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

This paper provides a sample of a LaTeX document which conforms to the formatting guidelines for ACM SIG Proceedings. It complements the document *Author's Guide to Preparing ACM SIG Proceedings Using LaTeX2$_\epsilon$ and BibTeX*. This source file has been written with the intention of being compiled under LaTeX2$_\epsilon$ and BibTeX.

The developers have tried to include every imaginable sort of "bells and whistles", such as a subtitle, footnotes on title, subtitle and authors, as well as in the text, and every optional component (e.g. Acknowledgments, Additional Authors, Appendices), not to mention examples of equations, theorems, tables and figures.

To make best use of this sample document, run it through LaTeX and BibTeX, and compare this source code with the printed output produced by the dvi file.

1. INTRODUCTION

This is introduction.

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 [].

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 [], one of the most popular security architectures is Intel Software Guard Extensions (SGX) []. 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 [], 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 []. 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

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

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 []. 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 **??** primally utilize these three operations in SGX to provide rollback attack protections.

2.1.3 SGX Counter

Intel has recently added support for monotonic counters (MC) [] 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 [], 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 **??** 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 **??**.

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 []. 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 []. 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 [].

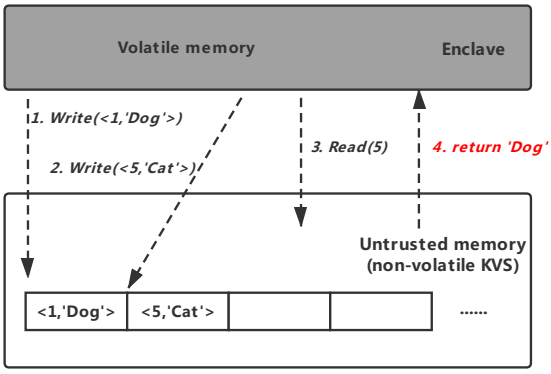

Figure 1: An example of rollback attack towards KVS on SGX-enabled host. The malicious OS returns an older version of value in KVS and the trusted enclave (in gray) cannot detect it. The records should be sealed/unsealed but we omit these operations for simplicity.

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 []. 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 [].

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 []. 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 [] and Apache Zookeeper []. Thus, directly applying monotonic counter to preserve freshness is impractical.

4.1.1 Limitations

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

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 [].

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 [] 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 [].

4.2 ROTE

To overcome the slowness of SGX Monotonic Counters and provide stable persistent rollback attack protections, ROTE [] 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.

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.

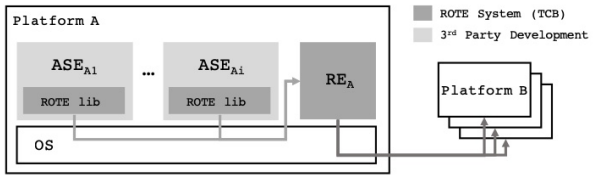

Figure 2: The ROTE system architecture.

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 Protocols

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

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 [] 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.

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.

5. EVALUATION PLAN

This is evaluation part.

6. CONCLUSION

This is conclusion.

7. REFERENCES