A Framework for Resilient, Transparent, High-throughput, Privacy-Enabled Central Bank Digital Currencies

E. Androulaki, M. Brandenburger, A. De Caro, K. Elkhiyaoui, L. Funaro,
A. Filios, Y. Manevich, S. Natarajan, M. Sethi
IBM Research

ABSTRACT

Central Bank Digital Currencies refer to the digitization of life-cycle's of central bank money in a way that meets first of a kind requirements for transparency in transaction processing, interoperability with legacy or new world, and resilience that goes beyond the traditional crash fault tolerant model. This comes in addition to legacy system requirements for privacy and regulation compliance, that may differ from central bank to central bank.

This paper introduces a novel framework for Central Bank Digital Currency settlement that outputs a system of record—acting a a trusted source of truth serving interoperation, and dispute resolution/fraud detection needs—, and brings together resilience in the event of parts of the system being compromised, with throughput comparable to crash-fault tolerant systems. Our system further exhibits agnosticity of the exact cryptographic protocol adopted for meeting privacy, compliance and transparency objectives, while ensuring compatibility with the existing protocols in the literature. For the latter, performance is architecturally guaranteed to scale horizontally. We evaluated our system's performance using an enhanced version of Hyperledger Fabric, showing how a throughput of >100K TPS can be supported even with computation-heavy privacy-preserving protocols are in place.

1 INTRODUCTION

Central Bank Digital Currency (CBDC for short) is a *central-bank liability* on par with physical cash and bank reserves. It is designed to serve the functionalities of medium of exchange, store of value and a unit of account, and intended for both *wholesale* and *retail* payment transactions.

In recent years, CBDC has been positioned as a viable approach to address the current inefficiencies in financial markets and payment systems. In particular, wholesale CBDC is anticipated to significantly reduce settlement delays and drastically decrease counterparty risk in financial markets, as well as, speeding up cross-border payments. Retail CBDC is expected to reduce transaction fees, and break the existing monopoly of today's payment service providers, enhance financial inclusion, and trigger innovation in the digital payments space.

In light of these advantages, more than 130 central banks are actively exploring CBDC and publishing periodic reports on the functional and non-functional requirements of the CBDC infrastructure, the evolving architectural considerations [1, 2], and the outcomes of their various CBDC experimentations. Indicatively, a handful Central Banks have even launched CBDC pilots [2], while the European Central Bank has just launched a legislation proposal for the adoption of a *digital euro* [3].

Although CBDC systems will be regulated differently depending on the jurisdiction, they agree on certain aspects.

First of all, the critical impact of CBDC infrastructure on the money supply establishes that its governance will be in the hands of the central bank only (i.e., centralized). This is because Central Banks and only Central Banks are tasked with the "*formulation of a country's monetary policy to achieve and maintain economic and price stability*"[4].

Despite the centralised nature of a CBDC system's governance, a CBDC system is required to operate in a highly *robust and resilient* manner, i.e., functioning properly even if parts of the system crash, or are compromised (e.g., after successful cyber- or insider-attacks). This is also reasonable given the critical nature of a CBDC system, and imposes a distributed if not a decentralized deployment of the system.

In addition, there is the need for *interoperation* and a preference towards *programmability*. In terms of interoperation, a CBDC system should interoperate with existing payment and settlement infrastructure, other CBDC systems, and the emerging digital asset systems. In terms of programmability, the system should offer flexibility for new capabilities to be developed as per the dynamically evolving needs for the financial ecosystem. Programmability is a requirement associated to a common expectation of CBDC systems to "*foster innovation in payments*".

Regulatory compliance bring to the fore requirements for *efficient dispute resolution*, *fraud detection* and *auditability*. For example, AML[1] and CFT[2] regulations stipulate that suspicious payment transactions be detected, attributed to their origin and reported to the relevant authorities, while PSD2[3] [5] emphasizes the importance of fraud detection and dispute resolution.

Fraud detection and dispute resolution further require *accountability*. Accountability ensures that the various parties cannot repudiate their actions in the system, and is important for CBDC consumers (cannot repudiate their payments) as well as CBDC system entities (cannot repudiate their decisions). Dispute resolution and interoperation requirements typically impose design principles of *transparency* in decision making process, and transaction processing. Transparency is optimally served by a *ledger of (processed) transactions* that acts as a single point of reference, and truth (trusted).

