2025 23rd International Symposium on Network Computing and Applications (NCA) | 979-8-3315-7842-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/NCA67271.2025.00050

EvoChain: A Recovery Approach for Permissioned Blockchain Applications

Francisco Faria, Samih Eisa, David R. Matos and Miguel L. Pardal
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
{franciscofaria00, david.r.matos, miguel.pardal}@tecnico.ulisboa.pt samih.eisa@inesc-id.pt

***Abstract*—Blockchain technology provides decentralized data storage and processing. It can ensure data integrity and auditability. Many applications now adopt it in scenarios with multiple stakeholders and shared ownership, such as supply chain management. However, strict immutability makes it difficult to correct mistakes or intrusions in real-world deployments.**

This paper presents *EvoChain*, a chaincode extension for Hyperledger Fabric, a permissioned blockchain platform. *EvoChain* adds *controlled mutability*, allowing data to be corrected or recovered under time-limited or specific conditions. Changes can be made during a grace period, after which immutability is enforced again.

We evaluated our approach with WineTracker, a supply chain application. It was modified to allow some users to cancel unwanted operations while preserving the security and consistency of data in the blockchain. Performance results show minimal overhead with functional benefits.

***Index Terms*—Controlled Mutability, Data Redaction, Intrusion Recovery, Blockchain, Smart Contracts**

I. Introduction

Blockchain technology has emerged as a prominent research topic since the introduction and success of the Bitcoin cryptocurrency [1]. A blockchain introduces the possibility of maintaining a distributed ledger of transactions without a central authority. Instead, the ledger is maintained by a peer-to-peer network that stores the data in blocks that have multiple, distributed replicas. The data blocks are linked and protected using cryptographic primitives, allowing the verification of authenticity, integrity, and non-repudiation of transactions. Each network peer replicates the ledger and, with the aid of a consensus protocol, agrees on a unified view of the chain. These characteristics combine to make an immutable and tamper-resistant system that simplifies data auditing processes [2].

Public blockchains are open networks where anyone can join and validate transactions. In contrast, permissioned blockchains restrict participation to approved members, who are known and verified. This controlled access improves efficiency, privacy, and governance while still keeping the benefits of a shared ledger. As a result, permissioned systems are often better suited for enterprise and multiple stakeholder applications such as supply chain management.

Work supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 and UID/PRR/50021/2025 (INESC-ID) and Project Blockchain.PT – Decentralize Portugal with Blockchain Agenda, (Project no 51), WP 1: Agriculture and Agri-food, Call no 02/C05-i01.01/2022, funded by the Portuguese Recovery and Resilience Program (PRR), The Portuguese Republic and The European Union (EU) under the framework of Next Generation EU Program.

Immutability is a key blockchain property, but it can also be a hindrance to the adoption of the technology [3], [4]. Many practical applications require some flexibility for data redaction [4], to rectify errors made by human or machine actors. Human errors are unavoidable, and intentional misconduct must also be taken into account. Since the nature of the blockchain does not allow for data modification, fraudulent records would persist in the system. These issues can become an obstacle to the adoption of the technology in several industries that require some flexibility, such as finance, insurance, healthcare, and supply chains.

Several approaches have been proposed to cope with the limitations presented and introduce recovery mechanisms for blockchain-based applications [5], [6]. These approaches aim to implement redaction mechanisms on blockchain systems without breaking their core security properties. However, they either require developers to change the underlying blockchain or focus on the recovery of tokens and not arbitrary applications. In Section II-C, we provide an analysis of state-of-the-art on blockchain systems with mutability, where we discuss their vulnerabilities and limitations.

In this work, we propose a new framework, *EvoChain*, an *evo*lution of block*chain*, that introduces apparent transaction mutability while preserving the fundamental blockchain properties of integrity, authenticity, and non-repudiation. *EvoChain* proposes a high-level transactional model to generate a consistent and mutable view of the chain, taking advantage of Write-Ahead Logging (WAL) [7]. Unlike other recovery approaches, *EvoChain* does not require modifications to the underlying blockchain, since it implements a new data architecture within the Chaincode. We take a supply chain use case [4], based on a permissioned blockchain to implement and test an application built with the new controllable mutability.

II. Background

This section provides background on intrusion recovery research and blockchain technology, outlines the key properties of standard blockchain systems, and concludes with an overview of Hyperledger Fabric, a widely adopted blockchain framework.

A. Intrusion Recovery

Intrusion recovery is the process of identifying and reversing the effects caused by unintended operations in a system. Although new security technologies tend to make intrusions more difficult, they are still unavoidable in practice because of

979-8-3315-7842-8/25/$31.00 ©2025 IEEE
DOI 10.1109/NCA67271.2025.00050

bugs and user fallibility. Two common approaches are generally followed when dealing with intrusion recovery: *rollbacks* and *compensations*. Rollback is based on the idea of reverting all the activity of an application to a point in time prior to an intrusion, a checkpoint. Compensations [8] undo committed or uncommitted transactions that affect the state by applying special-purpose compensating transactions that revert the state changes caused by the unintended operation.

B. Blockchain

Blockchain is a distributed ledger system that operates without the need for central authorities or trusted third parties. A network of nodes with computing capabilities maintains and validates transactions through a consensus protocol.

Blockchains are designed to tolerate, at least, crash faults. CFT (Crash Fault Tolerance) ensures system reliability by handling failures where nodes stop functioning without malicious behavior, whereas BFT (Byzantine Fault Tolerance) extends this capability by tolerating arbitrary or malicious faults [9].

