Revisiting Rollbacks on Smart Contracts in TEE-protected Private Blockchains

Chen Chang Lew*
ETH Zürich
lewchenchang@gmail.com

Christof Ferreira Torres
ETH Zürich
christof.torres@inf.ethz.ch

Shweta Shinde
ETH Zürich
shweta.shinde@inf.ethz.ch

Marcus Brandenburger
IBM Research
bur@zurich.ibm.com

Abstract—**Blockchain technology offers decentralized security but fails to ensure data confidentiality due to its inherent data replication across all network nodes. To address these confidentiality challenges, integrating blockchains with Trusted Execution Environments (TEEs), such as Intel SGX, offers a viable solution. This approach, by encrypting all data outside the SGX enclave and making them unrecognizable to untrusted network nodes, ensures secure processing of data and computations within TEEs. Fabric Private Chaincode (FPC), an enhancement of Hyperledger Fabric, demonstrates this integration by securing smart contracts in enclaves, thereby enhancing confidentiality. However, FPC's reliance on states stored on the blockchain introduces vulnerabilities, especially to rollback attacks. This work provides a detailed analysis of rollback attacks in FPC, evaluates existing protection mechanisms, and proposes a solution: a Merkle Tree approach implemented in an FPC application named Secret Keeper. Through experimental validation, this solution shows significant security enhancements against rollback attacks within FPC contexts.**

1. Introduction

Blockchain technology, a decentralized ledger system, has revolutionized how transactions are recorded, offering transparency, integrity, and resilience. However, its inherent transparency poses significant privacy and confidentiality challenges, undermining the privacy of its users. Smart contracts have further extended blockchain's capabilities, automating transactions without the need for intermediaries. Yet, a smart contract cannot easily work with encrypted data. To address these privacy concerns, Trusted Execution Environments (TEEs), such as Intel SGX [1], [2], have emerged as a solution [3]. By integrating smart contracts with TEEs, it is possible to execute these contracts in a secure execution space called an enclave, preserving the confidentiality of the data while maintaining the blockchain's integrity and consistency.

Hyperledger Fabric Private Chaincode (FPC) [3] utilizes TEEs to enhance the privacy and security of smart contracts on the Hyperledger Fabric platform [4]. Despite these advancements, FPC still suffers from rollback attacks, where adversaries manipulate the system by replaying old data. The authors of the original FPC paper [3] acknowledge the need for rollback protection but merely propose a prototype solution with significant limitations, thereby highlighting that there is an area for improvement.

Work done while at IBM Research - Zurich.

This paper delves into potential solutions within the FPC framework. The original FPC documentation suggests a strawman approach of consolidating all values under a singular state, a method that proves secure but inefficient and unscalable. It also discusses a Trusted Ledger Enclave solution, which was excluded from the FPC RFC [5] due to high maintenance costs and suboptimal performance. We propose a Merkle tree-based solution that retains up to 95% of FPC's original throughput without rollback protection, demonstrating a minor compromise in efficiency for significantly improved security.

Our contributions are multifaceted, extending from theoretical exploration to practical application:

- **Feasibility Analysis:** We assess the practicality of existing rollback protection mechanisms from literature in the context of the FPC, considering their efficiency and effectiveness in Section 2.6.
- **Solution Prototyping and Implementation:** We implement the Single Key-Value Storage and Trusted Ledger Enclave solutions as described in existing work in Section 2.7, and propose and implement a new solution against rollback attacks leveraging Merkle trees as presented in Section 3.
- **Experimental Evaluation:** We evaluate the security, size of the Trusted Computing Base (TCB), and performance implications of our proposed solution in Section 4, comparing it with existing work and offering insights into their practicality and deployability.

2. Problem Description

We introduce the concept of TEEs and FPC first, and then motivate the need for protection against *selective rollback* attacks.

2.1. Trusted Execution Environments

Trusted Execution Environments (TEEs) provide a secure space within processors, like Intel SGX [1], [2], AMD SEV-SNP [6], and ARM TrustZone [7], ensuring the confidentiality and integrity of the code and data executed within. *Remote Attestation*, a crucial feature, enables customers to verify that their binary code is indeed executing within the TEE, as asserted by the cloud provider. Intel SGX, for example, creates protected memory regions, or enclaves, safeguarding data from higher privilege levels and external tampering [8], [9]. Additionally, features like *data sealing* [10], a host-dependent capability, allows for the encrypted storage of enclave data, preserving its integrity for later retrieval. Despite their robustness against unauthorized access, TEEs face challenges from physical [11], [12] and side-channel [13]–[18] attacks, with continuous advancements in defense mechanisms [19], [20] enhancing their security.

2.2. Hyperledger Fabric

Hyperledger Fabric, a key project of the Hyperledger consortium hosted by the Linux Foundation, is an open-source, *permissioned blockchain* framework that supports smart contracts, known as *chaincode* [4]. Unlike permissionless blockchains, such as Bitcoin [21] and Ethereum [22], Fabric introduces a unique execute-order-validate architecture to enhance security, privacy, and efficiency. *Execution*: Transaction execution occurs without state updates in a process also referred to as chaincode simulation. *Ordering*: A dedicated service sequences transactions into blocks, ensuring ledger consistency across the network. *Validation*: Transactions are validated against specified policies, with only the valid transactions being committed to the blockchain and their state updates applied.