As important is the *privacy of payment transactions* in both retail and wholesale settings. Privacy refers to the right of data owners to control who accesses their transactional information. For example, PSD2 states that the processing of personal information must comply with the GDPR and its principles of data minimization, which restricts the collection of personal information to what is necessary

[1] Anti Money Laundering
[2] Combating the Financing of Terrorism
[3] Amendment proposed by the European Union to the Payment Service Providers Directive

for transaction processing. This can be interpreted in various ways: a conservative approach to data minimization will ensure that payment transactions are processed without leaking any information about the transacting parties or the values of the transactions. This, however, renders transaction monitoring and audit more difficult. A permissive approach will, on the other hand, reveal the value of the payments, and potentially, the identities of the payer and payee. A forward-looking CBDC system should be able to accommodate different interpretations of privacy, in balance with all the other requirements, particularly, performance and auditability. As the technology evolves, so will privacy regulations and requirements, and agility, should be built in.

Finally, retail CBDC systems should be able to compete with existing payment solutions, and, as such, must accommodate transactions of millions of users. This translates into being able to process tens of thousands of transactions per second (TPS) at peak time.

Prominent efforts to address the CBDC requirements [6, 7] focused on achieving high throughput in an architecture where both governance and transaction processing are *centralized*, resulting in systems that are vulnerable to single points of compromise. In other words, these solutions assume that the entities processing transactions will never arbitrarily misbehave, i.e., they are crash tolerant but not fault tolerant. We emphasize that given the critical nature of a CBDC system, and the resulting geographic distribution of its transaction processing components, ensuring correct operation of the system in the event of compromise becomes relevant. Notice that very frequently data centres located in different areas come with different trust domains.

To circumvent single points of control, financial institutions [8] and the research community (e.g., [9]) have been exploring decentralised consensus-based transaction processing systems such as distributed ledger technology (DLT). The relevance of DLTs is also in alignment to requirements for a trusted system of record, and programmability. Unfortunately, current DLT implementations penalize throughput. While work-arounds leveraging off-chain exchanges that are only occasionally anchored on-chain have become popular in DLT systems (a.k.a *layer-2 solutions*), have been dismissed in the context of CBDC, in the name of *fast* finality and full transparency for all payment transactions.

In this paper, we introduce a transaction processing framework for (fungible) financial asset management, and most prominently for CBDCs, that addresses the aforementioned challenges. We show that permissioned DLTs are advantageous with respect to transparency and resilience to compromised nodes—even with a centralized governance model—and can meet the CBDC performance and scalability requirements without significant performance discrepancies from distributed centralised systems that offer a system of record. In particular, we propose a system architecture and protocols within, exhibiting:

- *Strongly accountable and Transparent transaction processing* by employing strong identity management and a shared ledger that records all transactions that are processed by the system; we refer to that shared ledger as *system ledger*.
- *Resilience to compromised nodes* by employing byzantine fault tolerance in all phases of transaction processing and the system ledger evolution, in the eyes of improved Hyperledger Fabric DLT platform.
- *Pluggable transaction format and transaction security checks*, decoupling application-related logic from the actual DLT layer, where the system ledger is built and maintained. This derives from the DLT support for smart contracts, that further ensures the easy integration of different privacy mechanisms and functionalities into the system or extensions of its to support more use cases around regulated digital assets (e.g., bonds and securities and tokenized deposits).
- *Throughput and latency* in transaction processing that is *equivalent to the throughput and latency observed in today's (ledger-enabled) centralized solutions* (e.g., [7]); this is accomplished by employing (i) the *execute-order-validate* transaction processing model introduced by Hyperledger Fabric 1.0 [10], (ii) a state of the art *byzantine fault tolerant consensus* protocol in terms of performance and scalability for *order* (similar to [11]), and (iii) Two-phase commit transaction processing principles for *commit*.
- *Horizontal scalability of all application-layer logic* introduced in transaction processing. This is particularly important for applications that employ computationally heavy Zero Knowledge Proofs to offer privacy.

