DOLE
Ditto Offline-first Ledger Exchange

Dominik Kipfer and Julian Wanninger

University of Basel
New Trends for Local and Global Interconnects for P2P Applications (76295)
February 8, 2026

Abstract

Modern digital payment systems rely heavily on continuous internet connectivity and central intermediaries. In offline scenarios or partitioned networks, these systems fail because preventing double-spending traditionally requires global consensus mechanisms.

This project introduces a novel protocol for secure, offline-capable peer-to-peer payments. The proposed system adapts the theoretical framework of the GOC-Ledger, as proposed by Erick Lavoie [7], and integrates it with certified hardware acting as trust anchors. Rather than relying on resource-intensive consensus algorithms, our approach leverages tamper-proof hardware to locally enforce strict monotonic counters.

The implementation utilizes cryptographically chained Append-Only Logs for transport, distributed via the Ditto mesh framework. By strictly enforcing the protocol's rules at the source, our hardware-based solution ensures system integrity without the need for post-hoc reconciliation. Using an issuer-centric provisioning model to establish a root of trust, the resulting system combines the resilience of physical cash with the auditability and transparency of digital ledgers, functioning completely independently of internet infrastructure.

1 Introduction

Digital payment systems, ranging from traditional credit cards to mobile solutions like TWINT or Apple Pay, have become the standard for transactions in everyday life. However, these systems fundamentally rely on continuous connectivity to a central infrastructure or the internet. In scenarios such as crowded music festivals, basement venues with poor reception, or disaster relief situations where connectivity is unavailable, the ability to transfer value digitally collapses. Consequently, there remains a lack of a resilient digital equivalent to physical cash that operates securely in offline environments.

Creating such a peer-to-peer payment system while preventing double-spending and replay attacks presents a significant challenge in distributed computing. In traditional settings, a central bank or a distributed ledger relying on global consensus prevents a user from spending the same token twice. However, in an offline, partitioned network, global consensus is impossible to achieve in real-time. This creates a security gap where malicious actors could exploit network delays to bypass protocol rules, for instance, by defrauding merchants or other users by spending funds they do not possess.

To address the limitations of consensus-based models, recent research has explored state-based conflict-free replicated data types (CRDTs). Specifically, this project builds upon the theoretical framework of the "GOC-Ledger" proposed by Lavoie [7]. This model utilizes Grow-Only Counters to track value transfers, ensuring that ledger states can mathematically converge without the need for complex consensus mechanisms. While the GOC-Ledger offers a robust mathematical foundation for convergence, pure software implementations in adversarial environments face challenges regarding strict safety. As noted in the foundational research, ensuring non-negative balances remains difficult, potentially allowing temporary overspending or history forking.

DOLE, the project presented in this report, aims to bridge this gap. We propose a hybrid architecture that leverages the GOC-Ledger for network synchronization while anchoring the security in Java Cards, functioning as trusted hardware. By moving the critical logic of solvency checks and transaction sequencing into a tamper-proof secure element, protocol rules are physically enforced at the source. This approach effectively eliminates the possibility of double-spending while guaranteeing replay protection.

The remainder of this report is structured as follows: Section 2 details the system design and architecture, while Section 3 documents the concrete technical implementation. Section 4 provides a critical discussion of system boundaries and challenges, followed by an outlook on potential future extensions. Finally, Section 5 presents the conclusion of our project.

2 System Architecture

2.1 Theoretical Foundation: The GOC-Ledger

The theoretical foundation of DOLE is built upon the GOC-Ledger model, as introduced in the research paper "State-based Conflict-Free Replicated Ledger from Grow-Only Counters" by Lavoie [7].

Traditional Distributed Ledger Technologies such as Bitcoin or Ethereum utilize consensus mechanisms like Proof-of-Work or Proof-of-Stake to maintain a unified global ledger state across all network participants. However, this approach is not only resource-intensive but also practically unfeasible in offline scenarios or partitioned networks. Consensus mechanisms require continuous real-time communication, as they rely on a global agreement regarding the ordering of transactions. In the event of connectivity loss, divergent views of the ledger state (forks) can occur. Since these divergences cannot be resolved without reconnecting to the majority of the network, transaction finality is impossible, rendering the payment system effectively non-functional in isolated environments.

To address this limitation, DOLE adapts the concept of Conflict-Free Replicated Data Types (CRDTs) to financial transactions. The system leverages the property of Strong Eventual Consistency, which mathematically guarantees that the ledger converges to a consistent state, independent of the order of data exchanges and without the need for a reliable broadcast mechanism.

2.1.1 Accounts

DOLE implements an account-based model that differs fundamentally from traditional approaches regarding state management. If a participant's balance were stored as a simple, mutable variable, concurrent offline transactions would inevitably lead to synchronization conflicts, as determining the valid version of the balance would be ambiguous. To circumvent this issue, the account state is decomposed into Grow-Only Counters (GOCs).

Formally, a GOC is defined as a data structure subject to strict monotonicity: For any state transition from a value v to a new value v', the condition $v' \geq v$ must always hold. By definition, a reduction of the counter value is impossible, thereby enabling the subsequent conflict-free merging of divergent states [7].

Algorithm 1 Initialization and structure of an account (adapted from Lavoie [7])

```
1: function INITIALIZE_A(id)
2:     A_id ← id                    ▷ Unique ID of the account (Hash of Public Key)
3:     A_↑ ← 0                      ▷ minted: Cumulative sum of tokens created
4:     A_↓ ← 0                      ▷ burned: Cumulative sum of tokens destroyed
5:     A_→ ← {}                     ▷ sent: Map [Receiver_ID → Cumulative Amount]
6:     A_← ← {}                     ▷ received: Map [Sender_ID → Cumulative Amount]
7:     return A
8: end function
```

As shown in Algorithm 1, the state of an account A is defined by three central components:

1. **Account-ID (A_{id}):** Each account is assigned a unique identifier upon creation. In our implementation, this ID is deterministically derived from the hash of the account's cryptographic public key.
2. **Scalar Counters ($A_{\uparrow}, A_{\downarrow}$):** The fields *minted* and *burned* capture the cumulative creation and destruction of tokens. Since these operations do not require a specific counterparty within the network, they are stored as simple scalar values.

3. **Vector Maps ($A_{\rightarrow}, A_{\leftarrow}$):** The fields *sent* and *received* are implemented as dictionaries. Unlike a simple total balance, they store the cumulative transaction value granularly for each interaction partner (referenced by their ID).

2.1.2 Operations and State Transitions

In our model, accounts can perform three core operations: `MINT`, `BURN`, and `SEND`. While the former two operations only modify the local state, the send operation implies an interaction between two distinct parties.

Mint and Burn

Privileged accounts (Minters) can create new tokens, while all users can burn tokens. These operations strictly affect the scalar counters of the account. If an account A mints an amount *amount*, the resulting state A' is defined as:

$$A'_{\uparrow} \leftarrow A_{\uparrow} + amount$$

Analogously, the counter $A_{\downarrow}$ is incremented when tokens are *burned* [7].