Blockchain networks can be categorized according to the nature of participation: permissionless (public) and permissioned (private or consortium). In permissionless blockchains, such as Bitcoin [1] and Ethereum [10], anyone can participate in the network without the need for authorization. This open nature gives the system high resilience but also performance and scalability challenges. Inversely, permissioned blockchains only allow the participation of authorized parties, ensuring access control. These types of blockchains are often used in enterprise settings where data confidentiality and governance are desired characteristics. An example is Hyperledger Fabric [11].

A transaction is the fundamental unit of activity in the blockchain. Transactions involve either adding new data to the ledger, updating records, or transferring tokens. A token is a digital asset representing value, utility, or ownership within a blockchain. Each validated transaction is added to a storage unit, the block. Each block contains a set of validated transactions, along with metadata and a unique identifier.

Agreement and trust between nodes are achieved by performing a consensus protocol. There are several consensus protocols, each with its own strengths and trade-offs. Proof-of-Work (PoW) [1], [12], [13], Proof-of-Stake (PoS) [14], Delegated Proof-of-Stake (DPoS) [15] were designed specifically for public blockchain systems, while others, such as Practical Byzantine Fault Tolerance (PBFT) [9], have been adapted from previous research on fault-tolerant systems to private blockchains.

C. Blockchain Immutability

Immutability is a property of blockchain technology and ensures that a transaction, once executed, cannot be modified or deleted. This property is supported through the use of cryptographic primitives that are the base of the chain and certifies that a transaction, after being accepted on the chain, cannot be removed or changed (mutated).

The integrity of the data is ensured by the use of the Merkle Tree data structure [16], along with the collision resistance and block linking derived from the use of cryptographic hash functions [17]. Immutability is particularly desirable in contexts where data integrity and a simplified auditing process is crucial and necessary, such as in financial transactions. However, this property may contradict several privacy requirements and data protection rights, as well as the adoption of blockchain technologies for a wider range of applications where data requires some changes. Theoretical and technical implementations of a mutable system that retains the inherent security of a blockchain system are still in their early stages, although several cryptographic and innovative approaches have been emerging recently [5], [6]. These methods typically rely on one of two strategies: circumventing/bypassing immutability or conditionally removing it. Bypass strategies involve changes to how data is represented or stored on-chain, the introduction of new data structures, or the use of a decentralized set of judges. Although these strategies avoid heavy cryptographic primitives, they introduce new attack vectors to the system. Regarding removing strategies, we can also define three main techniques based on the concepts of Consensus [18], Chameleon-Hash [19] and Meta-Transactions [20]. These strategies involve hard forks of the main chain, cryptographic changes to hash functions, or changes to the blockchain data structures. They usually require heavier cryptographic primitives, introduce delays to the system, and often weaken auditability.

D. Hyperledger Project

Hyperledger [21] is a project framework hosted by The Linux Foundation, which offers guidelines, standards, and tools to develop cross-industry blockchain technologies. The goal is to provide the necessary infrastructure and standards to develop systems and applications for industrial use cases. Hyperledger includes various subprojects, including Fabric [11] and Caliper [22].

Hyperledger Fabric [11] is an open-source, permissioned distributed ledger platform designed to be used in enterprise contexts. With its modular and configurable architecture, it provides adaptability to a broad range of use cases in industries such as banking, finance, and supply chains. It is programmable and supports the use of smart contracts (chaincode) in general-purpose programming languages. Its modular architecture has several main components: the membership service, responsible for the entities in the network; the peer nodes, responsible for the ledger; the chaincode, responsible for the application logic; the ordering service, responsible for ordering transactions and the consensus protocol; and the peer-to-peer (P2P) protocol, responsible for communication between peers. Fabric uses the concept of *world state* which refers to the latest values of all key-value pairs in the blockchain network. Unlike traditional blockchains, Fabric separates the concept of world state and transaction history by implementing versioning of assets, which allows managing changes and updating assets over time. The world state is maintained in a state database, such as LevelDB or CouchDB, alongside the ledger, for efficient query and update operations. The world state enables quick access to the current value of any asset without having to traverse the

entire transaction log. When a key-value pair is updated, a new version is created, preserving a history of changes in the ledger.

III. EvoChain

We present *EvoChain*, a development framework for chaincode that brings in controlled mutability for decentralized applications. The main goal is to propose a chaincode development framework that allows for reversal and recovery of operations. In this Section, we introduce an overview of the components, a general sequence model of a redaction, where a submitted transaction is canceled and not considered to the final state of the system, and the template code of the main components.

EvoChain uses WAL [7] to generate a consistent and correct view of the blockchain data. For this, it leverages the Hyperledger Fabric world state that maintains individual logs for each processed transaction in a sequential manner.

A. Recovery Scenarios

Here are some of the scenarios where *EvoChain* is intended to work:

- **S1:** Account Theft: by means of phishing attacks, social engineering, or malware/keylogging, a user loses access to an account and cannot recover;
- **S2:** User fault: a user, inadvertently or due to a misunderstanding, issues a transaction that was not meant to be sent out to the blockchain network;
- **S3:** Incorrect Authorization and Access Controls: insufficient safeguards regarding access control and permissions granted to users create the potential risk of unauthorized entry and improper utilization of features;
- **S4:** Smart contract exploitation: chaincode has vulnerabilities and bugs, which can result in unintended behavior and security breaches

B. Threat Model

We assume that an attacker can perform the following operations from the application level:

- **A1:** A mistakenly issued transaction by an authorized user;
- **A2:** Tricking users to issue unwanted transactions;
- **A3:** Steal user private keys and illegally execute transactions;
- **A4:** Exploit smart contract vulnerabilities.

