A Sample *ACM* SIG Proceedings Paper in LaTeX Format*

Ke Zhang
3030058805
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
kzhang2@cs.hku.hk

Tianxiang Shen
3030058776
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
txshen2@cs.hku.hk

ABSTRACT

In this report, based on the mature secure enclave architecture Intel Software Guard Extensions (SGX), we survey three state-of-art secure key-value storage methods against rollback attacks. For each approach, we discuss respective practicality, efficiency, and secure guarantee level regarding to their inner mechanisms and experimental performances. Furthermore, we point out possible improving directions for methods we discuss in this reports.

1. INTRODUCTION

With the growth in cloud computing adoption, online data stored in data centers is growing at an ever increasing rate [5]. Modern online services ubiquitously use persistent key-value (KV) storage systems to store data with a high degree of reliability and performance [4]. Therefore, persistent KV stores have become a fundamental part of the cloud infrastructure.

At the same time, the risks of security violations in storage systems have increased significantly for the third-party cloud computing infrastructure [17]. Modern data processing services hosted in cloud environments are under constant attack from malicious entities such as database administrators, server administrators, hackers who exploit bugs in the operating system or hypervisor, and even nation states. This results in frequent data breaches that reduce trust in online services. Semantically secure encryption can provide strong and efficient protection for data at rest and in transit, but this is not sufficient because data processing systems decrypt sensitive data in memory during query processing. In an untrusted environment, an attacker can compromise the security properties of the stored data and query operations. In fact, many studies show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to storage systems [9].

*(Produces the permission block, copyright information and page numbering). For use with ACM_PROC_ARTICLE-SP.CLS V2.6SP. Supported by ACM.

However, securing a storage system is quite challenging because modern storage systems are quite complex [12]. Thereby, the enforcement of security policies needs to be carried out by various layers in the system stack, which could expose the data to security vulnerabilities. Furthermore, since the data is stored outside the control of the data owner, the third-party storage platform provides an additional attack vector. The clients currently have limited support to verify whether the third-party operator, even with good intentions, can handle the data with the stated security guarantees.

A approach to enable secure query processing is to trusted execution environments (TEEs), such as Intel Software Guard Extensions (SGX) [8] or ARM TrustZone [23], provide an appealing approach to build secure systems. Enclaves can protect sensitive data and code, even from powerful attackers that control or have compromised the operating system and the hypervisor on a host machine. While enclaves can mitigate several attacks, using them requires careful refactoring of applications into trusted and untrusted components to achieve desired security and privacy goals. Furthermore, ensuring high level security properties such as confidentiality, integrity, and freshness requires additional logic to protect secrets when they leave the enclave and verify their integrity when they are read. This task is relatively simple in applications such as password checkers, key management systems and simpler data processing frameworks. In fact, given the importance of security threats in the cloud, there is a recent surge in leveraging TEEs for shielded execution of applications in the untrusted infrastructure. Shielded execution aims to provide strong security properties using a hardware-protected secure memory region or enclave.

While SGX can be considered as a big step forward towards trustworthy cloud computing, some attack vectors nevertheless remain. One important open issue are rollback and forking attacks on stateful applications that make use of persistent storage. Whereas SGX provides mechanisms against main- memory replay attacks, persistent storage is not under the direct control of SGX and therefore harder to secure. The need to handle system restarts, operating system crashes, and power outages makes a completely secure solution for state continuity difficult to achieve.

In this report, we mainly survey three state-of-art methods in solving the secure key-value storage with rollback attacks protections, *i.e.*, Monotonic Counter for SGX, ROTE system, and Speicher system. We separately describe their

inner designations and their limitations together with our insights for future directions.

2. BACKGROUND

2.1 SGX Overview

2.1.1 Secure Enclave

A secure enclave is a set of software and hardware features that together provide an isolated execution environment to enable a set of strong security guarantees for applications running inside the enclave. Enclave allows user-level as well as Operating System (OS) code to define private regions of memory, whose contents are protected and unable to be either read or saved by any process outside the enclave itself, including processes running at higher privilege levels [6].