Send

The `SEND` operation transfers a specified *amount* from a sender S to a receiver R with the associated ID R_{id}. The state transition occurs within the vector maps, where the sender's output counter is incremented at the receiver's index by the transferred amount:

$$S'_{\rightarrow}[R_{id}] \leftarrow S_{\rightarrow}[R_{id}] + amount$$

To maintain system integrity and prevent negative balances, this operation is only permissible if the invariant $amount \leq \text{Balance}(S)$ is satisfied.

Difference to the Reference Model

A significant distinction from Lavoie's theoretical approach [7] lies in the handling of fund reception. While the original paper requires the receiver to actively confirm receipt via a dedicated operation (`AckFrom`), we treat reception as an implicit consequence of the (`Send`) Operation.

As soon as sender S has updated and published their status $S_{\rightarrow}$, the amount is computationally considered booked by the receiver R. Consequently, the receiver's virtual state updates as follows:

$$R'_{\leftarrow}[S_{id}] \leftarrow R_{\leftarrow}[S_{id}] + amount$$

This optimization, which we term the No-ACK Model, significantly reduces the number of required log entries within the network, as the possession of a signed SEND log serves as a sufficient Proof-of-Payment.

Convergence

A key advantage of this model is guaranteed, conflict-free synchronization. When two nodes in the mesh network hold different versions of the same account state (A and A'), a new state A'' is deterministically derived.

This process follows the Maximum Rule: Mathematically, the new state A'' corresponds to the Least Upper Bound of the two initial states within the state space. Formally, the merge operation $A'' \leftarrow A \sqcup A'$ is realized through the element-wise maximum of all contained counters.

For the scalar counters *minted* and *burned*, the respective higher value is adopted:

$$A''_{\uparrow} \leftarrow \max(A_{\uparrow}, A'_{\uparrow}) \quad \text{and} \quad A''_{\downarrow} \leftarrow \max(A_{\downarrow}, A'_{\downarrow})$$

For the vector maps, represented as dictionaries, the entries for sent ($A_{\rightarrow}$) and received ($A_{\leftarrow}$) amounts are compared key-wise:

$$\forall id : A''_{\rightarrow}[id] \leftarrow \max(A_{\rightarrow}[id], A'_{\rightarrow}[id])$$
$$\forall id : A''_{\leftarrow}[id] \leftarrow \max(A_{\leftarrow}[id], A'_{\leftarrow}[id])$$

In a Grow-Only system, a higher counter value implies that more events have occurred. Therefore, the maximum always represents the most current state of information and overwrites outdated values. This ensures that all replicas in the network eventually converge to the identical state. [7]

Derivation of Solvency

In this model, the effective available balance of an account is not a persistent state variable, rather, it is a projection derived from the counter states. Specifically, the balance is computed by subtracting the sum of all outflows (*burned* and *sent*) from the sum of all inflows (*minted* and *received*). Consequently, the solvency of a participant A is dynamically determined by the following formula:

$$\text{Balance}(A) = \left(A_{\uparrow} + \sum A_{\leftarrow}\right) - \left(A_{\downarrow} + \sum A_{\rightarrow}\right)$$

In our project, this mathematical model serves as the foundational logic for transaction validation. While the theoretical model defines how the state is computed, the physical enforcement of monotonicity and thus the protection against counter manipulation is realized through the hardware architecture described in Section 2.4.

2.1.3 Ledger

The ledger L represents a node's local, subjective view of the network by maintaining the current state of all known accounts. Formally, L is a Grow-Only Dictionary mapping unique account IDs to their respective states.

The synchronization of the local ledger L with that of another participant L' is performed via the merge operation $L'' = L \sqcup L'$, where L'' is the resulting, updated ledger. For each account identifier id, the process is defined as follows:

1. **Merging new Accounts:** If an account exists in L' but is currently unknown in the local ledger L ($id \in L' \wedge id \notin L$), its state is directly copied into the new ledger L'':

$$L''[id] \leftarrow L'[id]$$

2. **Merging Existing Accounts:** If an entry for id exists in both ledgers, the new state is derived using the account-wise merge operation defined in the Convergence section ($A \sqcup A'$). The ledger stores the result, which combines the most up-to-date information from both parties:

$$L''[id] \leftarrow L[id] \sqcup L'[id]$$

This mechanism ensures that global knowledge within the network grows monotonically. It guarantees that all participants eventually reach the identical state (Strong Eventual Consistency), regardless of the order of data exchange and without the need for global consensus.

While the GOC model establishes the mathematical rules for state transitions and solvency calculation, it does not specify how these states are stored securely in a distributed, trustless network. To ensure tamper resistance and causal ordering, the abstract state transitions must be mapped to a persistent structure. The following section introduces Append-Only Logs, which serve as the immutable transport medium in our architecture for propagating these state changes.

2.2 Data Structure: Append-Only Logs

To guarantee a unique, tamper-proof history in an environment lacking a central authority and to lay the foundation for subsequent cryptographic enforcement (see Section 2.4), abstract state transitions must be mapped to a persistent, linearly ordered structure. Consequently, we implement Append-Only Logs as the Single Source of Truth for propagating ledger updates in our network.

In this architecture, the log functions as an immutable container for data exchange via Ditto [2], while the actual ledger state acts as a dynamic projection derived from these logs.

2.2.1 Log Chain Structure


```
[
  {
     "_id":"a11ce0000000000000000000000000000000001_0",
     "seq":0,
      "prevHash": "0000000000000000000000000000000000000000000000000000000000000000",
     "type":"GENESIS",
     "author":"a11ce0000000000000000000000000000000001",
     "target":"0000000000000000000000000000000000000000",
     "goc":0,
      "signature": "30450221008d5c...",
      "publicKey": "04a2c3b4...",
      "certificate": "304402201b..."
  },
  {
     "_id":"a11ce0000000000000000000000000000000001_1",
     "seq":1,
      "prevHash": "04474396ecdda07a64f9272356d18e1af6ace9c2e4339b37ba483618a164c417",
     "type":"SEND",
     "author":"a11ce0000000000000000000000000000000001",
     "target":"b0b0000000000000000000000000000000000002",
     "goc":150,
      "signature": "304402206c9a..."
  }
]
```


Fig. 1: Exemplary Log Chain of user Alice (`a11ce...`).

As illustrated in Figure 1, a log entry consists of two primary components: the signed log header and the appended metadata (*_id*, *signature*, *certificate*).

Log Header

Although the JSON representation implies a flat hierarchy as shown in Figure 1, we conceptually distinguish between the Log Header, which contains signed protocol data, and metadata. The header maintains a strictly static length of 81 bytes. It is composed of the following six attributes:

1. **Sequence (`seq`, 4 Byte):** A continuous, zero-indexed numbering of logs, defining the relative temporal order within the chain.
2. **Previous Hash (`prevHash`, 32 Byte):** The SHA-256 hash of the header of the preceding log entry.
3. **Type (`type`, 1 Byte):** An identifier indicating the type of the transaction.
4. **Author (`author`, 20 Byte):** The ID of the transaction initiator.