Fabric's architecture is bolstered by its *endorsement policy* [23], which specifies the required peers for transaction approval, thereby ensuring a transaction's validity before it is committed to the ledger. This policy is a cornerstone of Fabric's modular and permissioned design, which not only facilitates scalability and robustness but also allows for the development of confidential applications by involving identifiable members and supporting traditional Byzantine-Fault Tolerant (BFT) consensus mechanisms. Additionally, Fabric's adaptability is highlighted by its support for smart contracts developed in popular programming languages such as Go [24], Node.js [25], and Java [26]. This feature positions Fabric as an ideal platform for creating permissioned blockchain applications that require a high degree of customization and privacy.

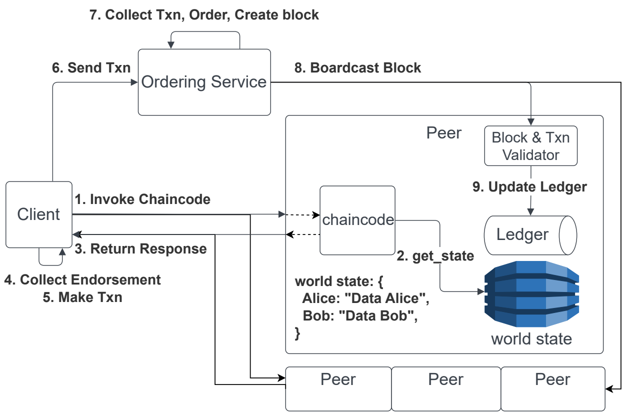


Figure 1: Hyperledger Fabric architecture, highlighting the transaction flow when a client invokes some chaincode.

A Hyperledger Fabric network comprises multiple peers, clients, and an ordering service, that collaboratively maintains a distributed ledger. Each peer holds the ledger of transactions and a "world state", a key-value database that reflects the latest status of all assets for swift retrieval without reviewing the entire transaction history. Figure 1 depicts the execution flow when a client invokes some chaincode: (Step 1) A client sends an `invoke chaincode` command to the peer nodes, which execute the corresponding chaincode; (Step 2) If needed, a peer node calls `get_state()` to fetch necessary data from the world state during chaincode execution; (Step 3) The peers return a cryptographically signed response (also called *endorsement*) to the client; (Step 4) The client gathers the endorsements from multiple peers (according to the endorsement policy); (Step 5) Once enough endorsements are collected, the client creates a transaction and (Step 6) sends it to the ordering service. (Step 7) The ordering service accumulates transactions, sequences them, and crafts blocks of transactions; (Step 8) A newly created block is broadcasted to all peers using a gossip protocol [27]; (Step 9) Each peer validates the transactions in the received block and appends them to their ledger. Note that only valid transactions update the world state accordingly. For more details on Hyperledger Fabric, please refer to [4], [23].

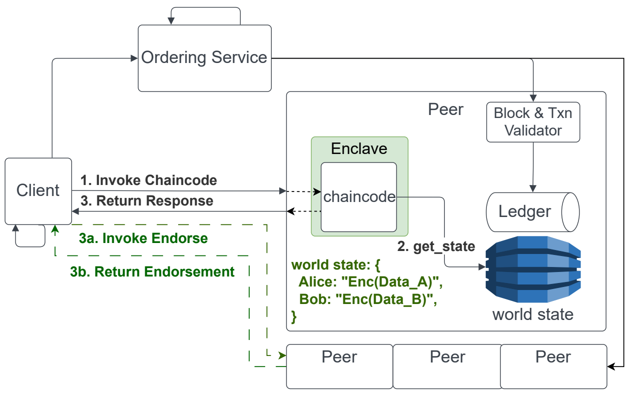


Figure 2: FPC architecture, emphasizing the execution flow when a client invokes chaincode within an enclave.

2.3. Fabric Private Chaincode

Hyperledger Fabric Private Chaincode (FPC) [3] enhances Hyperledger Fabric by ensuring data confidentiality and integrity through integration with Intel SGX enclaves, termed as *enclave chaincode*. This setup secures transaction data use, encrypts data transfers between clients and chaincodes, and encrypts data stored in the world state. The FPC client SDK enhances this security by transparently encrypting chaincode invocation requests, ensuring that data remains hidden from malicious peers. Additionally, FPC's integration of remote attestation allows enclaves to verify their authenticity to external entities, like Fabric clients or peers, ensuring interactions only occur with verified enclaves running the expected chaincode, thus maintaining a high level of privacy and security in transactions. This architecture is particularly aimed at applications requiring strong confidentiality, such as privacy-centric analytics or secret ballot voting. In traditional Fabric setups, application data is visible to all peers, compromising the privacy of the application data.

As depicted in Figure 2, FPC integrates into the Fabric architecture without major modifications. The primary difference is the encapsulation of the chaincode within an enclave, encrypting world state values. The execution flow starts with an enclave-invoked request, with responses sent back to the client. (Step 3a) The client seeks endorsements from designated peers, (Step 3b) which verifies the transaction's validity by checking the enclave signatures. After obtaining the necessary endorsements, the transaction is sent to the ordering service, following the same steps as in the standard Fabric process.