Our model focuses on application-level attacks since the Hyperledger Fabric's design and architecture already address several external threats: Sybil attacks [23] by using permissioned networks with Membership Service Providers to control participant identities; Network Partitioning [24] since Fabric's channels and endorsement policies ensure that valid transactions are eventually committed; Eclipse attacks [25] by presenting the isolation of validating peers' endorsement policies and gossip protocols; and Blockchain Reorganization attacks [26] since Fabric offers a strong consistency model and chain validation process, therefore preventing the tampering of past transactions.

We also do not consider possible flaws in the underlying consensus algorithms, hash functions, and digital signatures, since delving into underlying technologies is out the scope of this work.

C. Design

EvoChain introduces controlled mutability into application logic. It brings transaction concepts to the application layer by storing transaction objects in the world state rather than storing only individual assets. By recording changes as transaction objects, *EvoChain* enables fine-grained control of the Fabric world state and therefore controlled mutability. The interface remains largely unchanged, but *EvoChain* adds a `cancel` operation to support undo semantics. Rather than writing asset values directly to the world state, each *EvoChain* transaction is (1) recorded as a Fabric transaction in the ledger and (2) represented as a transaction object in the world state. Such transaction objects participate in view generation but are only considered consolidated (and thus immutable) after their mutation policy and consolidation conditions are satisfied.

EvoChain defines two types of transaction objects: Mutable Transactions (MTs) and Canceling Transactions (CTs).

Mutable-Transactions (MT) are standard blockchain transactions extended to accommodate four fields: submission time, permanent state time, validity, and delay, as well as the objects regarding the application logic. MTs are issued in a *pending* validity state, with a submission time related to the time they were issued and a default delay

Canceling-Transactions (CT) work as special-purpose transactions that update the state of previously issued *pending* MTs to a *canceled* validity state. CTs are accepted if they refer to a MT that has not reached the $mutation\ policy$ specified, namely if the consolidation delay has expired, or a certain condition, for example, the issue of a specific dependent transaction within the system. That cancellation (undo) will affect every transaction that is dependent of that MT, producing a rollback effect. This rollback will revert the changes performed by a transaction that will be noticeable in the view generation.

In *EvoChain*, transactions that are dependent on the outcome of a previous must be governed by a mutation policy that expires in a later timeframe.

For example, consider two transactions, A and B, where B depends on A denoted as $(B \rightarrow A)$. This dependency arises when both transactions modify the same object, and A was issued before B. In this case, B can be consolidated only if A has already been consolidated, i.e., $C(B) \Rightarrow C(A)$.

The mutation policy is defined in the application's chaincode, specifying the actors and the conditions under which a Canceling Transaction (CT) can be issued. Each transaction is assigned a submission time that serves as its version identifier, with the original Mutable Transaction (MT) representing the first version of the corresponding state.

D. Operations

The *EvoChain* chaincode framework provides four generic operations: $IssueTransaction$, $CancelTransaction$, $GetAsset$, and $GetTransactions$.

$IssueTransaction$ works as a standard operation to submit a transaction to the network. It submits a transaction that is, considered within the application chaincode, a MT. It is verified by the chaincode and, if valid, submitted.

$CancelTransaction$ is issued with the goal of canceling a MT. The mutation policy is verified by the chaincode that accepts the transaction if it is not yet considered consolidated. If accepted, from that moment forward, the view generated by the system changes and the final object stops affecting the canceled MT.

$GetAsset$ is the standard query operation of an asset created in the network, returning the most recent and not affected by canceled transactions version of an object, considering the view generated by the chaincode.

$GetTransactions$ is an extended query operation of an asset. Returns every pending and consolidated transaction that affects the state of an object.

Query operations consolidate pending transactions by applying expired delays, ensuring that transaction views are updated only when required. This on-demand consolidation introduces additional overhead during query execution, as discussed in Section V-C3.

E. Mutation policies

Hyperledger Fabric comes with built-in policies that ensure transaction and block validation, as well as resource access control. *EvoChain* leverages these and extends them to also become *mutation policies*. These consist of a set of rules and conditions that define the interplay between Mutable-Transactions (MT) and Canceling-Transactions (CT) so that modifications to the ledger state are accepted or denied. They also implement access control on CTs.

A *pending* transaction can still be canceled. A transaction is *pending* if the associated delay has not already expired or if it has not yet been consolidated by a conditional procedure. Each transaction is associated at the time of execution with a delay, which can be seconds or minutes, depending on the application context. This delay is the interval between the moment a transaction is issued and the moment it should become immutable. The delay can be altered by an administrator, as long as the transactions that it is dependent on expire in a shorter timeframe.

There are two ways that a transaction can become *consolidated*: by expiration or by satisfied condition. The *expiration* is checked when a later MT is issued, or there is a query. The view generation component checks the transaction that affected the corresponding object and if its delay has expired. If a checked transaction has an expired delay, its status is updated to consolidated and can no longer be mutated. The *condition* is checked and can trigger the consolidation of a previously issued transaction that it is dependent on. This relationship is configured within the chaincode and also depends on the application logic.

F. Dependency Graph

EvoChain leverages Hyperledger Fabric's world state to access the current value of objects without traversing the entire transaction log.

A dependency graph is constructed from the transactions to systematically track relationships and dependencies between objects. This graph ensures that the lifecycle of each object can be efficiently inferred and analyzed by the View Generation component, especially when multiple actors and transactions are involved.

In Figure 1, each node represents a Mutable Transaction (MT), which creates new objects or alters the state of existing ones. Multiple actors can issue MTs that affect the same objects. Directed edges indicate dependencies between transactions: an edge from T_i to T_j signifies that T_j depends on the prior execution of T_i ($T_i < T_j$ and $\text{TS}(j) > \text{TS}(i)$). For example, in the graph shown, if a mutable transaction T_4 is altered, it directly influences T_5 and T_6.