5. **Target (`target`, 20 Byte):** The ID of the addressee (if applicable).
6. **GOC Value (`goc`, 4 Byte):** The most current Grow-Only Counter value in the relationship between the author and target at the time of creation.

The Single-Chain Principle & Integrity

We adhere to the Single-Chain Principle, wherein every user maintains exactly one unique, linear log chain. The integrity of this chain is secured via hash-chaining: every entry cryptographically references its predecessor. As seen in Figure 1, the log with $seq = 1$ contains the hash of the previous log with $seq = 0$. This chaining guarantees the immutability of the history, as any retrospective modification of a past block would invalidate all subsequent hashes.

Identities & Padding

To reference accounts, we utilize a 20-byte ID derived from the first 20 bytes of the SHA-256 hash of the account's public key. This provides a compromise between storage efficiency and collision resistance.

Since certain transaction types (e.g., `GENESIS`) do not require target addresses or specific values, we employ Zero Padding: Unused fields in the header are filled with zeros. This guarantees a static block size of exactly 81 bytes, which significantly simplifies parsing and signing operations on the resource-constrained hardware.

Metadata and Attachments

The static log header is wrapped in a JSON container and complemented by variable metadata required for transport and verification.

1. **Database Identifier:** The underlying database requires a unique primary key for every document (`_id`). We deterministically derive this key by concatenating the author's ID and the sequence number (e.g., `author_seq`). This serves as an index for storage and retrieval and is not included in the cryptographic signature generated by the hardware.
2. **Signature:** Every log entry contains the `signature` field. The account signs the static 81-byte header using its private key (ECDSA). This signature fulfils two critical functions simultaneously: First, it ensures Authenticity, allowing every network participant to mathematically verify that the log entry has not been modified and was created by the owner of the author ID. Second, it serves as a Proof-of-Hardware-Enforcement: Since the private key never leaves the hardware's Secure Element (see Section 2.4), a valid signature implicitly proves that the transaction successfully passed the hardware-enforced protocol rules, such as monotonicity and solvency checks.
3. **Identity Attachments:** To establish a Chain of Trust without a centralized database, the `GENESIS` transaction carries additional fields in its attachment section. This includes the complete $publicKey$ of the account, which is necessary as the header stores only its hash for efficiency reasons. Furthermore, it contains the $certificate$, which is a digital signature over the public key issued by the smartcard manufacturer. This certificate serves as cryptographic proof that the participant is utilizing certified, tamper-proof hardware rather than a software emulation.

2.2.2 Transaction Types

The abstract state transitions of the GOC model are operationalized in the protocol through four specific transaction types, which are encoded in the header via the *type* field.

GENESIS

This transaction constitutes the origin ($seq = 0$) of every log chain. Its primary purpose is initiating the trust relationship by establishing the account's digital identity within the network.

To achieve this, the complete public key and the device certificate (the manufacturer's signature over this key) are published in the transaction's attachments. This enables other network participants to validate hardware authenticity and register the new account as a trusted entity in their local ledger.

To minimize data overhead within the network, the `GENESIS` transaction is the only type that carries this extensive identity data. All subsequent transactions reference the account solely via the storage-efficient *id*.

An example of a Genesis transaction can be observed in Figure 1. Here, Alice registers herself in the network via the first log in her chain by propagating her public key and the certificate. By definition, a Genesis transaction has no predecessor and transfers no value. Consequently, the fields $prevHash$, $target$ and goc are logically empty. To preserve the static header structure defined in Section 2.2.1, these fields are filled with Zero Padding.

MINT and BURN

These two transaction types serve to modify the scalar counters for money creation ($A_{\uparrow}$) and destruction ($A_{\downarrow}$), as defined in Section 2.1.

As stipulated in Lavoie's theoretical model [7], there are privileged accounts that are exclusively authorized to create new tokens. In practice, this role is assumed by currency exchanges or banks acting as gateways for the entry and exchange of physical cash. To ensure transparency and auditability, Minters are also subject to strict protocol rules: A Minter cannot generate funds implicitly. Instead, they must make these funds visible within the network through an explicit `MINT` transaction prior to any transfer. Therefore, if a Minter intends to send tokens, they must first increment their *created* counter, ensuring the consistency of the global money supply.

As a complementary operation, `BURN` allows every participant to permanently remove funds from circulation, thereby reducing the Total Supply. While destroying one's own assets may appear economically irrational, it is an essential mechanism for the system design, particularly for cash-outs: If a user exchanges digital tokens back for physical cash at a Minter, the Minter must burn the received tokens to adjust the digital money supply to align with the reduced physical backing.

Since creation and destruction are endogenous state changes that do not address a specific receiver in the network, the *target* field in the log header remains unused. To preserve the static log structure, this field is filled with Zero Padding, analogous to the `GENESIS` transaction.

SEND

The `SEND` transaction facilitates the actual value transfer between two parties. A log's *author* transfers tokens to a specific account, which is addressed in the header via the *target* field.

Consistent with the CRDT model, *goc* does not store the individual transaction amount, but rather the new cumulative total value ($A_{\rightarrow}$[target]) of all transactions the *author* account has ever sent to this specific receiver. Since this represents the protocol's only exogenous operation, `SEND` is the sole transaction type that populates the *target* field with a valid 20-byte ID.

Figure 1 illustrates this process in the log entry with $seq = 1$. Here, Alice confirms that the current GOC value in her relationship with Bob is now 150. The effective transaction value (Δgoc) is always derived from the difference relative to the previous state:

$$\Delta goc = GOC_{\text{new}} - GOC_{\text{old}}$$

Since the preceding log ($seq = 0$) is the `GENESIS` transaction, implying no prior transfers existed ($GOC_{\text{old}} = 0$), this new counter value corresponds to an effective transfer of 150 tokens ($150 - 0$).

The digital signature over the header guarantees authenticity: It prevents Identity Spoofing and ensures that no malicious actor in the network can propagate incorrect or forged send events in the author's name.

2.2.3 Mesh Network Transport: Ditto Live

To transport and synchronize the Append-Only Logs in an offline environment, we utilize the Ditto Live framework [2]. Ditto functions as a resilient transport layer, dynamically establishing Peer-to-Peer connections via Bluetooth Low Energy (BLE) or WiFi Aware to create an ad-hoc mesh network. Data propagation is managed via a gossip protocol that automatically replicates the JSON-encapsulated logs as soon as devices are within range.

In our security model, Ditto acts as a container: The framework solely ensures data availability, while transaction integrity and authenticity are guaranteed independently of the transmission channel through the Java Card's cryptographic signatures.

2.3 The No-ACK Optimization

As outlined in Section 2.1, our protocol doesn't use explicit receiver `ACK` logs. Instead, the signed `SEND` logs function as an implicit Proof-of-Payment. This fundamentally shifts the system's definition of an account's balance: Solvency is not determined by an entry within one's own chain, but rather by the possession of third-party `SEND` logs addressed to one's own ID.