We provide a prototype implementation of our framework as an *evolved version* of Hyperledger Fabric [10], coupled with the four CBDC privacy models: (i) standard UTXO support using standard PKI with no privacy in place; (ii) standard UTXO support with accountable pseudonymity/anonymity for transactors, (iii) UTXO support enhanced with anonymity and exchanged amount confidentiality, and (ii) untraceable UTXO utilizing the cryptographic means in [12] for full transactor (accountable) privacy. We further evaluated our system's performance using three consensus protocols: i) a crash fault tolerant consensus protocol, i.e., Raft [13], ii) a byzantine fault tolerant consensus protocol in the wild, i.e., BFTSmart [14] and iii) a new byzantine-fault tolerant architecture inspired by [11], exhibiting state of the art performance and scalability.

Our results show that for the standard UTXO pseudonymity model, our prototype implementation can process up to 80, 000 TPS in the case of Raft and BFTSmart and more than 150, 000 TPS in the case of emerging consensus algorithms. Our results further demonstrate the horizontal scalability of transaction processing compute. In fact, we show that the same numbers *can be accomplished* in stronger privacy scenarios where the exchanged amounts are concealed, and / or activity of individual users concealed at the cost of more powerful equipment. The obtained performance numbers correspond to a CBDC system that offers privacy for end-users while allowing authorized auditors to inspect transactional information, and settlement components to properly process transactions.

Layout. The paper is organized as follows. Section 2 provides an overview of the system entities, the system's trust assumptions, and requirements. In Section 3, we provide an overview of Hyperledger Fabric and the proposed enhancements to support the demands of the CBDC use-case, while Section 4 presents various token architectures to be considered for privacy in a CBDC system. Section 5

describes our framework by combining Hyperledger Fabric and the token architectures proposed in Section 4. Section 6 benchmarks our system's performance. Section **??** we provide an overview of related work, and, finally, in Section 7 we conclude the paper.

2 SYSTEM OVERVIEW

At a very basic level, a CBDC system involves a **central bank**, which decides the monetary policy, manages the overall liquidity in circulation, and validates transactions via a **settlement engine**. There are also **users**, who hold CBDC and exchange it for goods and services. In an intermediated CBDC, one also finds *intermediaries*, corresponding usually to the commercial banks, which are tasked with performing KYC[4] checks and managing user accounts, both deposits and CBDC.

A user with a CBDC account, can request CBDCs from the Central bank[5] via a *withdrawal request* that would typically result into *issuance* of new CBDCs by the Central Bank. A user can make payments that will be subjected to AML checks and CFT measures just like digital payments today. Therefore, for accountability and non-repudiation purposes, each user must be equipped with a *long-term identity* that binds the physical identity of the user to a public key, and which will be used subsequently for authentication and transaction authorization. This process can be facilitated by a **registration authority** that *onboards* the users by assigning them a *unique enrollment identifier*, and keeps track of the mapping between the physical identity of the user, their unique enrollment identifier, and their public key, plus any other additional attribute that might be needed. Given that CBDC users will hold secret keys, they will be provided a *digital wallet* that will help them track their CBDC holdings and authorize transactions.

If privacy against the central bank is required, then it is important to restrict its access to payment data. In particular, the central bank with the settlement engine should validate CBDC payments without learning their value or the identities of the parties involved. However, this raises the challenge of transaction monitoring necessary to enforce AML and CFT regulations. This can be addressed by introducing **auditors**, which are entities independent of the central bank, that are authorized to inspect payment information and determine if a transaction is compliant or not based on historic data. An example of such auditors are the intermediaries.

In what follows, we describe in detail the system participants, their interactions and the associated trust assumptions. For simplicity, we consider non-intermediated CBDC, where users directly submit withdrawal requests or payments to the central bank without going through the commercial banks.