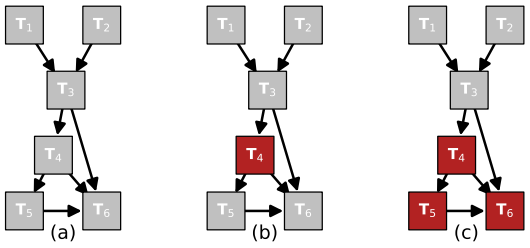

Fig. 1: Dependency Graphs

G. View Generator

The *EvoChain* View Generator produces a canonical representation of object state within the system. It filters and structures data according to the dependency graph, ensuring users observe the most recent confirmed version of each object. Selection and filtering follow application-defined confirmation and validation rules.

The view generator retrieves the latest valid consolidated state and then replays subsequent valid transactions (pending or consolidated) to construct the final object presented to the application. We adopt a high-level transactional model in the chaincode: object data is stored together with the transactions that modified it in the ledger. The dependency graph leverages transaction metadata (e.g., submission time and permanent-state timestamps) to determine inter-transaction dependencies when generating views.

Integrating data with transactions enables an interconnected transaction model that composes object state by traversing the ledger's transaction graph. Because object state may be modified by multiple transactions, *EvoChain* traverses the graph to compose a consistent state, taking into account confirmed, pending, and canceled transactions. The resulting view is the authoritative state used by the application when creating or validating new transactions.

The pseudocode of the view generation algorithm is presented on Algorithm 1. Each transaction that alters the state of an object is ordered by its submission time, following the chronological order maintained by the ledger. This allows to view the object state history and undo operations that have been performed. When a transaction is issued, it is in a mutable state until the *mutation policy* condition is reached. The transactions that precede that transaction, if not in a permanent state, can alter the new object state within that transaction.

We can guarantee that if a transaction is consolidated and was issued after the permanent state of the last canceled transaction, the corresponding object state is valid. Therefore, this object can serve as a reliable starting point for the view generation process, from which the remaining history can be inferred.

H. Recovery Process Example

Figure 2 shows a sequence of transactions created by three clients. In this example three clients interact with the Fabric network via the chaincode API. Client 1 may issue and cancel transactions, Client 2 is an observer, and Client 3 may only issue transactions.

- Client 3 issues a MT that creates an object with id = 1 and value "a", step (1). The network confirms the MT and returns a response containing the created asset's properties to Client 3, step (2);
- Client 2 queries the ledger for the asset with id = 1, step (3). Because the chaincode models transactions as first-class objects, the node runs the View Generation algorithm (Algorithm 1) to collect the transactions that affected asset 1, generate a consistent view, and return the reconstructed object to Client 2, step (4);
- Client 1 issues a CT that targets the MT previously issued by Client 3, step (5). If the MT has not yet been consolidated, the CT marks the MT's validity as **CANCELED**. After the CT is applied, view generation no longer considers the original creation of asset 1, so queries for that asset return no result, step (6);
- Client 1 then issues a new MT that creates an object with id = 1 and value "b", step (7). Because the prior MT was canceled and the current view no longer contains asset 1, the new MT is accepted and the system returns the newly created asset to Client 1, step (8);
- Finally, Client 2 queries the ledger again for transactions affecting asset 1, step (9). The View Generation algorithm now produces a view containing the object created by Client 1 (id = 1, value "b") and returns it to Client 2, step (10).

Referring to the threat model in Section III-B, *EvoChain* supports recovery from unwanted operations, provided that mutation policies are properly configured and the relevant threat scenarios are anticipated.

For attack scenarios A1 and A2 (erroneous transactions or social-engineering that induces users to perform unwanted operations), the system can recover by permitting cancellation. Specifically, administrators may issue cancel transactions (CTs), and the mutation policy can allow users to cancel their own transactions while those transactions remain pending.

In scenarios A3 and A4 (compromise of private keys and unauthorized transaction execution), recovery requires a consolidation policy and an operational procedure. If the consolidation delay is set sufficiently high, network operators

Algorithm 1: View Generation Algorithm

Data: Transactions influencing object id
Result: Final reconstructed object

Function `ViewGenerator`(txs, id)**:**
  $initialObj \leftarrow \emptyset$; $confirmedTx \leftarrow null$;
  // Initialize base object and confirmation flag
  Sort txs by $submissionTime$ (desc);
  // Sort all transactions (latest first)
  $consolTxs \leftarrow FilterByConsolidated(txs)$;
  $cancTxs \leftarrow FilterByCanceled(txs)$; sort $cancTxs$ by $PermanentState$ (desc);
  // Separate and order consolidated and canceled transactions
  foreach *cTx in consolTxs* **do**
    if $confirmedTx \neq null$ **then**
      break
    // Stop if already found a confirmed transaction
    $filtered \leftarrow FilterLowerSubTime(cancTxs, cTx)$;
    // Canceled txs older than current consolidated one
    if $filtered.isEmpty()$ **then**
      $confirmedTx \leftarrow cTx$; **break**;
      // No cancels affect it → confirmed
    foreach *cancel in filtered* **do**
      if $cTx.SubTime > cancel.PermaState$ **then**
        $confirmedTx \leftarrow cTx$; **break**;
        // Valid confirmed tx unaffected by cancel
      else
        continue;
        // Influenced → check next
  if $confirmedTx \neq null$ **then**
    $initialObj \leftarrow confirmedTx.getObject()$;
    $txs \leftarrow FilterByNotCanceled(txs)$;
    $txs \leftarrow FilterByHigherSubTime(txs, confirmedTx)$;
    // Use confirmed tx as base; keep later valid ones
  else
    $txs \leftarrow FilterByNotCanceled(txs)$;
    // No confirmed tx → use all valid ones
  foreach *tx in txs* **do**
    $initialObj \leftarrow ApplyChanges(initialObj, tx)$;
    // Sequentially apply valid transactions
  return $initialObj$;
  // Return final reconstructed object