Referring back to Figure 1, Alice transfers an amount of 150 tokens to Bob in the entry with $seq = 1$. Since the protocol strictly enforces the invariant

$$\text{Balance(Alice)} \geq amount$$

and Alice, as a standard user, cannot execute `MINT` transactions, the existence of this log implies, that Alice must have previously received at least 150 tokens.

Figure 2 illustrates the corresponding proof: A log entry from a third party exists that addresses Alice as the *target*.

```
{
    "_id":"29913b432db8441ab3319f3201dfe3440b56fbf1_50",
    "seq":50,
    "prevHash": "9f3c2b6e4a8d71c5f0e2a94b7d6c1f83a5e4b2d0c9a7f6e8d3c1b4a592e7f0d6",
    "type":"SEND",
    "author":"29913b432db8441ab3319f3201dfe3440b56fbf1", // A Third Party (Sender)
    "target":"a11ce0000000000000000000000000000000001", // Alice (Receiver)
    "goc":200,                                          // Alice has received 200 in total
    "signature": "304602..."
}
```

Fig. 2: Log of a `SEND` transaction with Alice as the target.

Advantages of the Model

This architecture offers significant advantages over traditional approaches:

Reduced Storage and Computational Overhead

Omitting explicit `ACK` logs significantly reduces the total volume of records stored and synchronized within the network (by approximately 50%), as the `SEND` log implicitly contains the proof of receipt. This yields direct improvements in system performance:

- **Storage Space:** Log chains grow at a slower rate, which is particularly critical for mobile devices with limited storage capacity.
- **Computational Load:** Since every transaction requires cryptographic verification, the number of computationally expensive signature checks required by both the hardware and the client devices is halved.

- **Network Load:** Synchronization latency and required bandwidth within the mesh network are reduced accordingly. Consequently, the model is resource-efficient and optimized for low-bandwidth environments.

Censorship Resistance

A further critical advantage lies in ensuring an open, censorship-resistant monetary system via the principle of non-interactive receiving.

In systems requiring active receiver consent, powerful actors or regimes could enforce a blacklisting infrastructure to economically isolate specific demographic groups. The act of explicitly accepting a payment from a sanctioned individual could be construed as active support and subsequently prosecuted.

In our model, the receiver's wallet acts as a passive entity that is technically incapable of rejecting incoming payments. Since reception occurs automatically and without user intervention, the leverage for third parties to pressure the receiver into blocking a transaction is eliminated. This preserves the neutrality of payment transactions, particularly within politically unstable environments.

Consistency of the Money Supply

The implicit sending model guarantees the mathematical invariant of the global money supply. In systems requiring an active acknowledgement, a state of uncertainty arises during the interval between sending and confirmation.

Consider a scenario with explicit acknowledgement: Suppose Alice holds a balance Y given a total money supply of X. She transfers her entire balance Y to Bob, but he fails to confirm receipt (e.g., due to network failure). Consequently, the following holds for the local balances of both accounts:

$$\text{Balance}_{\text{Alice}} = 0, \quad \text{Balance}_{\text{Bob}} = 0$$

The circulating supply of the network effectively drops to $Z = X - Y$, even though no reduction of the supply via `BURN` transactions has occurred. The missing amount Y becomes "shadow money", theoretically existing within the total supply but absent from circulation.

The No-ACK design renders this inconsistent state impossible. Since the debiting of the sender's account and the computational crediting of the receiver's account are logically coupled to the same atomic proof, the `SEND` log, the sum of all balances corresponds exactly to the issued money supply minus burned tokens at all times. Modifications to the Total Supply are therefore transparently restricted to `MINT` and `BURN` operations.

2.4 The Role of Hardware

The integrity of the architecture described in the preceding sections relies on a fundamental premise: protocol rules must be strictly enforced, even within a trustless environment. In a pure software implementation on a commodity device, this cannot be guaranteed, as attackers could circumvent software logic through manipulation of the operating system or the program code.

To close this security gap, we offload critical operations to tamper-proof hardware (Java Programmable Cards), which functions as a physical Root of Trust. This hardware component is essential for the validity of the data structure: For the Append-Only Logs defined in Section 2.2 to be accepted by other network participants, they must be signed with the private key isolated within the Card's Secure Element. To achieve this, the card internally generates the static 81-byte log header and outputs it coupled with the cryptographic signature.

For our implementation, we utilize the NXP J3R180 model [5]. This chip is based on certified Secure Element technology and features physical protection mechanisms that effectively prevent the readout of memory contents or the modification of executed code according to the current state of the art (tamper-proof).

2.4.1 Hardware as a Trusted Execution Environment

The Java Card provides a Trusted Execution Environment that is completely isolated from the smartphone's operating system. This architecture affords fundamental security guarantees that would be otherwise unachievable:

Key Isolation

The smartcard acts as a Trust Anchor by generating the cryptographic key pair directly on-chip. The private key never leaves the protected memory area of the Secure Element at any point. This establishes an effective security boundary against the smartphone: even if the host operating system is compromised by malware, the key cannot be extracted, as all cryptographic operations are performed in isolation within the chip.

Additionally, access to these signing functions is protected by a user-defined, 4-digit PIN code. This ensures that even in the event of physical loss, third parties cannot gain access to the digital identity or funds, as the card computes signatures only upon explicit request and subsequent to successful PIN verification.

Atomic Execution

A decisive advantage of the Java Card environment is transactional integrity as defined in the runtime specification [12]. The execution of the protocol logic occurs atomically within the enclave. This guarantees that cryptographic operations are inseparably coupled with the logical state updates.

It is technically impossible to generate a signature for a `SEND` log without incrementing the internal GOC counter ($totalSent$) in the exact same atomic step. In the event of a power loss during the process, the entire state is automatically rolled back to the last valid value. This mechanism effectively excludes inconsistent states or the creation of ghost transactions.

Binding Identity to Protocol Logic

Since the private key is non-exportable and intrinsically tied to the specific hardware instance, an implicit binding exists between the digital identity (Public Key) and the code installed on the hardware. An adversary cannot migrate the key to a different hardware environment running manipulated code to circumvent protocol rules. Consequently, a valid signature serves as proof not only of the author's identity but also as a guarantee that the transaction was processed by the uncompromised, verified code of the smartcard.

2.4.2 Protocol Enforcement

The smartcard operates not merely as a passive key storage but acts as a state machine that actively enforces adherence to protocol rules within an isolated environment. The following invariants are guaranteed by the hardware:

Enforced Sequencing

The card internally maintains a strictly monotonically increasing sequence counter seq, which is persisted in non-volatile memory. This counter is incremented with every transaction. Since write access to this memory region is restricted exclusively to the applet, resetting or skipping sequence numbers is impossible. This mechanism effectively prevents Forking, as the hardware is technically incapable of signing two distinct log entries with the identical sequence number.

Internal Hash Chaining

The card calculates and internally stores the hash of the most recently signed log ($prevHash$). This value serves as a mandatory input for the construction of the subsequent block. Neither an attacker nor a compromised smartphone can inject an arbitrary predecessor hash to force the validation of an alternative history. This guarantees the cryptographic integrity of the log chain at its source.