Intuitively, secure enclave fundamentally ensures the correctness and isolation in executing given process. The confirmation of input data freshness is hard to achieve, especially when the enclave encounters crash or restart. There are several wildly used secure enclave services [3], one of the most popular security architectures is Intel Software Guard Extensions (SGX) [8]. However, a mature secure enclave designation as SGX still shows unsatisfied performance towards rollback attacks. In this report, we focus on the SGX architecture and its existing promotions in proposing protection against rollback attacks.

2.1.2 SGX Architecture

In a standard SGX as specified in [8, 22], apart from the confidentiality and integrity nature of SGX, there are fundamentally three operations we concern in this report, *i.e.*, the enclave creation, the sealing, and the attestation.

- **Enclave creation.** An enclave is created by the user client. In enclave creation, the client specifies the code to be processed in SGX. Security mechanisms in the processors create a data structure called SGX Enclave Control Structure (SECS) that is stored in a protected memory area. Enclaves' code created by the client 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 a 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.
- **Sealing** Enclaves can save confidential data across executions. Sealing is the process to encrypt and authenticate enclave data for persistent storage [2]. All local persistent storage (*e.g.* disk) is controlled by the untrusted OS. For each enclave, the SGX architecture provides a sealing key that is private to the executing platform and the enclave. The sealing key is derived from a Fuse Key (unique to the platform, not known to Intel) and an Identity Key that can be either the Enclave Identity or Signing Identity. The Enclave Identity is a cryptographic hash of the enclave measurement and uniquely identifies the enclave. If data is sealed with Enclave Identity, it is only available to this particular enclave version. The Signing Identity is provided by an authority that signs the enclave prior to its distribution. Data sealed with Signing Identity can be shared among all enclave versions that have been signed with the same Signing Identity.
- **Attestation** Attestation is the process of verifying that certain enclave code has been properly initialized. In local attestation a prover enclave can request a statement that contains measurements of its initialization sequence, enclave code and the issuer key. Another enclave on the same platform can verify this statement using a shared key created by the processor. In remote attestation the verifier may reside on another platform. A system service called Quoting Enclave signs the local attestation statement for remote verification. The verifier checks the attestation signature with the help of an online attestation service that is run by Intel. Each verifier must obtain a key from Intel to authenticate to the attestation service. The signing key used by the Quoting Enclave is based on a group signature scheme called EPID (Enhanced Privacy ID) which supports two modes of attestation: fully anonymous and linkable attestation using pseudonyms [8]. The pseudonyms remain invariant across reboot cycles (for the same verifier). Once an enclave has been attested, the verifier can establish a secure channel to it using an authenticated key exchange mechanism.

In this report, protocols described in Section 4 primally utilize these three operations in SGX to provide rollback attack protections.

2.1.3 SGX Counter

Intel has recently added some supports for monotonic counters (MC) [11, 15, 1] as an optional SGX feature. The Monotonic Counter can be utilized by enclave developers for rollback attack protection.

SGX supports creating a limited number of MCs for each enclave. Monotonic counters are shared among enclaves that have the same code. An enclave can query availability of counters from the Platform Service Enclave (PSE). If supported, the enclave can create up to 256 counters. The default owner policy encompasses that only enclaves with the same signing key may access the counter. Counter creation operation returns an identifier that is a combination of the Counter ID and a nonce to distinguish counters created by different entities. On creating a MC, it gets written to the non-volatile memory in the platform. The enclave must store the counter identifier to access it later, as there is no API call to list existing counters. After a successful counter creation, an enclave can increment, read, and delete the counter. Because each enclave shares the same value of the monotonic counters, it guarantees the verification for data freshness. In other words, only when an enclave preserves the same counter value as the others in the platform, its reserving data are the latest. Also, when one enclave encounters crash or reboot, it can recover data with the help of monotonic counters shared in the platform.

According to the SGX API documentation [16], 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.

2.2 Rollback Attack

Rollback attacks remain a potential secure problem in secure enclave. In a rollback attack, attackers replace the latest data with an older version without being identified by the system.