2.4. Threat Model

We assume a Hyperledger Fabric network comprising multiple clients and peers equipped with Intel SGX. The peers are considered to be malicious and may collude with a malicious client. Peers have full control over the system's operation, the ledger, and the world state but cannot tamper with the execution of the chaincode residing inside the FPC enclave. FPC encrypts and authenticates all the messages between the *FPC client* and the *chaincode enclave*, and all the data stored on the ledger and world state. Malicious peers cannot decrypt the data nor manipulate the data without detection. Moreover, a peer can still observe the data flow to and from the enclave, although they cannot directly access or alter the enclave's internal code and data. The system is considered secure if malicious peers and clients cannot glean new information as they act honestly, with the assumption that the Fabric blockchain maintains data integrity and consensus against a minority of malicious peers. External threats, such as vulnerabilities within enclaves, side-channels, and denial-of-service attacks are out of the scope of this paper.

2.5. Rollback Attack

A rollback attack on a TEE is a security exploit that targets the state persistence of TEE applications. An attacker may try to manipulate the TEE to reuse old data or stale state, thereby "rolling back" the application to a valid but outdated state. In our TEE application (i.e., chaincode), the state is a key-value store hosted outside the TEE under the control of the peer node. We consider two forms of rollback attacks: *full rollback* and *selective rollback*. In a full rollback attack, all key-value pairs in the world state are from the same version, but not the latest version that the peer has seen. Conversely, a selective rollback attack targets a subset of key-value pairs in the world state, resulting in a mix of different versions. In our threat model, our primary focus is on protecting against breaches of confidentiality rather than integrity, which is already ensured by Fabric's properties [4]. Previous work [3] has shown that a full rollback attack is equivalent to the situation where a peer node does not have the latest version of the ledger, for instance, due to network delays. In that case, the attacker do not gain any new information and the execution will not lead to a valid transaction that causes a state update. However, it is the selective rollback attack that poses a more potent threat, as it may undermine the confidentiality of the system. Specifically for FPC, selective rollback attacks are an issue. Considering the case when the chaincode enclave calls `get_state()` to read data from the world state. A malicious peer could

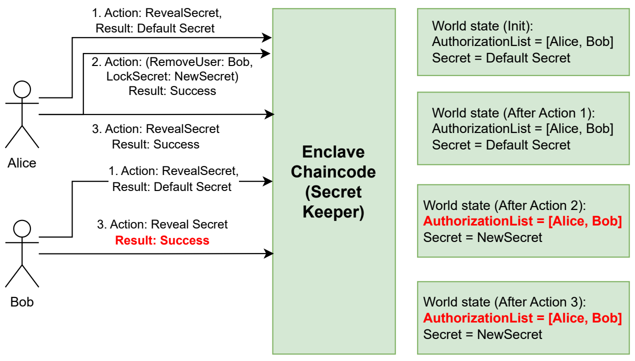


Figure 3: Example of a rollback attack on Secret Keeper.

return the mix of old and recent state to the enclave, compromising the confidentiality of the system.

We illustrate rollback attacks on FPC with a use-case scenario called *Secret Keeper*, where some chaincode implements a vault application. Authorized users can store and retrieve secrets, and modify access control to the vault. The *Secret Keeper* maintains two key-value pairs in the world state: `AuthorizationList` contains the public keys of users permitted to interact with the vault, whereas `Secret` holds the secret information. To better illustrate its functionality, a demo video is available on Loom[1].

Suppose a user has been removed from the vault and a new secret has been stored. When the user tries to access the new secret, the enclave loads `AuthorizationList` using `get_state()` and checks whether the user is in the list. In this example, the user will be rejected. However, the user may collude with a malicious peer to access the new secret by mounting a rollback attack as illustrated in Figure 3. The user invokes the enclave, but this time `get_state()` will return an old version of the `AuthorizationList` that still includes the user. Thus, the enclave will call `get_state()` which returns `Secret` which contains the latest secret. Even though the data is encrypted and authenticated, the FPC enclave cannot verify the freshness of the received data. The user passes the authorization checks and receives the secret. This violates the integrity and even breaks the confidentiality of the *Secret Keeper* application. We show the rollback attack in a demo video available on Loom[2].

2.6. Related Work and Analysis

One solution to protect against rollback attacks is to use a Monotonic Counter to achieve State Continuity. Monotonic counters [28] allow to detection of state reversions through persistent, increment-only values. However, as acknowledged by other works [29]–[31], this approach faces performance limitations, and write operation capacity, and may not be available on every TEE platform. Moreover, due to the *execute-order-validate* architecture of Fabric, a client may choose not to submit the transaction to the ordering service, which may result in an inconsistency between the ledger and the enclave's monotonic counter.

1. Secret Keeper Demo: https://www.loom.com/share/e3dca62f8df849229e2c6414fd374289?sid=473acda1-92ac-4ad6-89c4-ddb2f5786dd5

2. Rollback Attack Demo: https://www.loom.com/share/e540bf6395f94ab8ba547bd43942d063?sid=b32de7ea-e5d7-43ef-9e06-bf11b3cfc6ff

Distributed systems like ROTE [30] propose data consistency through multi-enclave communication to maintain a consistent view. However, this approach does not align with the design of FPC as inter-enclave communication is not suitable for Fabric. EnclaveDB [32] and Nimble [33] suggest storing sensitive data in SGX enclaves and using fault-tolerant storage to detect rollback attacks. These approaches appear to be impractical and expensive for FPC considering the large world state maintained inside the enclave. Furthermore, verification methods, including formal verification tools [34] and libraries [35], offer rollback protection by validating state continuity or logging enclave operations, yet they demand precise rule crafting and increase the burdens of the chaincode developers. Lastly, addressing the challenge of side-channel attacks is critical for enclaves. Software solutions such as compiler support for constant-time implementations can limit the side-channel leakage [36]. These enhancements aim to preserve security properties defined at the source code level through to the binary level, mitigating vulnerabilities introduced by compiler optimizations [37]. However, these approaches often lack support for a broad range of processor architectures, typically require specialized knowledge for effective use, such as secure coding practices, and may not fully support modern processor features or the latest programming language standards. We leave side-channels out-of-scope for this paper; future works can address this threat for FPC.

