1 which would result in a rollback. To prevent such attacks, the node can request s_1 to remove the saved counter from their memory and only after a successful confirmation the node proceeds with the state update to s_2. The list of nodes to communicate is stored in sealed format on the disk, and after master enclave restart the master enclave unseals the list and contacts the nodes on it. If the adversary replays an old sealed data, the master enclave gets no responses (as the counter has been deleted) and aborts unsealing.

Another difference is the recovery of stored counters after master enclave restart. The master enclave maintains counters for other nodes in its runtime memory (no sealing to prevent endless update propagation). When a master enclave is restarted, it needs to obtain its own counter value, and additionally the lost counter values of other nodes. In asymmetric groups, stored counter recovery must be handled differently. One option is to request the counter back from the sender, after a master enclave restart. This requires that the processor identity for each stored counter is saved on the local persistent storage.

Lastly, there is no mutual trust between the nodes, since they are running under different administrators. Hence, the participating entities have to be aware that this approach does not guarantee the same properties as in the symmetric group establishment.