Solvency

To enforce the system's economic rules, the card persists the scalar counters (*minted*, *burned*) and the aggregated vectors (*sent*, *received*) within its protected, non-volatile memory as defined in Section 2.1.

Prior to signing any log, the firmware atomically verifies whether the solvency invariant $\text{Balance}(A) \geq \textit{amount}$ is satisfied. Only if this condition holds, is the transaction signed and the internal counter updated. This atomic coupling of state transition and signature generation effectively prevents Double-Spending at the source: It is technically impossible to obtain a valid signature without the corresponding funds being simultaneously deducted internally. Any external tampering with the log (e.g., modifying the recipient address) would invalidate the cryptographic signature and cause rejection by the network.

Replay Protection

To prevent the repeated processing of the same incoming transaction, the card internally stores the last known GOC state for every communication partner within its *sent* and *received* vectors.

When crediting an incoming log, the card enforces strict monotonicity: A log is accepted only if the contained GOC value is strictly greater than the value already stored for that sender:

$$GOC_{\text{new}} > GOC_{\text{stored}}$$

This hardware-enforced constraint guarantees that funds are credited exactly once and that stale or duplicated packets remain ineffective.

2.5 Issuer-Centric Provisioning Model

For our network to operate securely without centralized supervision, every participant must be able to cryptographically prove two fundamental properties to third parties:

1. **Hardware Integrity:** The usage of valid, tamper-proof hardware (Genuine Hardware).
2. **Software Integrity:** The execution of unmodified, verified protocol code (Code Attestation).

Algorithmic fraud is effectively precluded only if both conditions are met simultaneously. To ensure this, DOLE implements an Issuer-Centric Provisioning Model, aligning with traditional smart card management standards [11]. In this approach, smartcards are not distributed as blank cards to be configured by the user. Instead, they are fully pre-initialized within a controlled environment.

The Initialization Process

In our implementation, we simulate the role of a Central Trust Authority (the card manufacturer), which retains custody of the smartcards prior to their distribution to end-users. This Pre-Initialization workflow proceeds as follows:

1. **Key Generation:** First, the manufacturer installs the immutable applet onto the card. Subsequently, the card is instructed to internally generate a new, unique key pair. Since this operation occurs within the card's Secure Element, it guarantees that the private key never existed outside the hardware boundary [5] and thus could not be copied.
2. **Certification:** The associated public key is exported and digitally signed by the manufacturer using their own Root Private Key. The resulting signature, termed the Device Certificate, is then written back into the card's persistent memory.

From this point forward, this certificate acts as an inseparable digital seal: It mathematically proves that the public key residing on the card belongs to a hardware instance that was authorized by the manufacturer and provisioned with the correct code. Should a user subsequently attempt to delete or manipulate the applet, the security mechanisms of the Java Card platform would automatically eradicate the cryptographic keys, thereby rendering the certificate invalid.

Network Integration and Result

Upon joining the network, a user propagates the `GENESIS` log. Within this log, the card publishes its public key along with the Device Certificate. By signing this log using the internal private key, the card furnishes a Proof-of-Possession: It demonstrates physical ownership of the exact key certified by the manufacturer.

This mechanism fundamentally shifts the trust model: A participant (e.g., Bob) is not required to trust an unknown peer (e.g., Alice). Instead, validation is reduced to checking the digital signature of the manufacturer. If valid, the hardware architecture transitively guarantees that Alice enforces the protocol rules. The tamper-proof nature of the hardware effectively prevents certificates from being copied or misappropriated for unauthorized purposes.

3 Implementation

While Section 2 outlined the theoretical foundations and architectural design decisions of DOLE, this chapter focuses on the concrete technical realization. In the following sections, we describe the software architecture through its core modules. Our focus lies not only on the functional logic and class design but particularly on the specific technical challenges encountered in practice and the concrete solutions developed to overcome them.

3.1 Project Organization

The solution is implemented as a multi-module Gradle project, strictly separating embedded hardware code from client-side logic.

We utilise Java 21 for the client applications to leverage modern language features, while the smart card modules maintain compatibility with Java 8 (Java Card Classic Edition) for deployment on the secure element. The project consists of five sub-projects:

- `common`: Shared data structures and protocol definitions.
- `card`: The embedded Applet code running on the smart card chip.
- `client-core`: Shared business logic, cryptography, and state management.
- `pc-app`: Entry point and hardware abstraction for desktop operating systems.
- `android-app`: Integration with the Android OS and NFC hardware.

3.2 Module: Common

This module acts as a bridge between the smart card environment and the application logic. It is a pure Java library defining shared data structures and constants in `dole.Constants`.

Challenges and Solutions

The main challenge was ensuring binary compatibility between the modern Java 21 desktop application and the restricted Java Card runtime. The card does not support standard Java features like complex Enums.

To address this, we restricted the `common` module to primitive data types (bytes, shorts) and static final definitions. This ensures that both the client and the card use the exact same binary format, preventing linking errors on the hardware.

3.3 Module: Card (Smart Card Applet)

This module contains the embedded software running inside the Secure Element of the NXP J3R180 chip. To build and deploy this module, we utilised `ant-javacard` [10] for compilation, `GlobalPlatformPro` [11] for management, and the `Oracle JavaCard SDK (jc320v25.1)` [8].

The implementation is divided into three core components:

- `card.Card`: The main Applet extending `javacard.framework.Applet`. It handles APDU communication, manages persistent EEPROM state (Keys, GOC counters), and orchestrates atomic updates via `JCSystem`.
- `card.MathLib`: A utility library providing arithmetic operations for 32-bit integers. Since the hardware natively supports only 16-bit values, this class performs addition, subtraction, and comparison on `byte` arrays.
- `tool.provisioner.Provisioner`: A setup utility that handles the initial certificate installation and locks the state afterwards.

Challenges and Solutions

We encountered hurdles regarding cryptographic support and integer arithmetic. The standard libraries did not automatically support the required Elliptic Curve parameters, and the 16-bit architecture posed a risk of overflows during high-value transactions.

To address this, we manually injected the bytecode for the SECP256R1 curve parameters using a custom `initCurve` method. For data integrity, we implemented a Sign Bit check in `MathLib` to detect and reject overflows. Additionally, the Applet enforces strict logic rules, such as ensuring Sequence 0 is always a GENESIS transaction and preventing re-initialization by locking the certificate state.

3.4 Module: Client-Core

The `client-core` serves as the central backend for both supported platforms (Desktop and Mobile). Its development began by evaluating the official Ditto Java-Server quickstart [3]. However, we significantly adapted the codebase to meet our specific requirements. As we aimed for a standalone native application rather than a web service, we removed the Spring Boot framework to eliminate server overhead. Instead, we integrated JetBrains Compose Multiplatform [6] to ensure the UI components are reusable across both Android and Desktop JVMs. The architecture is split into a **Java Layer**, containing the business logic and cryptography, and a **Kotlin Layer** used for the GUI.

3.4.1 Smart Card Interaction (`dole.card`)

This package defines how the software interacts with the physical world, centred around the `SmartCard` interface.