2.7. Existing Solutions

A naive approach to tackle the rollback problem is the *Single Key-Value Storage (SKVS)* solution as mentioned in the FPC RFC [38]. In this approach, all key-value pairs are encapsulated and stored with a single call to `put_state()`. During execution, the enclave must load the entire state before it can access individual key-value pairs. While this approach prevents the attack as explained in Section 2.5, applications with large states and multiple writers will experience bad performance, as the use of a single key-value pair will cause transactions to fail due to concurrent write issues. Since the Fabric architecture operates on a three-phase execute-order-validate model. Transactions are rejected during the validation phase if, for example, the read-write set conflicts with the current world state. This would occur if two transactions in the same block attempt to modify the same key in the world state. Given that the SKVS solution involves only one key, if there is one write transaction in a block, subsequent transactions will experience read or write conflicts, leading to rejection. From scalability as the data footprint grows, leading to inefficiencies in handling multiple write transactions within the same block. The implementation of the SKVS solution can be found on GitHub [39].

Alternatively, the authors of FPC [3] proposed the *Trusted Ledger Enclave (TLE)*, designed to preserve the integrity and the consistency of the ledger. The TLE maintains integrity metadata for each key-value pair, allowing for verification of data versions obtained from peers. Each time the enclave calls `get_state()` on the peer, it simultaneously requests `get_meta()` from TLE. This allows the chaincode enclave to check whether the data obtained from the peer matches the version known to TLE, thus preventing potential rollback attacks.

At first glance, TLE seems an ideal solution against rollback attacks. However, the first version of the FPC RFC [38] abandoned this solution due to high implementation complexity as it duplicates the peer validation logic inside an enclave, leading to code redundancy and maintenance difficulties. The implementation of the TLE solution can be found on GitHub [40], [41].

3. Our Solution: The Merkle Tree Approach

In this section, we propose a new solution called *Merkle Tree Apporach (MTA)* to protect against rollback attacks for FPC by employing Merkle trees [42]. Inspired by existing client-side verification techniques [31], our solution enhances data security and integrity through efficient summarization and verification of transaction data.

At a high level, our solution works as follows. Each peer maintains a Merkle tree of their world state. Before clients invoke the chaincode enclave, they collect the latest Merkle root from multiple peers and send them together with the transaction to the enclave. When the enclave receives the invocation, it validates the consistency of the received Merkle roots and executes the chaincode. During chaincode execution, the enclave can verify that each key-value pair received via `get_state()` is consistent with the world state of the other peers with the help of the Merkle path and the Merkle root.

An important consideration is whether the enclave should store the Merkle root and update it after each chaincode invocation. This approach faces challenges as the execution of chaincode in Hyperledger Fabric happens speculatively, that is, the transaction is executed before ordering and validation without any guarantee that the resulting state change is actually applied to the world state [4]. For example, a client may drop the endorsement and never submit the transaction. If the enclave creates a new Merkle root, it does not know when the corresponding transaction gets committed and the state change becomes live. The only way to learn this is to track the ledger, which is the core idea of the TLE approach.

3.1. Architecture

We introduce the *Merkle Tree Component* and the *Key Enclave* to the FPC architecture as depicted in Figure 4. The Merkle Tree Component maintains a Merkle tree of the world state for each peer. It provides operations to retrieve the Merkle root and a Merkle path for a given key-value entry. Messages containing the Merkle root are cryptographically signed by the peer. The Merkle tree is updated with every transaction committed to the ledger.

The Key Enclave is co-located with the chaincode enclave and stores the peers' public keys and endorsement policies. It retrieves the configuration transaction from the ledger, verifies them, and stores the public keys of the peers and the endorsement policy for each chaincode. The Key Enclave provides operations to the chaincode enclave to fetch these public keys and to verify messages signed by the peers. Note that the key enclave stores all the data in its memory, although it might suffer from a rollback attack on the enclave state. Preventing such attacks can be straightforward, utilizing a monotonic counter, especially given the assumption that configuration changes are infrequent and have low throughput. Alternatively, any of the solutions discussed in Section 2.6 could be applied to mitigate this risk.

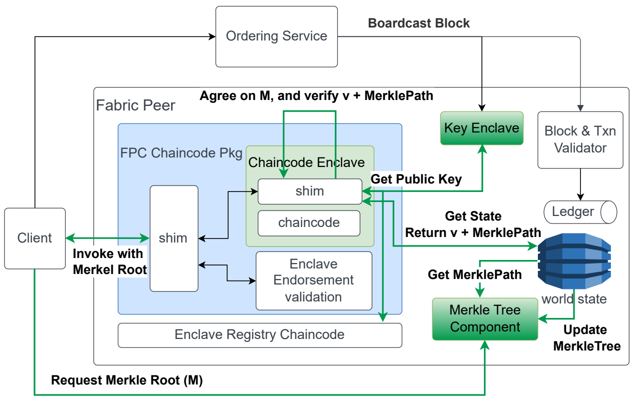


Figure 4: Architecture of our Merkle Tree Approach. The Key Enclave and the Merkle Tree Component represent additions to the system.