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

In reality, enclaves cannot easily detect this replay, because the processor is unable to maintain persistent state across enclave executions that may include reboots or crash. Another way to carry out rollback attacks in secure enclaves is to create multiple instances of a same process and route update requests to one instance and read requests to the other. Due to the characteristic of secure enclave, the instances are indistinguishable to remote clients or OS.

To avoid rollback attacks, most commonly considered direction is to record the time related information for every state change. In this paper, we mainly discuss three designations in rollback attacks protection built on the SGX architecture. The goal of methods specified in Section 3 are to guarantee the data integrity, confidentiality, and freshness towards rollback attacks based on SGX architecture. Note that for different methods, different level of adversary's strength is considered, which is listed and compared in Section 4.

3. PROBLEM

As the infrastructure of cloud computing grows rapidly, storage service providers use Key-Value Stores (KVS) in data centers to persist user data, with high throughput and low end-to-end communication latency [18, 14]. Many users store their sensitive data (e.g., password, medical record) in these systems, while the protection of these data is not enough. Specifically, there are three dominant security properties in KVS: confidentiality, integrity and freshness. (a) **Confidentiality** is to ensure that other unauthorized parties (e.g., malicious OS) cannot read the plaintext data of personal record in KVS. (b) **Integrity** is the property that the typical *read* and *write* operations of KVS cannot be tampered with, such as the changes to records in persistent storage. (c) **Freshness** is the ability to detect stale state of data, in case a malicious KVS returns an older version of a request record.

Intel Software Guard eXtension (SGX), a popular security hardware on commodity available Intel CPUs, is promising to provide the first two security properties in KVS [8]. SGX provides an abstraction of secure enclaves, which is a secured memory zone isolated from untrusted memories. By sealing enclave objects with secret SGX keys to untrusted memory (i.e., persistent storage on host) and unsealing encrypted objects to enclave, SGX ensures that in-enclave data is unavailable from the outside, even with a malicious OS or hypervisor [6].

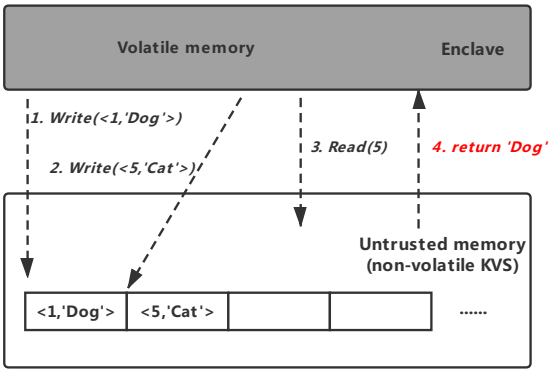

Figure 1: An example of rollback attack towards KVS on SGX-enabled host. The malicious OS returns an older version of value in KVS and the trusted enclave (in gray) cannot detect it. The records should be sealed/unsealed but we omit these operations for simplicity.

Unfortunately, the freshness cannot be guaranteed by simply running KVS on SGX-enabled hosts. The problem lies in the lack of version check when an enclave loads objects from untrusted memory. Figure 1 shows a typical rollback attack in a local scenario. The enclave calls the *write* operation of KVS twice to store two different key-value pairs, respectively. When the enclave requests for the latest value by calling *read*, the attacker returns a previous version of value to the enclave. Since the enclave can only verify source of the returned object from the correct platform through local attestation, the incorrect returned object cannot be detected by KVS users.

To formalize, in addition to leverage the protection of SGX, we should also develop a freshness protection mechanism to protect against rollback attacks that replay old state of objects. In other word, we aim to expand the security protection of SGX from trusted volatile memory of enclaves to untrusted non-volatile memory of the outside, even when the system reboot, crash or during migration.

4. SOLUTIONS

In this section, we mainly introduce three state-of-art solutions in solving the problem we mention in §3. We separately describe their motivation, inner designations, and respective improving directions in detail.