Challenges and Solutions

The core challenge in supporting both Android and Desktop environments is the discrepancy between hardware libraries. Android utilises the `android.nfc` API, whereas Desktop PCs rely on `javax.smartcardio`. These libraries are incompatible, preventing a single unified implementation of the hardware driver.

We solved this by defining the platform-agnostic `SmartCard` interface. This forces the specific platform implementations (in the `pc-app` and `android-app` modules) to provide uniform methods. Consequently, the overlying business logic remains completely decoupled from the underlying operating system and hardware drivers.

3.4.2 Cryptography (`dole.crypto`)

This package handles the security primitives and binary formatting.

- `ProtocolSerializer`: Responsible for converting high-level Java objects into raw Byte-Arrays (APDUs) for the card and vice versa.
- `CryptoUtils`: A utility class for SHA-256 hashing, signature verification, and key format conversions.

Challenges and Solutions

A specific technical constraint of smart cards is the limited payload size and transmission speed. Sending large data packets, particularly those containing redundant authentication data, led to performance bottlenecks during the handshake.

To address this, we optimized the transmission protocol in the `ProtocolSerializer`. By analysing the authentication flow, we determined that the PIN does not need to be transmitted with every command if the session is already authenticated. Furthermore, we implemented tight binary packing instead of JSON for card communication, significantly reducing the payload size.

3.4.3 Data Model (`dole.transaction`)

This package defines the immutable data structures used throughout the application, including the `Transaction` sealed interface and its implementations (`MintTransaction`, `BurnTransaction`, `SendTransaction` and `GenesisTransaction`).

Challenges and Solutions

Handling different transaction types with distinct payloads can lead to complex and error-prone code. To solve this, we leveraged Java **Sealed Interfaces** and **Records**. This structure enforces a strict type hierarchy, which allows the compiler to help us ensure that every transaction type is handled correctly and safely.

3.4.4 State Management (`dole.ledger`)

This package forms the heart of the state management on the client device.

- `Ledger`: This class functions as the state manager. It represents the user's local view of the network, aggregated from received logs.
- `LedgerEntry`: The central data object representing a single log entry. It parses raw JSON data into a usable format for the Java application.
- `LedgerService`: An interface defining the contract for saving entries and observing ledger updates.

Challenges and Solutions

The biggest technical challenge was the efficiency of state reconstruction. Since the Ledger is based on Event Sourcing, theoretically, the entire log history would need to be re-read and validated every time the app starts. As the history grows, verifying the cryptographic signatures of every past transaction would become too slow for a good user experience.

To mitigate this, we implemented a **Snapshotting** mechanism (via the `SettingsService`). The validated state (sequences, hashes, and counters) is serialized and persisted locally whenever critical metadata updates or the application terminates. Upon restart, this state is reloaded, allowing the system to trust the persisted state and skip validation for known logs. Only new logs received since the last snapshot require full verification.

3.4.5 Display Logic (`dole.balance`)

This package processes the technical counters into a displayable format for the Kotlin GUI.

- `BalanceCalculator`: The core logic for interpreting transaction histories.
- `TransactionDelta`: A data class representing the effective change in value for specific transactions.
- `BalanceResult`: A record that holds the total balance and the list of changes.

Challenges and Solutions

The SmartCard stores transactions based on a strictly monotonic Global Operation Counter (GOC) rather than individual amounts. This creates a display logic challenge: a naive implementation would display the cumulative counter value rather than the effective transfer amount.

To resolve this, the `BalanceCalculator` iterates chronologically through the sorted list of transactions and calculates the **delta** between the current GOC and the previously stored GOC. Only this calculated value is passed to the UI, ensuring the user sees the actual transaction amount.

3.4.6 Core Integration (`dole.wallet`)

This package connects all other modules to build the actual application.

- `WalletService`: The central controller connecting the SmartCard, Ledger, and Settings. It manages the real-time state.
- `SettingsService`: Handles the persistence of the snapshot data and account information to a local JSON file.
- `StoredAccount`: A data record representing the local user metadata (ID, name, and PIN hash).

Challenges and Solutions

Connecting the components required careful management of the data flow. A key challenge was ensuring that the physical card stays synchronized with the network, while also keeping the application state available even when offline or after a restart.

To solve this, the `WalletService` implements an **Auto-Sync** mechanism. It detects transactions recorded on the Ledger that are not yet on the card and automatically writes them to the hardware. Additionally, the `SettingsService` saves the entire local state (including names and balances) to a JSON file using GSON [4]. This allows the application to restore its context instantly upon restart without re-processing the history.

3.4.7 Visualization (`dole.gui`)

This module implements the graphical user interface using Kotlin and JetBrains Compose Multiplatform [6]. The GUI logic is distributed across ten classes:

- `WalletApp`: The entry point of the UI application. It defines the navigation graph and manages the routing between different screens.
- `WalletViewModel`: The central state holder that connects the user interface with the background logic.
- `HomeScreen`: The landing page displaying the scrollable list of available account cards.
- `SetupScreen`: Manages the registration process, including the initialization of new hardware and the creation of the Genesis block.
- `AuthScreen`: Handles the user login, specifically the secure PIN entry and validation against the smart card.
- `DashboardScreen`: The main view after authentication, showing the current balance, the transaction history, and enabling the execution of new transactions.
- `SettingsScreen`: Provides access to local configuration, allowing users to modify account names, change PINs, or remove stored accounts.
- `WalletCard`: A reusable UI component that visually represents a smart card with animated status indicators.
- `ResizableNumPad`: An adaptive keypad component that resizes dynamically based on the available screen width.
- `LayoutUtils`: A collection of helper functions and shared UI constants to ensure consistent styling.

Challenges and Solutions

A primary hurdle was managing the physical connection to the smart card. If the connection is lost during a critical process like PIN entry, the application must handle this gracefully. We implemented a monitoring system that immediately blocks the screen if the card is removed, preventing undefined states.

A security risk was "Card Swapping", where a user might start a process with one card and switch to another before finishing. We solved this by continuously checking the card's ID. If it changes during a session, the transaction is stopped.

Finally, ensuring the app looks good on both Desktop and Android was a challenge. Instead of writing two separate apps, we implemented a flexible layout system. The app detects the screen size and automatically switches between a portrait layout for phones and a landscape layout for computers using the shared components in `LayoutUtils`.

3.5 Module: Platform Implementations

While the `client-core` contains the shared logic, the `pc-app` and `android-app` modules provide the necessary entry points and hardware drivers. Crucially, these modules integrate the specific Ditto SDK implementations optimized for their respective operating systems to ensure efficient mesh networking.

3.5.1 Desktop Application (`pc-app`)

This module provides the runtime environment for Windows, Linux, and macOS. It implements the `SmartCard` interface and initiates the standard Java version of the Ditto SDK. The application is launched via the `Main` class.

Challenges and Solutions

A critical instability was observed with the `javax.smartcardio` library. If a card reader was unplugged and reconnected during runtime, the library often failed to re-initialize the context, occasionally crashing the entire JVM.