3.2. Protocol

Figure 5 illustrates the protocol of our solution as integrated into the standard FPC transaction flow. When a client invokes some chaincode, it starts by first (Step 0) querying multiple peers to retrieve the Merkle root M representing their current world state, and their latest ledger height. (Step 1) The collection of Ms and ledger height information are relayed to the chaincode enclave as a parameter with the transaction invocation. (Step 1a) The enclave then verifies the peer signatures on each M with the help of the Key Enclave which selects a root consistent with the FPC's endorsement policy and starts the execution of the chaincode with that ledger height. (Step 2a) Every subsequent `get_state()` call to a peer returns a pair: the value and its corresponding Merkle path. (Step 2b) The enclave verifies this path against the initially chosen Merkle root, and any inconsistency results in an immediate halt. If the verification succeeds, we continue with the standard FPC transaction flow as described in Section 2.3. Finally, (Step 10) the Merkle Tree is updated accordingly containing any changes to the ledger and world state. The implementation of the `MerkleTree` solution is available on GitHub [43], [44].

4. Evaluation

Next, we evaluate and compare our *Merkle Tree Approach (MTA)* with the *SKVS* and *TLE* solutions as described in Section 2.7, focusing on security analysis, size of the Trusted Computing Base (TCB), and performance impact under various conditions.

4.1. Security Analysis

As mentioned in Section 2.4, malicious peers and users aim to compromise the system's confidentiality. This section will illustrate how each solution we implemented

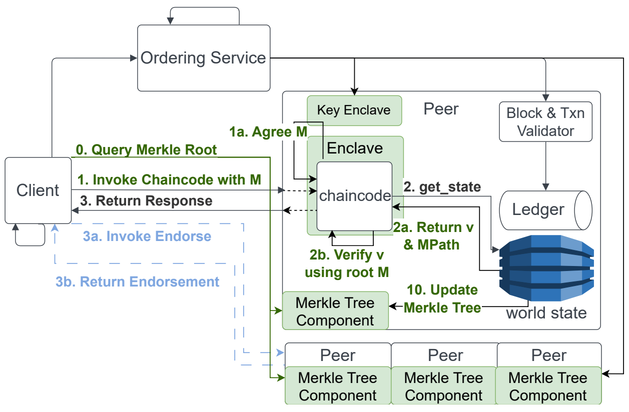


Figure 5: The execution flow within our Merkle Tree Approach when a client invokes some chaincode.

protects against the rollback attacks. The implementation of malicious peers can be found on Github [45].

Single Key-Value Storage (SKVS): We store all the data under a single key for every distinct version. As a result, malicious peers and users cannot induce a version mismatch and thereby compromise confidentiality. Even if a malicious peer successfully rollbacks the world state, users will not glean any new information. A demo video is available on Loom[3].

Trusted Ledger Enclave (TLE): The TLE verifies all transactions directly from the trusted orderer and thereby maintains its own view of the ledger. This ensures that the TLE always possesses the latest metadata for each key in the world state. If a malicious peer attempts to send outdated data to the enclave, the enclave recognizes the discrepancy in the value's metadata and aborts the execution. In a scenario where a malicious peer controlling the network, blocks the blocks that the TLE receives from the orderer and then sends old data to the enclave, the query will still execute successfully. However, malicious users will not acquire any new information. A demo video is available on Loom[4].

Merkle Tree Approach (MTA): In our model, we assume that the majority of peers ($> 50\%$) are honest. The client must relay sufficient Merkle roots to the enclave, enabling it to determine the correct Merkle root. This ensures that the enclave always decides based on the Merkle root. If a malicious peer attempts to supply data from an alternative version, it cannot forge a Merkle path leading to the same Merkle root due to the hash collision properties of SHA256. However, there's a potential scenario where malicious peers could coerce the enclave into accepting an older version of the Merkle root (e.g., if some honest peers are slow), which is equivalent to rolling back to an older version of the ledger. Again, malicious users will not glean any new information. A demo video is available on Loom[5].

3. SKVS Rollback Demo: https://www.loom.com/share/4790cd0cba9e4433879083b41158d42d?sid=a0f367e7-dc49-4065-926d-df7b2fbf55a1

4. TLE Rollback Demo: https://www.loom.com/share/4e8814304bcd49c9860861be524febbd?sid=c64b4cdb-0ae3-4125-93af-f369349f11f8

5. MTA Rollback Demo: https://www.loom.com/share/632e389fc03540c4ae1de35440ccbbd1?sid=29d97578-61f5-43a8-b791-6ee5cce450ec

TABLE 1: Lines of changes for each solution's codebase.

Solution	TCB Changes	Non-TCB Changes
`SKVS`	141	0
`TLE`	6,862	0
`MTA`	534	2,105

4.2. Trusted Computing Base (TCB)

Table 1 summarizes the distribution of code changes for each solution. For the SKVS solution, 141 lines were added while the TLE solution adds a total of 6,862 lines to the TCB. MTA only incorporates 534 lines into the TCB. An additional 2,105 lines are added outside of the TCB for the Merkle Tree Component and the communication between clients. These components are deemed vulnerable to compromise by malicious actors and, therefore, are not included within the TCB.