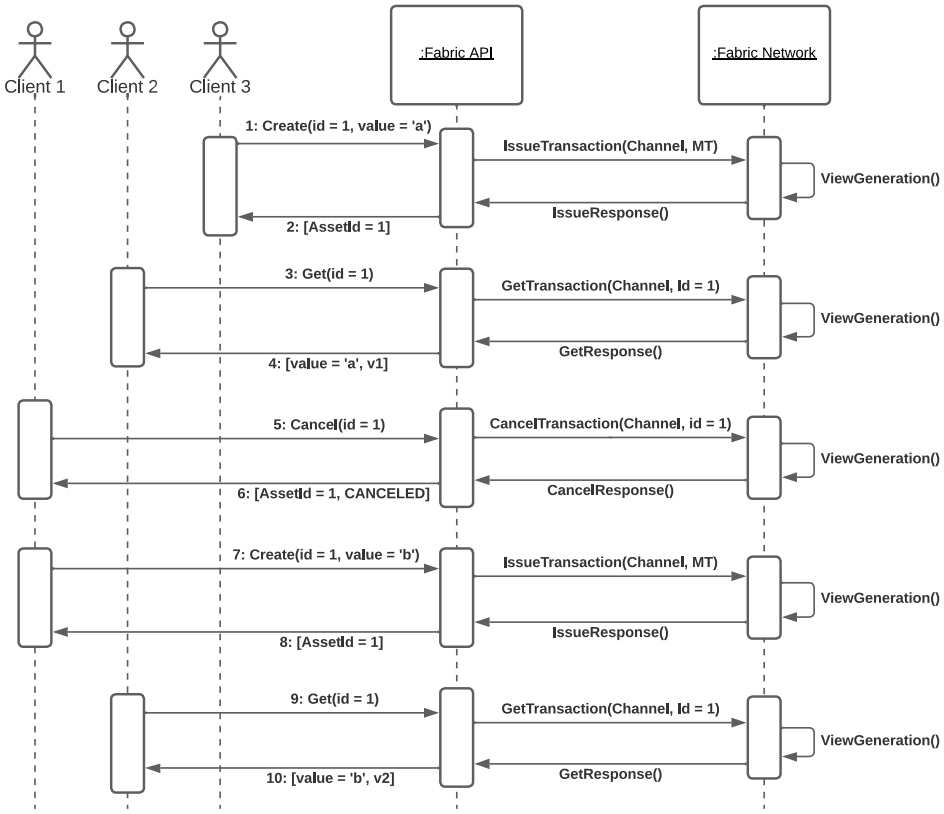

Fig. 2: Sequence diagram for a Recovery example

can first revoke the compromised user's authorization and then authorize a trusted operator (for example, an administrator) to submit CTs that cancel the illicit transactions. Executing these CTs reverts the reconstructed view to a prior consistent state and mitigates the attack.

IV. Use Case

To evaluate *EvoChain*, we designed a use case based on a simplified wine supply chain model.

A. WineTracker

Biswas et al. [27] proposed a blockchain-based wine traceability application to address the growing problems of counterfeiting, adulteration, and unsafe production practices in the wine supply chain. Their goal was to enable consumers to verify the authenticity and properties of each wine batch through immutable records registered by supply chain participants.

The WineTracker system models multiple entities, each responsible for a specific stage of the supply chain. As shown in Figure 3, objects are sequentially created, transformed, and transferred between these actors. Growers produce the initial object, *Grapes*, which is sold in bulk to Wine Producers. Producers transform the grapes into *BulkWine* and transfer it to Fillers. Fillers bottle the wine, assign a unique identifier to each bottle, and distribute the final product to Distributors, Retailers, and Consumers.

Assets are submitted to *EvoChain* as serialized JSON (JavaScript Object Notation) objects. Each asset includes a set of attributes defined by its type, such as `batchId`, `sellId`, and `owner`, as well as additional fields required for the traceability process.

B. EvoChainWineTracker

The *EvoChain*WineTracker is an adapted version of the original WineTracker application, reimplemented to leverage the *EvoChain* framework.

In this version, the underlying data structures are redesigned to follow a transactional model within the chaincode, enabling fine-grained control over state changes. This modification extends the original design by integrating transactional traceability and enabling reversible state computation through *EvoChain*'s mutation and view generation mechanisms. Each operation that modifies an object's state is represented as a transaction recorded on the ledger. This design enables a view generation

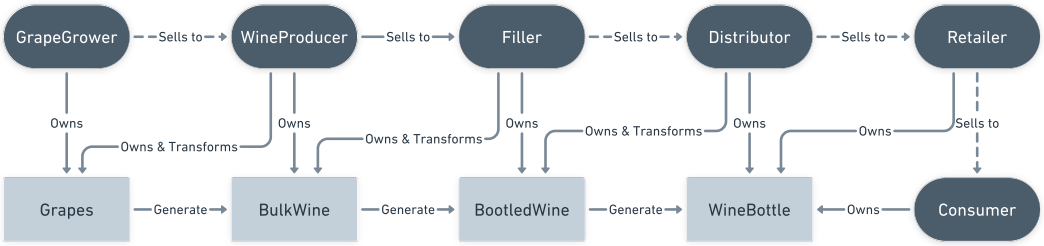

Fig. 3: WineTracker relationships

component within the chaincode, capable of reconstructing a consistent view of the system's data—even after redactions or transaction cancellations.