4.1 Monotonic Counter

In the latest version of SGX, Intel releases the abstraction of *monotonic counter* which can be utilized to protect against rollback attacks that replay objects [1]. When calling the *sgx_cerate_monotonic_counter* function from the SGX library, it automatically creates a limited number of monotonic counters (MC) for each enclave instance on the platform. The MC is shared among all the instances who

run the same code. Upon creating a new MC, it gets written to the non-volatile untrusted memory through a secure channel, preventing malicious OS or hypervisor from changing the counter value or replaying value [19].

With the help of monotonic counter, a basic approach is to store the state of objects with the counter into persistent memory and check the counter value each time request for the object. This approach is trivially feasible to to address rollback attacks but suffer from a significant weakness. The performance of SGX monotonic counter is not well documented [21]. Many prior work did experiments on the *write* performance of monotonic counter and found that writes of counter values to persistent memory is slow (around 10 writes a second). This weakness largely limits the performance in current high throughput KVS systems such as Redis [14] and Apache Zookeeper [10]. Thus, directly applying monotonic counter to preserve freshness is impractical.

4.1.1 Limitations and Future Directions

Though being as a selective feature in SGX architecture, it has strict memory constraints and performs slow during experimental tests [1].

The SGX Monotonic Counter updates take 80-250 ms and reads 60-140 ms. When an enclave needs to persistently store an updated state, it can increment a counter, include the counter value and identifier to the sealed data, and verify integrity of the stored data based on counter value at the time of unsealing. However, such approach may wear out the used non-volatile memory. Assuming a system that updates one of the enclaves on the same platform once every 250 ms, the non-volatile memory used to implement the counter wears out after approximately one million writes, making the counter functionality unusable after a couple of days of continuous use. Even with a modest update rate of one increment per minute, the counters are exhausted in two years. Thus, SGX counters are unsuitable for systems where state updates are frequent and continuous. Additionally, since the non-volatile memory used to store the counters resides outside the processor package, the mechanism is likely vulnerable to bus tapping and flash mirroring attacks [20].

Note that SGX also provides the SGX trusted time feature for checking the timestamp of one stored data record. However, including a timestamp to each sealed data version only allows an enclave to distinguish which out of two seals is more recent, enclaves cannot identify if the sealed data provided by the OS is fresh and latest.

However, the idea of counter increment technique does exist and recent papers [4, 15, 13] have shown that users can indeed benefit from such protection against rollback attacks. Basically, there are two kinds of solutions for counter-based rollback protection. The first technique is *inc-then-store*, where the enclave first increments the counter value and then stores the sealed object together with the incremented value to persistent memory. This approach guarantees that the platform can detect any rollback of stored objects by checking the latest counter value. Even when the system crashes after the rollback, the enclave can restart and check the counter value in the persistent memory, and restore to the updated state (value) of the counter without breaking the protection mechanism. But if the system fails at runtime, the *inc-then-store* can not recover because the counter has a future value while the latest stored object in persistent memory has a smaller counter value. Due to the deterministic increase of the counter, the system cannot recover from system crash.

The second approach is *store-then-inc*, where the enclave first stores the object with an incremented counter value to persistent memory and increments the counter thereafter. This technique can greatly improve the throughput of KVS because the enclave no longer needs to wait for a complete process of incrementing counter and writing the value to persistent memory, instead, the enclave can batch the increment operation of counters and avoid the bottleneck of writing counters to persistent memory ($80 \sim 250$ ms). Another benefit of this technique is that if the system crashes, the system can recover from the failure by referring to the counter value in persistent memory, even in the runtime of protocol. Because the system can detect a future value of counter from persistent memory, by referring to the current state of the counter, the system can check for the missing records and ignore the records with future counter value.

The two techniques are both practical but have different drawbacks which should be taken into consideration in system build up. The drawbacks of *inc-then-store* technique mainly include: (a) it cannot recover from runtime failure and (b) it has relatively slow throughput as each seal operation should wait for the write of counter. For the *store-then-inc* technique, it has higher throughput but may suffer from replay attacks [7].