4.3. Performance Impact

In our performance evaluation, we focus on assessing the impact on throughput and latency of all three solutions under different conditions. We separate our evaluation into four specific experiments. We examine the throughput and latency while (a) altering the number of clients, (b) varying network delays on these parameters, (c) changing the number of peers, (d) in a real-world scenario (*ExtSecretKeeper*) to ensure practical applicability.

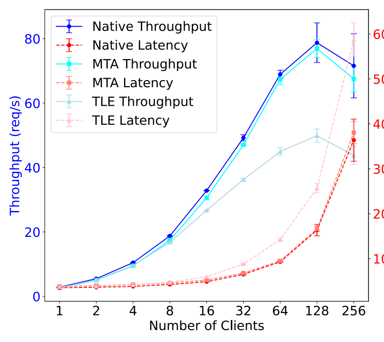
The experiments were conducted on an Apple MacBook Pro 13.3.1, equipped with an Intel Core i7 CPU, and 32 GB of memory. Note that SGX hardware was not available on this machine, necessitating the running of all experiments in simulation mode. However, we posit that this setup does not compromise the validity of our results. We believe our solution primarily introduces additional communication overhead between the client and the peer at the network level, making the absence of SGX hardware inconsequential for our evaluations. To simulate realistic network conditions in our local experiments, we introduced artificial delays to mimic the Round Trip Time (RTT) between a client, the orderer, and an FPC peer, assuming they are on separate servers. We leave the evaluation in a distributed setup for future work.

We fork the FPC's branch *go-support-preview* [5], which employs Hyperledger Fabric (v2.3.3) [46]. All our enclaves were complied and executed with EGo (v1.0.0) [47] in simulation mode, using the parameter `OE_Simulation=1`. For the experiments (a)-(c), we utilize a chaincode named *kvs-test-go (KVS)*. The KVS chaincode essentially offers key-value storage, encompassing three primary functions: `get()`, `put()`, and `delete()`. We also exclude the SKVS solution from experiments (a)-(c) because of the execution read-write ratio of 50:50, many transactions in the SKVS solution will be rejected, severely affecting performance. This outcome does not represent the intended use case for the SKVS solution. Further details can be found in Section 2.7. For experiment (d), we leverage a chaincode named *ExtSecretKeeper*, an extended version of the Secret Keeper (see Section 2.5), to mimic real-world applications. To set a baseline for our experiments, we executed the same chaincode without any rollback protection. In the following sections, we refer to this baseline as *native FPC*.

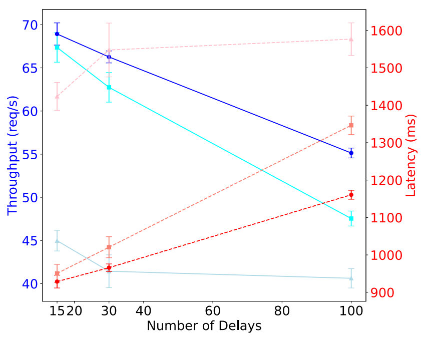
(a) The Impact of Increasing Number of Clients. This experiment evaluates the throughput and latency of the MTA and TLE solutions against various clients. The setup includes the KVS chaincode running on 2 peers, utilizing the *YCSB workload A* [48] with a 50:50 read-write ratio, an RTT of 15ms, and a Maximum Transaction Number (MaxTxnNum) per block set to the client count. Experiments were conducted for a different number of clients (1, 2, 4, 8, 16, 32, 64, 128, 256). We hypothesize that both throughput and latency will increase as client numbers rise. Figure 6a shows peak performance for all solutions at 128 clients, with saturation beyond this point. For a low number of clients, all solutions perform similarly, but in a higher load, our MTA performs the best. At peak performance, the TLE solution achieves nearly 65% of the native FPC's throughput, while MTA achieves nearly 95% of the native FPC's throughput. While our MTA meets our expectations, the TLE solution lags behind the native FPC in throughput after 16 clients. The reason for that is that TLE allocates excess memory and threads, taxing local computational resources. Additionally, the lack of TLE state versioning meant that state updates were locking the TLE state, impeding chaincode enclaves from accessing key-value pair metadata.

(b) The Impact of Network Delays. This experiment compares the TLE with our MTA with different network delays. We employed the KVS chaincode on 2 peers, adopting the *YCSB workload A* [48] with a 50:50 read-write ratio, 64 clients and MaxTxnNum per block set to the client count. Delays were tested for RTTs of 15ms, 30ms, and 100ms. We chose 64 clients because of its high performance and relatively small variance as we observed in the previous experiment. We anticipated that throughput decreases with rising RTT delays and expected a linear rise in latency as RTTs increase. As shown in Figure 6b, MTA performs as expected in terms of throughput and latency. In contrast, the TLE solution remains largely unaffected by network latency changes, suggesting bottlenecks remain elsewhere.