2.1 System Participants

The participants of a typical CBDC system are depicted in Figure 1:

- *Users* hold CBDCs and initiate withdrawal requests and payments. A user is assigned a unique enrollment identifier eid and a credential cred, that certifies the mapping between the user's enrollment identifier and their public key. The pair (eid, cred) corresponds to the long-term identity of the user. Each user maintains a wallet to manage the secret keys, track the holdings, and authorize the transactions.
- The *Registration Authority* assigns a unique enrollment identifier to each user, and produce the user's identity credential. More specifically, during a registration request of a user u, the authority assigns to u a unique enrollment identifier, eid, and verifies that u knows the secret key underlying the provided public key, say pk. After a successfully registration, the registration authority provides cred to u, and keeps the mapping between the physical identity of the user and their long term identity in the CBDC system. The Registration Authority can be operated by the Central Bank and facilitated by the Commercial Banks.
- The *Central Bank* issues CBDCs by introducing them into the system, and is responsible for the settlement of CBDC payment, issuance and redemption transactions. CBDCs are owned by users, represented by their unique enrollment identifier. The central bank is also responsible of redeeming CBDCs, when those need to be removed from circulation.
- The *Settlement Engine* is responsible for settling CBDC payments, as well as issuance and redemption requests under the purview of the central bank. In our solution the settlement engine comprises a *distributed/decentralized* network governed by the central bank. It validates the system transactions against the system's rules and a global state, maintained in a *ledger*. If a transaction is valid, the settlement engine updates the global state (i.e., ledger) accordingly. As we discuss in Sections 3 and 5, our system's settlement engine is based on Hyperledger Fabric[10] permissioned DLT.
- *Auditors* monitor user transactions for compliance purposes. Notably, they access the content of the transactions and determine whether they are legitimate or suspicious. An Auditor

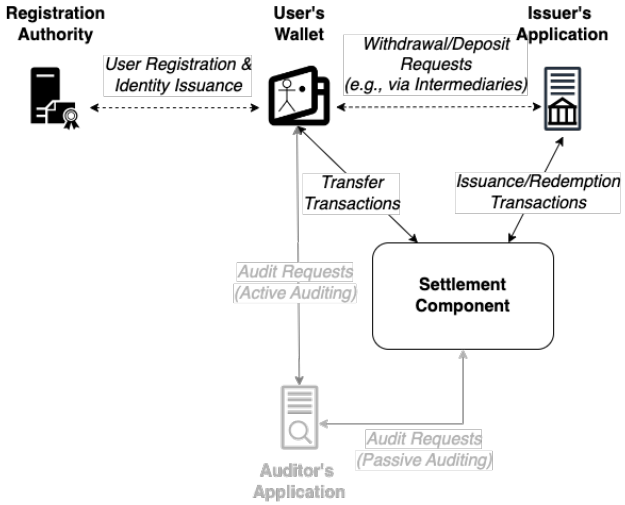

Figure 1: The participants of a typical retail CBDC system include end-users (payers, and payees), a registration authority, granting users the right to make use of the system, a central bank accommodating liquidity requests for users (typically via intermediaries), and settles payments, and auditors who are responsible for monitoring user or CBDC activity.

[4]Know Your Customer.
[5]The request is submitted to the intermediary, if available.

is said to be *active* if their sign-off is required by the settlement engine to process the transaction itself, and *passive* if they are only expected to audit transactions asynchronously, after the settlement engine has processed them.

2.2 Trust Assumptions

The central bank is trusted with issuance and redemption of CBDCs. That is, an issuance (resp. a redemption) is only triggered upon a user request and in accordance to the central bank's monetary policies. [6] Notice that to add resilience in the decision making process, the central bank can leverage threshold cryptography to distribute the responsibility of these operations among different divisions or authorities in the central bank organisation. For example, the issuer's *secret key* can be split in multiple pieces and distributed to independent entities within the central bank who must collaborate to generate a valid signature. The system continues without disruptions even if a subset of these entities get compromised.

Additionally, we trust the registration authority to assign to each user exactly one enrollment identifier and produce one credential per user. Similar to the central bank, resilience to the registration authority can be added by distributing the trust, e.g., via threshold issuance of credentials.

The settlement engine is a distributed component that is trusted as a whole. That is, the settlement engine will operate correctly, in the presence of a dishonest minority. We consider two failure models: *Crash* and *Byzantine*. The former implies that settlement engine participants behave correctly but they may crash and not respond. The latter entails that a subset of the settlement engine participants can behave *arbitrarily*. In this paper, we evaluate both models and discuss their implications.