Our solution was to isolate the hardware access into a separate system process running the `SmartCardService` class. The main application communicates with this service via standard input/output streams. If the driver hangs or crashes, the main app simply kills the subprocess and spawns a fresh instance, effectively rebooting the hardware connection without affecting the user interface.

3.5.2 Android Application (`android-app`)

This module targets mobile devices and handles the integration with the Android OS. The application entry point is the `MainActivity`, which manages the lifecycle and system permissions.

Challenges and Solutions

A key challenge was navigating the strict Android permission system. To enable the Ditto mesh network, the app requires explicit runtime permissions for Bluetooth (Scan, Connect, Advertise) and Nearby Wi-Fi Devices.

We implemented a robust permission handler in `MainActivity` to request these rights at startup. Additionally, we found that stable P2P connectivity on Android required using the secure WebSocket protocol (`wss`), unlike the desktop client which defaulted to standard HTTPS.

3.6 Testing Environment

To validate the implementation, we performed functional testing on both platforms.

The desktop application was primarily tested on Microsoft Windows using the ACS ACR1252U-MF NFC Reader III [1].

The mobile implementation was verified on smartphones running Android 13 equipped with integrated NFC hardware.

4 Discussion and Future Work

The fundamental design decision to shift the enforcement of protocol rules from the network to trusted hardware effectively eliminates the double-spending problem in offline scenarios. However, this shift inevitably introduces new systemic dependencies. The security challenge transforms from an algorithmic consensus problem into a problem of hardware management and scalability. Throughout the design and implementation phases, we identified specific limitations stemming from the physical nature of the hardware and the resource-constrained environment of smartcards. This chapter critically analyses these challenges and outlines architectural extensions required to overcome these boundaries in a production-grade system.

4.1 Hardware Loss and Recovery

An inherent characteristic of our bearer-instrument model is the risk of total loss. Since the private key is stored as non-exportable data within the Java Card's Secure Element, the physical loss or malfunction of the smartcard results in the irreversible loss of access to all stored assets.

Standard recovery mechanisms, such as mnemonic seed phrases used in cryptocurrencies like Bitcoin [9], are fundamentally incompatible with our security model. Providing the user with the private key in plaintext would allow them to import it onto manipulated hardware or software emulators. Because the manufacturer's Device Certificate is mathematically bound to the public key (and thus to the exported private key), the network would continue to validate signatures from the manipulated device as trustworthy. This would allow a malicious user to circumvent hardware-enforced invariants, such as solvency checks and sequential ordering, effectively breaking the protocol's security guarantees.

Proposed Solution: Two-Layer Key Wrapping

To enable recovery without compromising the security model, the private key must be exportable without ever being exposed in plaintext, neither to the user nor to the manufacturer. We propose a two-layer encryption scheme established during the genesis process:

1. **Inner Layer (User-Bound):** The card encrypts the generated private key using a strong user secret (e.g., a high-entropy passphrase or PIN).
2. **Outer Layer (Issuer-Bound):** The resulting ciphertext is subsequently encrypted with the Issuer's public key. This encrypted data package, termed the Recovery Blob, is publicly stored within the Genesis Log.

The Recovery Process

In the event of hardware loss, the user authenticates themselves with the manufacturer. The manufacturer uses their private key to decrypt the Outer Layer and injects Inner Layer, which is still encrypted, directly into the Secure Element of a replacement card. Only upon entry of the user's PIN does the new card internally decrypt the private key.

Implementation Limitations

While theoretically sound, this approach introduces a dependency on the availability of the central authority. Furthermore, the secure injection of keys into the protected memory area necessitates low-level access to the card operating system or proprietary extensions of the Java Card API. Implementing such mechanisms exceeded the scope of this prototype. Consequently, the realization of this recovery feature has been classified as future work.

4.2 Hardware Constraints and Scalability

A central limitation of the current architecture stems from the inherent physical restrictions of the smartcard hardware. To deterministically prevent replay attacks (the replay of previously processed logs) the card must persistently store the current GOC state (specifically the latest *totalReceived*

value) for each individual interaction partner. Given that EEPROM storage on standard Java Cards is severely limited [5], the internal lookup table cannot grow indefinitely. In our reference implementation, the card is restricted to a fixed capacity of direct contacts (200 peers). High-volume actors with frequent interactions with unique partners, such as merchants or currency exchanges, would reach this limit rapidly. Any attempt to interact with a 201^{st} participant would be rejected at the hardware level in the current design, triggering an error. This has critical implications for system usability: A user would be unable to settle valid incoming logs from new senders, as the card lacks a free slot in its internal registry. Although the funds would theoretically exist in the off-card ledger, they could not be claimed or spent via the hardware trust anchor, equating to an effective loss of liquidity.

Proposed Solution: Stateless Hardware via Merkle Proofs

To decouple system scalability from the constraints of EEPROM storage, we propose an architectural shift towards Stateless Hardware. In this model, the card no longer retains the complete list of peer states but instead stores a single 32-byte Merkle Root Hash. The entire contact database is offloaded to the smartphone application. The root hash stored on the secure element acts as a cryptographic fingerprint of this externalized state. The protocol operates as follows:

1. **Outsourcing:** The smartphone manages the GOC states of potentially thousands of peers within a Merkle Tree structure. The individual contact states form the leaves of this tree. The card internally stores only the hash of the tree's root.
2. **Proof over Storage:** When the app needs to update a specific account balance (e.g., Alice's), it transmits the current leaf value to the card, together with the corresponding Merkle Proof.
3. **Verification & Update:** Using the provided path, the card verifies whether the submitted value corresponds to its stored root hash. Upon successful verification, the card accepts the state, executes the booking logic, recalculates the path with the updated value, and saves the resulting new root hash.

This mechanism renders the card's effective storage capacity virtually infinite, as validation depends solely on the correctness of the proof, regardless of how many other contacts exist in the tree. A trade-off is computational latency: verification requires the card to perform hash operations, which is slower than direct memory access. However, since SHA-256 is hardware-accelerated on modern Java Cards, this remains the most scalable solution for mass adoption. For the present prototype, the array-based solution was chosen due to the high implementation complexity of this approach.

4.3 Provisioning Model

While the Issuer-Centric Provisioning Model described in Section 2.4 ensures the functional security of the prototype, it introduces significant centralization into an otherwise decentralized system. Since the manufacturer is responsible not only for hardware fabrication but also for code flashing and the generation of cryptographic identities, they become a Single Point of Failure within the trust model.

Centralization Risks

This model requires that users blindly trust the manufacturer. A compromised or malicious issuer could substitute the public open-source protocol with a manipulated applet or insert hidden backdoors into the code before distribution.

Critically, the Java Card architecture explicitly prevents the read-back of installed bytecode to protect intellectual property. Consequently, the end-user possesses no mechanism to non-destructively verify the integrity of the software running on their card. As a result, an adversary does not need to break the protocol's strong cryptography. Instead, they can bypass security by attacking the manufacturer directly.