4.2 ROTE

To overcome the slowness of SGX Monotonic Counters and provide stable persistent rollback attack protections, ROTE [13] is proposed as a distributed trusted counter service based on a consensus protocol.

4.2.1 Overview of ROTE

ROTE is a *inc-then-store* based system that protects against rollback attacks. To overcome the low throughput of monotonic counter increment, ROTE uses a distributed secure counter storage to help verify the version of a target enclave. The intuition behind is simple, that a single SGX-enabled platform is difficult to prevent rollback attacks but many platforms can work together to assist the process of verification. The assisting servers are incentive to do this job as they can also benefit from such protection.

With the assistance of a group of servers, ROTE assumes a strong adversary that can either control the OS of the target platform or any of the assisting platforms. The adversary can break the protection of SGX and even act as a network-level administrator that controls the interactive communication in the network by delaying, replaying or revising network packets. However, as a *inc-then-store* based system, ROTE assumes no tolerance of some of the platform crashes and by default no crash will happen in a protocol run. If crash tolerance is required, then a *store-then-inc* technique is required, and even the system should support both of the techniques to allow users choose by their tolerance of crash.

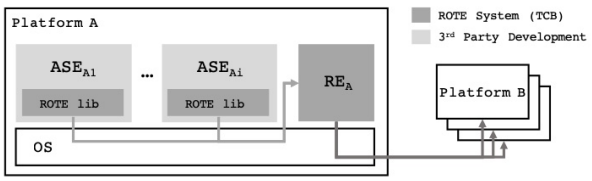

Figure 2: The ROTE system architecture.

The update stage of ROTE works as follows. A client first triggers a counter increment in local enclave, the enclave increments the counter (initialized as zero) in runtime memory, signs the counter value and sends the signed counter value to all assisting servers. Upon receiving the signed counter value, each assisting server updates the value of targeting (client's) counter table in memory and sends back their state of counter value. Note that the value is temporarily stored in memory and not sealed to disk to avoid endless propagation. When the client receives q feedbacks, it compares the value and returns ACKs if the value matches its own. Then, the client can ensure that the version is correct, seal the current counter value and the object to disk.

ROTE also develops a protocol to recover from system reboot/failure, and a distributed mechanism to securely store and compare counter values in remote assisting servers. With a strong network adversary model, ROTE protects against both network partitioning and replay attacks. The update protocol, recover protocol and distributed secure storage mechanism work together to make ROTE a robust KVS system that provides protection against rollback attacks.

4.2.2 System Description

Figure 2 shows ROTE system architecture. Every user application running on platforms matches an Application-Specific Enclave (ASE). The ROTE system provides a Rollback Enclave (RE) and a ROTE library for ASEs as a rollback protection service. The RE maintains a Monotonic Counter (MC), increases it for every ASE update, distributes it to REs running on assisting platforms, and includes the counter value to its own sealed data.

For easier descriptions, we denote n as the number of assisting platforms, f as the number of compromised processors, and u as the tolerance of unreachable assisting platforms when the system proceeds write/read operation. These three parameters have a dependency $n = f + 2u + 1$ to fulfill the data integrity, attestation and freshness of the ROTE system consensus protocols. In the ROTE system, there are three protocols designed for ASE state update, RE restart, and ASE start/read. Specifically, messages transmitted in the ROTE system are all encrypted with respective session keys for data confidentiality concerns. We respectively specify them as follows.

- **ASE State Update Protocol** When an ASE is ready to update its state, it starts the state update protocol. This protocol can be regarded as a modification of the Echo broadcast [].