The implementation was configured with a set of mutation policies and access control rules defined as follows:

- *Role-based permissions:* Users within an organization are authorized to perform transactions corresponding to their assigned role in the network model;
- *Cancel Transactions (CTs):* Both users and administrators can issue CTs, which allow the cancellation of previously submitted transactions;
- *Administrator privileges:* Administrators may issue CTs for any transaction that has not yet been consolidated;
- *User privileges:* Users may issue CTs only for transactions that they have created themselves, and that are still pending (not yet consolidated);
- *Consolidation delay:* The delay was configured to several hundred seconds to ensure that performance tests were affected by transaction consolidation only when explicitly intended.

While *EvoChain* introduces cancel transactions to enable recovery and error correction, this mechanism implies additional privileges for certain actors. In a permissioned environment, an authorized client could, in theory, misuse this privilege to cancel valid transactions. To address this, every cancel transaction in *EvoChain* is immutably recorded, cryptographically signed, and fully auditable.

Additionally, administrative actions can be governed by role-based access control and multi-party endorsement policies, ensuring that no single entity can perform unilateral redactions. These measures transform the "trusted administrator" concept into a verifiable, accountable, and policy-driven governance mechanism, preserving the integrity and auditability expected in blockchain systems.

C. EvoChainWineTracker Transactional Architecture

Each "Transforms" and "Sells to" operation in Fig. 3 corresponds to a transaction that can be issued by a specific actor in the network. In the original model, these operations directly modified the object state, committing the new values to the ledger and updating the world state. While it is possible to retrieve the historical data for a given key from the blockchain, reconstructing the dependencies between these changes across the supply chain is complex. This makes it difficult to revert the world state to a consistent previous version. By integrating *EvoChain* into WineTracker, object histories become traceable through explicit transaction relationships. Consequently, the application gains the ability to rollback processes to earlier states when required.

All transactions extend a `Transaction` class. If a transaction modifies the state of an object, it includes the updated version of that object. Transactions responsible for object creation contain the initial version of the created object. In simpler cases, transactions that consume or modify an object only once may store only the resulting version of that object, as they do not affect other transactions. From this point onward, transactions—rather than assets—are recorded in the ledger, each accompanied by a unique identifier.

V. Evaluation

We evaluate the performance of our prototype, *EvoChain*WineTracker, by comparing it with the original WineTracker implementation. Both smart contracts are deployed under identical conditions and process the same workload of requests. However, *EvoChain*WineTracker introduces additional functionalities. Most notably, the ability to cancel transactions while preserving the core features and behavior of the original system.

A. Methodology

To assess the capabilities and performance of *EvoChain*, the following steps were taken:

1) *Application implementation*: a supply chain application was implemented using Hyperledger Fabric to establish a baseline for comparison;
2) *Framework implementation*: we integrated the *EvoChain* framework into the same supply chain application, simulating a real-world use case;
3) *Performance assessment*: we performed cloud deployments and measured key metrics – request latency (in milliseconds) and throughput (in Transactions Per Second, or TPS) – to compare the performance of both application versions;
4) *Validity assessment*: we performed a series of functional tests to ensure the validity and correctness of the results, focusing on qualitative parameters.

Load tests were conducted to evaluate the performance overhead introduced by the *EvoChain*-enhanced version of the application compared to the original implementation. The evaluation focused on measuring latency, throughput, and system breakpoints, providing insight into the performance impact of integrating *EvoChain* into an existing system.

Multiple functionalities were tested to assess not only the performance of the *cancel* operations but also the overhead introduced to the core WineTracker functionalities described in Section III-D.

B. Experimental Setup

In our deployment, the blockchain network consists of four organizations (Org1–Org4), each with a single peer and a dedicated certificate authority. All peers are connected via a single channel to enable transaction communication, while certificate authorities issue credentials to their respective peers and clients.

The network was deployed on a Google Compute Engine instance (e2-highcpu-16) with 16 vCPUs, 16 GB of memory, and a 100 GB Balanced Persistent Disk. Our tests were performed on Hyperledger Fabric images, version 2.5.0, and the system was deployed using container technology to modularize and manage our system components. Docker v24.0.6 and Docker-Compose 1.29.2 were used to achieve containerization. Performance benchmarking was conducted using Hyperledger Caliper v0.5.0. The following application software and configurations were used:

- LevelDB was used as the world state database;
- Single application channel, with 4 organizations, 1 peer per organization and a single orderer;
- Ordering consensus was provided by a Raft-based [28] algorithm;
- Chaincode was implemented in Java using the Contract API and deployed on the network;
- The endorsement policy was set to *Majority*, requiring a majority of endorsing peers to validate each transaction;
- No private data collections or range queries were utilized;
- TLS (Transport Layer Security) was enabled to secure communications among all network entities;
- All other settings followed the default Fabric policies and configurations.

C. Performance Evaluation

We evaluated the performance of core functionalities, query operations, and cancel transactions using three test cases (TCs).

1) TC1: The core functionalities were evaluated by comparing both application versions in terms of throughput, latency, and memory usage, to assess the overhead introduced by the *EvoChain* framework.

In this benchmark, ten workers simulated 10,000 transactions per round. The transaction rate started at 400 TPS and was gradually increased to 1,200 TPS per round. The results are presented in Figure 4.