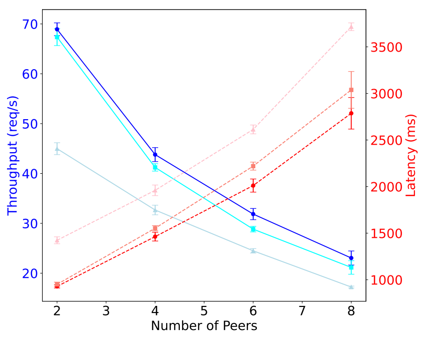
(c) The Impact of Increasing Number of Peers. We run the TLE and MTA with a varying number of peer counts. Using the KVS chaincode and using the *YCSB workload A* [48] with a 50:50 read-write ratio, 64 clients, an RTT of 15ms, and MaxTxnNum per block set to the client count, we tried various peer counts (2, 4, 6, 8). We expected a linear decline in throughput and a rise in terms of latency with increasing number of peers. Figure 6c shows that the results closely align with our expectations. The throughput of the TLE solution, while lower than native FPC, exhibited performance improvements, rising from 65% (2 peers) to 74% (4 peers), further to 76% (6 peers) and then settling at 75% (8 peers) compared to native FPC. However, it is worth noting that the TLE solution's latency has significantly increased when the peer count has reached a value of 8, suggesting it had reached saturation.



(a)



(b)



(c)

Figure 6: Impact on transaction throughput and latency of (a) varying number of clients; (b) varying time of network delays; and (c) varying number of peers.

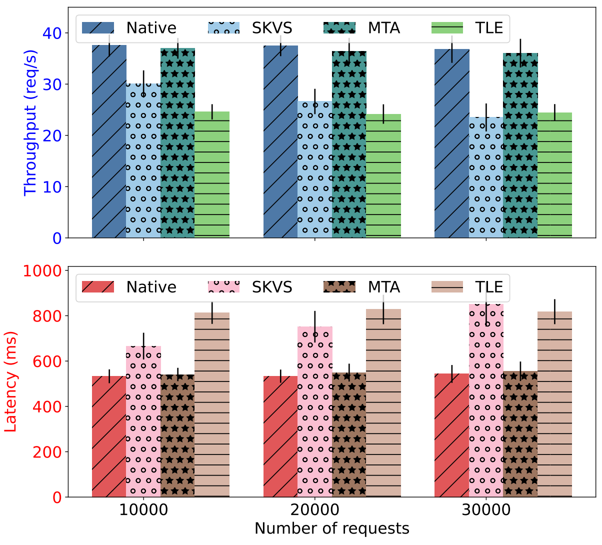


Figure 7: Performance of solutions in a real-world scenario over 30,000 requests, with data recorded at intervals of every 10,000 requests.

(d) Real-World Scenario Simulation. We deployed the *ExtSecretKeeper* chaincode on two peers, using a workload similar to the *YCSB workload B* [49] with a 95:5 read-write ratio, 20 clients, MaxTxnNum per block equal to client count, and a 15ms RTT network delay. This scenario was run with 30,000 requests for various solutions (SKVS, TLE, and MTA), with their latency and throughput recorded at every 10,000 requests. As shown in Figure 7, MTA is closely aligned with our expectations, achieving 97% of the native FPC throughput. In contrast, the TLE solution underperformed, reaching only 65% of the native FPC throughput. This reduction is largely due to the increased number of `get_state()` calls per request. Specifically, while the key-value storage (KVS) example involved a single `get_state()` call, this scenario requires two of such calls, thereby doubling the query load on the TLE solution and impacting its throughput. As anticipated, the SKVS solution exhibited latency increases with a higher total request count, resulting in a throughput decrease from 80% (10,000 requests) to 71% (20,000 requests) and finally to 64% (30,000 requests). These findings highlight the scalability limitations of the SKVS solution under heavy request loads.

4.4. Summary

Our evaluation shows that our Merkle Tree Approach (MTA) is a highly effective rollback protection strategy for FPC environments, closely matching native FPC performance and enhancing security. It stands out for its adaptability and comprehensive defense capabilities, suitable for performance and security-centric applications. Conversely, the TLE solution, despite its ambitious expansion of the codebase and additional enclave, encounters scalability and efficiency challenges, achieving only about 65% of native FPC throughput. Meanwhile, the SKVS solution, with its minimal code addition and simplicity, offers a direct fix to rollback attacks for read-heavy, low-storage applications, though it struggles with scalability and efficiency under increased data volume and read-write conflicts. This comprehensive evaluation not only affirms the efficacy of the proposed solutions but also lays down a comparative framework for future research and development in blockchain-specific rollback protection.

5. Conclusion

We revisited FPC's issue against rollback attacks, a prevalent threat where the trusted execution of smart contracts relies on externally stored states and lacks secure storage solutions. To that end, we implemented and compared the performance of three solutions *SKVS*, *TLE*, and our *MTA*, each tailored to address the issue rollback attacks within the FPC environment.

References

[1] M. Schunter, "Intel Software Guard Extensions: Introduction and Open Research Challenges," in *Proc. ACM Workshop on Software PROtection (SPRO)*, 2016.

[2] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Int. Workshop on HASP*, 2013.

[3] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, "Trusted Computing Meets Blockchain: Rollback Attacks and a Solution for Hyperledger Fabric," in *Proc. 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019.

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: a distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conference*, 2018.

[5] "FPC branch go-suuport-preview," 2023. [Online]. Available: https://github.com/chenchanglew/fabric-private-chaincode/tree/go-support-preview

[6] AMD, "Strengthening VM isolation with integrity protection and more," *White Paper*, 2020.