  1. The ASE triggers a counter increment using the RE.
  2. The RE increments its own MC, and signs the MC.
  3. The RE sends the signed counter to all REs in the protection group.
  4. Upon receiving the signed MC, each RE updates its group counter table kept in the runtime memory without sealing the received data.
  5. The REs that received the counter saves the echo in runtime memory and broadcasts an echo message containing the received signed MC.
  6. After receiving $q = u+f+1$ echos, the RE returns the echos to their senders.
  7. Upon receiving back the echo, each RE finds the self-sent echo in its memory. Then every RE checks if the value from echo, from the group counter table, and from the target RE are equal. If these three values match each other, the RE replies with a final ACK message.
  8. After receiving $q = u + f + 1$ final ACKs, the RE seals its own state together with the MC value to the disk.
  9. The RE returns the incremented ASE counter value. The ASE can now safely perform the state update. The ASE saves the counter value to its runtime memory and seal its state with the counter.

- **RE Restart Protocol** The goal of the protocol is to allow the RE to join the existing protection group, retrieve its counter value and the MC values of the other nodes. It supports at most u REs restart simultaneously.

  1. Reset cryptology keys and update system configuration informations
  2. The RE queries the OS for the sealed state.
  3. The RE unseals the state (if received) and extracts the MC.
  4. The RE sends a request to all other REs in the same protection group to retrieve its MC.
  5. The assisting REs check their group counter table. If the MC is found, the enclaves reply with the signed MC. Additionally, the target RE receives the complete table all signed MCs from assisting REs.
  6. When the RE receives $q = u + f + 1$, where $q \geq n/2$ with at least $f + 1$ counter values not zero, responses from the group, it selects the maximum value and verifies the signature. For each assisting RE, the target RE picks the highest MC and updates its own group counter table with the value. If the obtained counter value equals to the unsealed data, the unsealed state can be accepted.
  7. The RE stores and seals the updated state to both persistent and runtime memory.

- **ASE Start/Read Protocol** When an ASE needs to verify the freshness of its state, it performs this protocol.

1. The ASE queries the OS for the sealed data.
2. The ASE unseals the state (if received) and obtains a counter value from it.
3. The ASE issues a request to the local RE to retrieve its latest ASE counter value.
4. To verify the freshness of its runtime state, the RE performs the steps 4-6 from the RE Restart protocol, to obtain the latest MC from the network. If the obtained MC does not match the MC residing in the memory, the RE must abort and be restarted. If the values match, the current data is fresh and the RE can continue normal operation.
5. If all verification checks are successful, the RE returns a value from the local ASE counter table.
6. The ASE compares the received counter value to the one obtained from the sealed data.

Notice that in ROTE system defines a required quorum with size $q = u + f + 1$ for secure consensus. The reason behind q value is that if the counter is successfully written to $q = f + u + 1$ nodes, there always exists at least $u + 1$ honest platforms in the group that have the latest counter value in the memory. Because counter reading requires the same number of responses, at least one correct counter value is obtained upon reading. If the quorum cannot be satisfied in either the state update protocol or any counter retrieval, ROTE turns to halt and try to perform the same operation again.

4.2.3 Limitations and Future Directions

As is mentioned above, ROTE leverages *inc-then-store* counter increment technique as the foundation to defend against rollback attacks. The bottleneck of such technique still exists in ROTE: the crash during protocol run can totally ruin the system and prevent the system from recovery.

We propose two future directions for further improving ROTE's trust model by enabling crash recovery between sealing counter values and sealing objects to disk. Our first proposal is to ensure the atomicity of the *write_counter()* function and *seal_object()* function. Currently in ROTE, the crash may happen between the two functions, contributing to a counter with a future value and making the KVS unrecoverable. If we ensure the atomicity of the two functions, then the counter sealing and object sealing will succeed or fail at the same time, and the scenario where the counter has a future value will not appear.

Our second approach is to backup the counter value in disk, right before the verification of received counters in ROTE. In that case, if the crash happens after the sealing of counter (before sealing the object), the KVS can still recover to the older version of counter value by referring to the log.

4.3 SPEICHER

The SPEICHER system [4] is another KVS that provides rollback protection. It mainly has two differences compared to ROTE introduced above. First, SPEICHER is a typical *store-then-inc* based system that first stores object with pre-incremented counter values into persistent disk memory and increments counter value thereafter. The second difference lies in the architecture. The ROTE system uses several assisting severs organized by a trusted group manager while the SPEICHER system uses only localized protection.