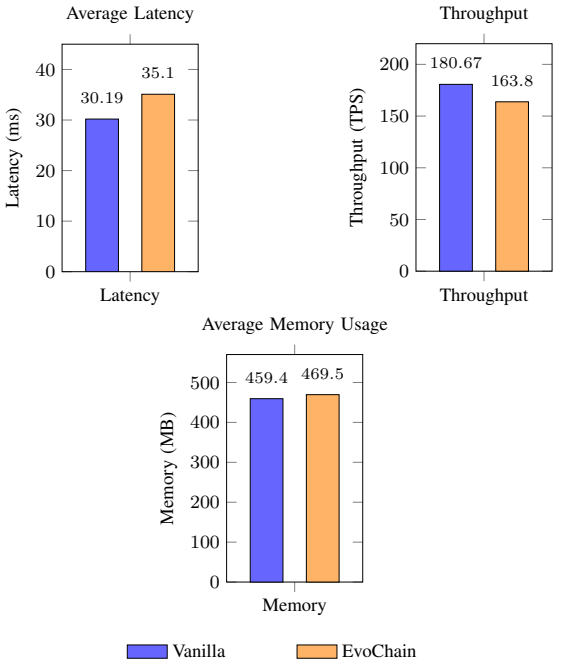

Fig. 4: Comparison of Vanilla and EvoChain Versions of WineTracker

2) TC2: The second test case evaluates the performance of cancel transactions (CTs) and their impact on core functionalities. CT performance was assessed through incremental rounds of sequential core functionalities. In our implementation, canceling a pending *createGrapes* transaction recursively cancels all subsequent transactions associated with the same batchId. The benchmark measured the performance of canceling a *createGrapes* transaction with varying numbers of dependent transactions, as well as issuing identical transactions that modify the same keys.

The benchmark comprised five transactional rounds: create, sell, transform, and cancel operations. Ten workers simulated 10,000 transactions per round, with the transaction rate starting at 400 TPS and increasing to 1,200 TPS.

Performance results are presented in Figure 5

3) TC3: We evaluate query transactions, which consolidate pending transactions whose delay period has expired. This evaluation measures the performance impact of operations that consolidate other transactions. Specifically, we compare the performance of query transactions before and after the delay expiration of the queried transactions. In this test case, we executed 5,000 *create*, *sellGrapes*, *transform*, and *sellBulk* transactions. Each transaction was subsequently queried 5,000 times before and 5,000 times after the expiration of the delay period to evaluate the system overhead. The benchmark involved ten workers, simulating 5,000 transactions per round, with the transaction rate controlled between 400 and 1,200

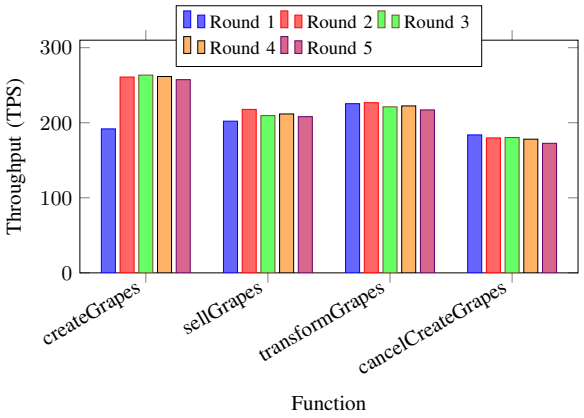

Fig. 5: Throughput (TPS) by Function and Test Round for TC2 Scenario

TPS. The results are summarized in Tables I and II.

TABLE I: Performance of TC3 Consolation Query Operations

Operation	Send Rate (TPS)	Avg. Latency (s)	Throughput (TPS)
getCreate	373.1	4.39	257.9
getSell	388.2	4.14	267.4
getTransform	402.0	4.31	270.5
getSellBulk	402.4	4.94	261.6

TABLE II: Performance of TC3 No Consolation Query Operations

Operation	Send Rate (TPS)	Avg. Latency (s)	Throughput (TPS)
getCreate	403.8	3.07	294.6
getSell	402.7	3.55	287.7
getTransform	407.4	3.25	292.7
getSellBulk	407.3	3.47	288.9

VI. Discussion

The performance evaluation of *EvoChain*WineTracker considered three aspects: core functionalities, cancel transactions (CTs), and query transactions, comparing the baseline WineTracker and the *EvoChain*-enhanced version.

For core functionalities (TC1), the modified system exhibits slightly higher latency and lower throughput than the baseline. Average latency increased from 30.19 to 35.61 seconds per transaction (17.95%), and throughput decreased from 180.67 TPS to 163.83 TPS (9.32%). Memory usage also increased modestly by 2.21%, from 459.39 MB to 469.54 MB. These differences are primarily due to the larger transaction sizes and additional computations introduced by the view generation algorithm (Figure 4).

Cancel transactions (TC2) impose additional overhead, as each CT recursively updates the state of dependent transactions. This extra computation, combined with validation steps, reduces throughput relative to standard operations. Early rounds show slightly lower throughput due to a cold-start effect, but steady-state performance stabilizes thereafter. Over time, the accumulation of canceled and mutable transactions increases blockchain height, which requires more storage and computation, slightly affecting confirmation time and throughput. Importantly, CTs do not directly impede core transactions, but in systems with complex or large objects, view generation may introduce noticeable processing delays.

Query transactions (TC3) further highlight the overhead introduced by state-altering operations. Queries that consolidate pending transactions after their mutation delay show increased latency and reduced throughput compared to queries that do not alter transaction states. Average latency rose from 3.34 s to 4.46 s (33.5%), while throughput decreased from 290.98 TPS to 264.35 TPS (9.15%) (Tables I and II). This behavior is expected, as consolidation requires confirmation and evaluation of pending transactions within the view generation process.