[7] Arm, "TrustZone for Cortex-A," 2023. [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-a

[8] V. Costan and S. Devadas, "Intel SGX Explained," Cryptology ePrint Archive, Paper 2016/086, 2016.

[9] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *Int. Workshop on HASP*, 2013.

[10] "Introduction to Intel® SGX sealing," 2016. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-intel-sgx-sealing.html

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *Proc. 11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[12] D. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX Amplifies the Power of Cache Attacks," 2017.

[13] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *SysTEX*, 2017.

[14] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. Lai, "SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution," 2018.

[15] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Proc. 27th USENIX Security Symposium*, 2018.

[16] S. van Schaik, A. Kwong, and D. Genkin, "SGAxe: How SGX Fails in Practice," 2020.

[17] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *Proc. 26th USENIX Security Symposium*, 2017.

[18] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *Proc. IEEE Symposium on Security and Privacy (SP)*, 2020.

[19] D. Page, "Defending against cache-based side-channel attacks," *Information Security Technical Report*, 2003.

[20] D. Genkin, W. Kosasih, F. Liu, A. Trikalinou, T. Unterluggauer, and Y. Yarom, "CacheFX: A Framework for Evaluating Cache Security," *CoRR*, vol. abs/2201.11377, 2022.

[21] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[22] V. Buterin, "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform," 2013.

[23] "Hyperledger Fabric Docs," 2022. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.3/

[24] "Hyperledger Fabric Client SDK for Go," 2022. [Online]. Available: https://github.com/hyperledger/fabric-sdk-go

[25] "Hyperledger Fabric Client SDK for Node.js," 2022. [Online]. Available: https://github.com/hyperledger/fabric-sdk-node

[26] "Hyperledger Fabric Client SDK for Java," 2022. [Online]. Available: https://github.com/hyperledger/fabric-sdk-java

[27] "Hyperledger Fabric - Gossip data dissemination protocol," 2024. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/gossip.html

[28] "Intel SGX documentation: SGX create monotonic counter." [Online]. Available: https://cdrdv2-public.intel.com/671564/intel-sgx-platform-services.pdf

[29] R. Strackx and F. Piessens, "Ariadne: A Minimal Approach to State Continuity," in *Proc. 25th USENIX Security Symposium*, 2016.

[30] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback Protection for Trusted Execution," in *Proc. 26th USENIX Security Symposium*, 2017.

[31] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and Forking Detection for Trusted Execution Environments Using Lightweight Collective Memory," in *Proc. 47th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks (DSN)*, 2017.

[32] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A Secure Database Using SGX," in *Proc. IEEE Symposium on Security and Privacy (SP)*, 2018.

[33] S. Angel, A. Basu, W. Cui, T. Jaeger, S. Lau, S. Setty, and S. Singanamalla, "Nimble: Rollback Protection for Confidential Cloud Services," in *Proc. 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[34] M. K. Jangid, G. Chen, Y. Zhang, and Z. Lin, "Towards Formal Verification of State Continuity for Enclave Programs," in *Proc. 30th USENIX Security Symposium*, 2021.

[35] S. Agrawal and K. Tu, "Enabling Verifiable Execution of Distributed Secure Enclave Platforms," Master's thesis, EECS Department, University of California, Berkeley, 2021.

[36] K. R. M. Leino, "Trusted Computing on Modern Platforms: Analysis, Challenges, and Implications," 2024.

[37] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing Digital Side-Channels through Obfuscated Execution," in *Proc. 24th USENIX Security Symposium*, 2015.

[38] "Hyperledger Fabric Private Chaincode RFC," 2021. [Online]. Available: https://github.com/hyperledger/fabric-rfcs/blob/main/text/0000-fabric-private-chaincode-1.0.md

[39] "FPC SKVS Implementation," 2023. [Online]. Available: https://github.com/chenchanglew/fabric-private-chaincode/tree/thesis/merkle-solution

[40] "FPC TLE Implementation," 2023. [Online]. Available: https://github.com/chenchanglew/fabric-private-chaincode/tree/thesis/tle-solution

[41] "Fabric TLE Support," 2023. [Online]. Available: https://github.com/chenchanglew/fabric/tree/feature/support-TLE

[42] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Annual Int. Cryptology Conference*, 1987.

[43] "Fabric MerkleTree Support," 2023. [Online]. Available: https://github.com/chenchanglew/fabric/tree/feature/support-merkle

[44] "FPC MerkleTree Implementation," 2023. [Online]. Available: https://github.com/chenchanglew/fabric-private-chaincode/tree/thesis/merkle-solution

[45] "Fabric Malicious Peer," 2023. [Online]. Available: https://github.com/chenchanglew/fabric/tree/feature/malicious-peer

[46] "Hyperledger Fabric v2.3.3," 2021. [Online]. Available: https://github.com/hyperledger/fabric/releases/tag/v2.3.3

[47] "EGo v1.0.0," 2022. [Online]. Available: https://github.com/edgelesssys/ego/releases/tag/v1.0.0

[48] "YCSB Workload A," 2019. [Online]. Available: https://github.com/brianfrankcooper/YCSB/blob/master/workloads/workloada

[49] "YCSB Workload B," 2019. [Online]. Available: https://github.com/brianfrankcooper/YCSB/blob/master/workloads/workloadb