4.3.1 Overview of Speicher

SPEICHER exports a Key-Value (KV) interface backed by Log-Structured Merge Tree (LSM) for supporting secure data storage and query operations. SPEICHER enforces these security properties on an untrusted host by leveraging shielded execution based on a hardware-assisted trusted execution environment (TEE)—specifically, Intel SGX. However, the design of SPEICHER extends the trust in shielded execution beyond the secure SGX enclave memory region to ensure that the security properties are also preserved in the stateful (or non-volatile) setting of an untrusted storage medium, including system crash, reboot, or migration. SPEICHER ensures data freshness by applying asynchronous trusted counters.

The update stage of SPEICHER works as follows. To update a record or add a new record in KVS, SPEICHER first inserts the record into a runtime data structure named as *memtable*. Then the counter is incremented to the value in the stored record. SPEICHER is a fully asynchronous system that decouples the increment operation and the store operation, which means that SPEICHER batches the increment operation after a *expected time*. During the expected time, the user can wait for the increment of counter to be stable, and can abort the update operation if time expires. This mechanism prevents users from potential rollback attacks. In case a system crashes between a store operation and increment operation, the stored record will become invalid as the counter value is beyond the older version of the counter stored in disk.

4.3.2 System Description

In this section, we primally introduce the rollback protection measurements of SPEICHER. Firstly, SPEICHER defines an Asynchronous Trusted Monotonic Counter, which fundamentally takes advantage of the lag in the sync operations in modern KV stores. Furthermore, SPEICHER leverages log file with a footer containing cryptographic hash to maintain and verify the data freshness.

- **Asynchronous Trusted Monotonic Counter**

  In order to protect the system from rollback attacks, a trusted counter whose value is stored alongside with the sensitive data is needed. To overcome the limitations of SGX counters, an Asynchronous Monotonic Counter (AMC) is proposed based on the observation that many contemporary KV stores do not persist their inserted data immediately. This allows AMC to defer the counter increment until the data is persisted without loosing any availability guarantees. As a result, AMC achieves 70K updates per second in the current implementation.

  AMC provides an asynchronous increment interface, because it takes a while since the counter value is incremented until it becomes stable, which means the

counter value cannot be rolled back without being detected. At an increment, AMC returns three pieces of information: the current stable value, the incremented counter value, and the expected time for the value to be stable. Due to the expected time and the controller having to be re-authenticated after a shutdown, the client only has to keep the values until the stable time has elapsed, to prevent any data loss in case of a sudden shutdown.

- **Log files**

  SPEICHER adapted two different log files to ensure the desired security properties, *i.e.,* (a) WAL for persisting inserted KV pairs until a top-level compaction; and (b) the Manifest to keep track of live files.

  Regarding WAL, every put operation appends a record to the current WAL. This record consists of the encrypted KV pair, and an encrypted trusted counter value for the WAL at the moment of insertion, and a cryptographic hash over both. Since the records are only appended to the WAL, SPEICHER can use the trusted counter value and the hash value to verify the KV pair, and to replay the operations in a restore event.

  The Manifest is similar to the WAL; it is a write-append log consisting of records storing changes of live files. We use the same scheme for the Manifest file as we do for the WAL.

With these two features, SPEICHER extends the trust in shielded execution beyond the secure enclave memory region to ensure that the security properties are also preserved in the stateful setting of an untrusted storage medium.

4.3.3 Limitations and Future Directions

We claim that the design of SPEICHER does improve rollback protection and is robust to restore at any time of protocol execution. As an SGX based KVS, SPEICHER also considers the limited size of enclaves (around 128MB), and designed a new data structure outside the enclave to preserve the integrity. Unfortunately, SPEICHER may have failed to breakdown the overhead by each SGX components. For example, the asynchronous trusted counter mechanism batches arrived counter increment operations and we expect to see the breakdown to see how this mechanism helps improve the throughput.

5. EVALUATION PLAN

To evaluate the effectiveness of the above-mentioned protection mechanisms and see how our proposed ideas may help improve the threat model, we plan to analyze the following features in the future.