Overall, the evaluation demonstrates that while *EvoChain* introduces some overhead in latency, throughput, and memory usage, the trade-off is acceptable given the enhanced capabilities for transaction redaction and view generation. The framework does not weaken Hyperledger Fabric's core security properties, including consistency, tamper-resistance, double-spending prevention, DDoS resistance, and pseudonymity, as it operates entirely within the standard Fabric architecture while providing more flexible transaction management.

VII. Conclusion

In this work, we reviewed the properties of blockchains and how cryptographic primitives support decentralization, transparency, and data immutability. While these characteristics are highly needed in domains requiring trust – such as financial services, healthcare, or supply chains – the strict immutability of data can limit practical adoption, as many systems need to correct mistakes or mitigate malicious intrusions.

We proposed *EvoChain*, a framework for chaincode development that introduces controlled mutability. By using a transaction model with access control policies, it allows grace periods during which data can be corrected or recovered. This approach enhances flexibility, enabling restoration of objects affected by unwanted operations.

A prototype supply chain application was implemented with and without *EvoChain* to assess the performance impact. The framework introduced a modest overhead: latency increased by around 15%, and throughput decreased by roughly 10%. Cancel transactions contribute to lower throughput due to extra computation and validation, though these operations are expected to be exceptional. Although storage and processing needs increase, these effects are limited and offset by the benefits of enhanced transaction control.

Overall, *EvoChain* demonstrates that controlled mutability can enable blockchains to handle errors and recover from intrusions while preserving security and trust. By providing this flexibility, the framework makes blockchain technology more practical for real-world applications without compromising its inherent security guarantees.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] Adeleke Ayobami, "Industry News 2024 How Blockchain Technology is Revolutionizing Audit and Control in Information Systems." https://www.isaca.org/resources/news-and-trends/industry-news/2024/how-blockchain-technology-is-revolutionizing-audit-and-control-in-information-systems, 2024. [Accessed 01-11-2024].

[3] B. Biswas and R. Gupta, "Analysis of barriers to implement blockchain in industry and service sectors," *Computers & Industrial Engineering*, vol. 136, pp. 225–241, Oct. 2019.

[4] S. Jabbar, H. Lloyd, M. Hammoudeh, B. Adebisi, and U. Raza, "Blockchain-enabled supply chain: analysis, challenges, and future directions," *Multimedia Systems*, vol. 27, pp. 787–806, Aug. 2021.

[5] E. A. Politou, F. Casino, E. Alepis, and C. Patsakis, "Blockchain mutability: Challenges and proposed solutions," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, pp. 1972–1986, 2021.

[6] D. Zhang, J. Le, X. Lei, T. Xiang, and X. Liao, "Exploring the redaction mechanisms of mutable blockchains: A comprehensive survey," *International Journal of Intelligent Systems*, vol. 36, no. 9, pp. 5051–5084, 2021.

[7] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, no. 1, pp. 94–162, 1992.

[8] H. F. Korth, E. Levy, and A. Silberschatz, "A formal approach to recovery by compensating transactions," tech. rep., USA, 1990.

[9] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 99, pp. 173–186, 1999.

[10] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform." https://ethereum.org/en/whitepaper/, 2014. Accessed: 2025-09-03.

[11] 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. Weed Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–15, 2018.

[12] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Annual international cryptology conference*, pp. 139–147, Springer, 1992.

[13] A. Back, "Hashcash - a denial of service counter-measure," 2002.

[14] P. Vasin, "Blackcoin's proof-of-stake protocol v2," *URL: https://blackcoin. co/blackcoin-pos-protocol-v2-whitepaper. pdf*, vol. 71, 2014.

[15] D. Ago, "DPOS Consensus Algorithm - The Missing White Paper," May 2017.

[16] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology — CRYPTO '87* (C. Pomerance, ed.), (Berlin, Heidelberg), pp. 369–378, Springer Berlin Heidelberg, 1988.

[17] S. Al-Kuwari, J. H. Davenport, and R. J. Bradford, "Cryptographic hash functions: Recent design trends and security notions," *Cryptology ePrint Archive*, 2011.

[18] C. Jentzsch, S. It, C. Jentzsch, and S. It, "Decentralized Autonomous Organization To Automate Governance," p. 31. Retrived from https://lawofthelevel.lexblogplatformthree.com/wp-content/uploads/sites/187/2017/07/WhitePaper-1.pdf.

[19] G. Ateniese, B. Magri, D. Venturi, and E. R. Andrade, "Redactable blockchain – or – rewriting history in bitcoin and friends," *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 111–126, 2017.

[20] I. Puddu, A. Dmitrienko, and S. Capkun, "μchain: How to forget without hard forks," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 106, 2017.

[21] "Hyperledger – Open Source Blockchain Technologies."

[22] "Hyperledger Caliper – Hyperledger Foundation."

[23] J. R. Douceur, "The Sybil Attack," in *Peer-to-Peer Systems* (P. Druschel, F. Kaashoek, and A. Rowstron, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 251–260, Springer, 2002.

[24] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *IEEE P2P 2013 Proceedings*, pp. 1–10, 2013.

[25] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on Bitcoin's Peer-to-Peer network," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 129–144, USENIX Association, Aug. 2015.

[26] A. Gervais, G. O. Karame, V. Capkun, and S. Capkun, "Is bitcoin a decentralized currency?," *IEEE Security & Privacy*, vol. 12, pp. 54–60, 2014.

[27] K. Biswas, V. Muthukkumarasamy, and W. Lum, "Blockchain Based Wine Supply Chain Traceability System," Nov. 2017.

[28] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (USA), p. 305–320, USENIX Association, 2014.