Technical Hurdles for Self-Setup

An alternative model, in which the user purchases a blank card and installs the applet at home, fails in the current ecosystem due to two fundamental technical hurdles:

1. **Hardware Attestation & NDA Barriers:** To prove that the code is running on genuine Secure Element hardware, access to the chip manufacturer's root certificates is required. However, these are subject to strict Non-Disclosure Agreements and are inaccessible to end-users or open-source projects. Without this certificate, the network cannot distinguish between a secure chip and an insecure PC simulator.
2. **The Lying Applet Problem:** Even if the hardware were verified, a mechanism for the remote attestation of the code is missing. A user, who holds administrator privileges during self-setup, could install a manipulated applet that mimics the original protocol externally but violates rules internally. Standard smartcard operating systems such as JCOP currently do not offer a tamper-proof interface to report the hash of the running application to the network.

Due to these limitations, the Issuer-Centric model remains the only viable solution to guarantee the consistency of the GOC-Ledger for this prototype, despite the associated centralization risks.

Proposed Solution: Remote Attestation

Theoretically, the provisioning dilemma can be resolved via an extended attestation mechanism that facilitates secure remote setup by the end-user. This approach requires a smartcard for which the manufacturer publicly documents the Root CA key and whose operating system exposes an open API for code signing.

The underlying security model relies on the trustworthiness of the operating system, which resides as an immutable entity within the card's ROM. A secure registration workflow would be designed as follows:

1. **Request:** The applet installed by the user generates a key pair and requests the operating system to certify it.
2. **OS Signature (Binding):** The operating system does not blindly sign the data provided by the applet. Instead, it builds a payload containing the applet's public key and the hash of the installed bytecode ($\text{Hash}_{\text{Applet}}$). Since the OS manages the memory, this hash cannot be forged by the applet.
$$\text{Sig}_{\text{Attestation}} = \text{Sign}_{\text{HardwareKey}}(\text{PubKey} + \text{Hash}_{\text{Applet}})$$
3. **Verification:** Other network participants validate this certificate. A valid signature proves the authenticity of the hardware. If the hash contained in the certificate additionally matches the known hash of the official open-source code, software integrity is ensured. The certificate is accepted only if both conditions are met.

Practical Obstacles

The practical implementation of this model on regular Java Cards is hindered by the lack of available interfaces. Standard JCOP cards often do not expose an Applet Self-Attestation API for third-party applications.

Implementing such custom attestation would require deep knowledge of proprietary GlobalPlatform extensions [11] and access to documentation protected by non-disclosure agreements, which exceeds the scope of this project. Consequently, secure user-provisioning remains an unsolved issue, which is mitigated in this prototype by the aforementioned Provisioning Model.

5 Conclusion

With DOLE, we have developed a functional protocol for digital peer-to-peer payments that eliminates the fundamental dependency of conventional systems on permanent internet connectivity. The project demonstrates that the security of financial transactions can be guaranteed without a global consensus algorithm (such as Proof-of-Work) by combining mathematical convergence with physical security.

By integrating the GOC-Ledger model [7] with the Java Card as a hardware trust anchor, we successfully secured the theoretical concept of Eventual Consistency against practical attacks such as double-spending and negative balances. Furthermore, architectural optimizations, specifically the "No-ACK" model and the static binary protocol, prove that this high level of security can be implemented efficiently on resource-constrained hardware, harmonizing effectively with modern mesh technologies like Ditto [2]. At the same time, this work highlights that our approach introduces new dependencies. The implemented Issuer-Centric model solves the problem of hardware integrity but introduces a central Single Point of Failure, which stands in tension with the decentralized ethos of the original concept.

For widespread mass adoption, the future of this architecture lies in direct integration into end devices. Porting the GOC applet to the Embedded Secure Element of modern smartphones would render physical cards obsolete and revolutionize the user experience by eliminating the need for external NFC interactions. While proprietary restrictions currently hinder developer access to these components, initiatives for open-source hardware like OpenTitan [13] offer a crucial perspective: They could anchor the security architecture of DOLE directly in the smartphone's silicon, granting developers the necessary access. In summary, DOLE validates the hypothesis that Trusted Hardware combined with the GOC-Ledger represents a viable and resilient alternative to blockchain-based systems when availability in isolated environments is prioritized.

We would like to express our gratitude to Dr. Erick Lavoie for his foundational research on the GOC-Ledger, which provided the theoretical backbone for this project.

Special thanks go to Prof. Christian Tschudin, who not only brought this research to our attention but also provided the crucial guidance to synergize our hardware-based approach with the GOC model. We deeply appreciate the time he dedicated to discussing our architectural drafts and his valuable feedback throughout the development of DOLE.

References

[1] Advanced Card Systems Ltd. *ACR1252U USB NFC Reader III*. Accessed: 2026-02-08. URL: https://www.acs.com.hk/en/products/342/acr1252u-usb-nfc-reader-iii-nfc-forum-certified-reader/.

[2] Ditto Live Inc. *Ditto Live: Distributed Data Platform*. URL: https://ditto.com/ (visited on 02/08/2026).

[3] Ditto Live Inc. *Java Server Quickstart*. Ditto SDK v5 Documentation. 2025. URL: https://docs.ditto.live/sdk/v5-java-server/quickstarts/java-server (visited on 02/08/2026).

[4] Google LLC. *Gson: A Java serialization/deserialization library to convert Java Objects into JSON and back*. JSON Processing Library. URL: https://github.com/google/gson (visited on 02/08/2026).

[5] *JCOP 4: Java Card OS for SmartMX3 Secure Microcontrollers*. Product Brochure, Rev. 0. NXP Semiconductors. July 2018. URL: https://www.nxp.com/docs/en/brochure/JCOP4SECIDAPPA4.pdf.

[6] JetBrains. *Compose Multiplatform*. Declarative UI Framework for Kotlin Multiplatform. 2026. URL: https://github.com/JetBrains/compose-multiplatform (visited on 02/08/2026).

[7] Erick Lavoie. *GOC-Ledger: State-based Conflict-Free Replicated Ledger from Grow-Only Counters*. 2023. arXiv: 2305.16976 [cs.DC]. URL: https://arxiv.org/abs/2305.16976 (visited on 02/08/2026).

[8] Oracle Corporation. *Java Card Development Kit*. Software Development Kit. URL: https://github.com/martinpaljak/oracle_javacard_sdks (visited on 02/08/2026).

[9] Marek Palatinus, Pavol Rusnak, and Aaron Voisine. *BIP-39: Mnemonic code for generating deterministic keys*. 2013. URL: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki (visited on 02/08/2026).

[10] Martin Paljak. *ant-javacard: JavaCard SDK integration for Ant*. Build Tool. URL: https://github.com/martinpaljak/ant-javacard (visited on 02/08/2026).

[11] Martin Paljak. *GlobalPlatformPro*. Card Management Tool. URL: https://github.com/martinpaljak/GlobalPlatformPro (visited on 02/08/2026).

[12] *Specifications for the Java Card 3 Platform, Version 3.0.5, Classic Edition*. Runtime Environment Specification. Oracle. Nov. 2017.

[13] The OpenTitan Coalition. *OpenTitan: Open Source Silicon Root of Trust*. URL: https://opentitan.org/ (visited on 02/08/2026).