Performance of Update and reboot. Either a protection mechanism is used *inc-then-store* or *store-then-inc*, it's significant to see the practicality in a large-scale KVS. In other word, by integrating the rollback protection, the expected performance of update stage, including both the I/O throughput and end-to-end communication latency should be well studied. If there is moderate overhead compared to the original design (e.g., Redis or Zookeeper), then the plan is considered to be both cost-efficient and protection-effective. However, if the update stage cost significant overhead, then it is impractical to be deployed in real world.

The recovery of KVS (i.e., restore) from a system crash, reboot or failure should be measured as well. As the integrity of SGX-based mechanism is usually assumed to be immune to parties outside the enclave, the only way to break the security property of these systems is to completely control all the parties engaged in version check. Thus the defense of such vulnerability might take extra overhead. We should evaluate how the recovery module effects the effectiveness of KVS systems, such as manually reboot the system or cut off power.

SGX overhead. The monotonic counter is slow, as introduced in both Background and §3. We need to evaluate the overhead of using SGX isolation. More precisely, we should breakdown the I/O overhead of entering/exiting SGX enclaves, the latency of counter mechanism (e.g., Speicher's asynchronous counter increment), and the *Read/Write* throughput with limited enclave size.

6. CONCLUSION

In this report, we discussed the freshness problem in state-of-art Key-Value Store (KVS) systems. Typically, so-called rollback attackers might return an older version of records to users. The use of Intel SGX helps nullify part of this problem, including the confidentiality of data and integrity of execution code. However, directly apply SGX cannot solve this problem because SGX also fails to check the latest version of records. Some prior works develop different counter increment based techniques to protect against rollback attacks, including ROTE [13], Speicher [4], EnclaveDB [15] and so on. These systems either incurring moderate overhead or being effective in a distributed setting, with some drawbacks in either evaluation or crash tolerance. Thus, we propose some improvements on the design and evaluation of existing work and plan to implement and evaluate our prototype.

7. REFERENCES

[1] F. Alder, A. Kurnikov, A. Paverd, and N. Asokan. Migrating sgx enclaves with persistent state. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 195–206. IEEE, 2018.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013.

[3] A. Atamli-Reineh and A. Martin. Securing application with software partitioning: A case study using sgx. In *International Conference on Security and Privacy in Communication Systems*, pages 605–621. Springer, 2015.

[4] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. {SPEICHER}: Securing lsm-based key-value stores using shielded execution. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 173–190, 2019.

[5] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: Gpu-accelerated incremental storage and computation. In *FAST*, volume 14, page 14, 2012.

[6] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure:{SGX} cache attacks are practical. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[7] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. Securekeeper: confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, page 14. ACM, 2016.

[8] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[9] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[11] V. Karande, E. Bauman, Z. Lin, and L. Khan. Sgx-log: Securing system logs with sgx. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 19–30. ACM, 2017.

[12] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of linux file system evolution. In *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, pages 31–44, 2013.

[13] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. {ROTE}: Rollback protection for trusted execution. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1289–1306, 2017.

[14] M. Paksula. Persisting objects in redis key-value database. *University of Helsinki, Department of Computer Science*, 2010.

[15] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using sgx. In *EnclaveDB: A Secure Database using SGX*, page 0. IEEE, 2018.

[16] C. Rozas. Intel® software guard extensions (intel® sgx), 2013.

[17] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. *HotCloud*, 9(9):3, 2009.

[18] M. Seeger and S. Ultra-Large-Sites. Key-value stores: a practical overview. *Computer Science and Media, Stuttgart*, 2009.

[19] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*, 2017.

[20] S. Skorobogatov. The bumpy road towards iphone 5c nand mirroring. *arXiv preprint arXiv:1609.04327*, 2016.

[21] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 875–892, 2016.

[22] R. Strackx and F. Piessens. Developing secure sgx enclaves: New challenges on the horizon. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, page 3. ACM, 2016.

[23] J. Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.