Last but not least, we have the users. Users are not trusted. They may attempt to increase their holdings unlawfully, or spend CBDCs they do not own.

When *user privacy* is a concern, the settlement engine only accesses the information necessary for correct transaction processing. The central bank is only privy to issuance and redemption information, whereas the auditors are trusted not to share the information they learn with third parties, unless required by the law. However all entities incl. users are expected to collude to escalate their data access rights, and compromise the privacy of individuals against the colluding entities' access rights.

Finally, auditors are trusted to deliver to regulators the audited information upon request correctly.

2.3 Requirements

In this section we elaborate on a retail CBDC system's requirements. Notice that, excluding performance, all the other requirements are present in wholesale CBDC systems, as well as to other financial asset transfer use-cases.

Security A CBDC system should meet the following security requirements:

- *Transaction Authorization.* A CBDC transaction should only be accepted by the settlement engine if it has been initiated by the lawful parties: the owner of the funds that are being transferred in case of payment transactions, and the authorised issuer(s) in case of CBDC issuance transactions.
- *Balance Preservation.* A Transfer transaction should preserve the total amount of CBDC in circulation. Namely, a transfer increases the holding of the recipient and decreases the holding of the sender by the same amount. An Issue transaction should increase the total amount of CBDC in circulation by the issued value; similarly, a Redeem transaction should reduce the CBDC circulation by the redeemed value.
- *Resilience.* In a distributed system, resilience refers to the system's ability to continue functioning and providing services in the face of various types of failures, errors, and unexpected events. It often requires redundancy, replication, backup and recovery strategies, load balancing, and automated failover mechanisms. It is crucial that critical infrastructures are distributed to reduce the trust cost and enhance resilience.

Privacy. Privacy requirements of CBDCs may vary from country to country; important properties appearing on this front encompass:

- *Transaction anonymity.* A CBDC transaction is anonymous if the origin of the transaction is hidden from the settlement engine and anyone accessing its ledger. We note that since only the central bank is authorized to issue CBDC, Issue transactions do not need to be anonymized.
- *Transaction Confidentiality.* This refers to the confidentiality of the content of the transactions (e.g., identities of involved parties, exchanged currency/value) against all participants except the transacting parties.
- *Transaction Unlinkability.* Transaction unlinkability ensures that the settlement engine and anyone with access to its ledger cannot tell if two CBDC transactions involve the same user, either as an issue beneficiary, a payer or a payee. Note that a transaction can be anonymous without being confidential and vice versa. For example, an anonymous transaction can reveal the value of the transaction, and a confidential transaction can leak the identity of the transaction origin. An unlinkable transaction is both anonymous and confidential.

Auditability/Regulation Compliance. Authorized auditors should be able to access the transactional information of the users falling under their jurisdiction. This access is governed by AML/CFT regulations that determine the information auditors should monitor to successfully flag suspicious transactions. For the same reason and in the presence of intermediaries, they too might be required to sign the transaction as a proof of acknowledgment.

In the presence of a single auditor, the latter inspects the content of all transactions. In multi-auditor settings, audit policies define which transactions any given auditor can access. An example of such a policy is to have the user, at time of registration, assigned an attribute "auditor" that identifies the auditor under which oversight they fall. Consequently, transactions involving that user should be accessible to the auditor identified in the "auditor" attribute. In another scenario, auditor assignment may be associated to an attribute already incorporated into the user's identity, e.g., country of residence.

[6] If intermediaries are available, then an issuance (resp. a redemption) is only triggered when a user exchanges commercial bank money for CBDC and vice versa

Audit-support may assume that the auditor is highly available, and can be reached over the course of all transactions that fall under their responsibility (active auditing). Additionally, audit-support may assume an offline auditor who is expected to asynchronously access transactional data by being exposed to the settlement engine transactions (passive auditing).

Performance. To enable retail payments, CBDC comes with stringent performance requirements. Notably, the transaction throughput and the transaction latency of any CBDC system should be on par existing payment solutions. Per the various retail ECB reports [15], a viable CBDC solution is expected to accommodate around a few tens of thousands transactions per second (TPS) in the first years of operation, gradually leading to up to 150, 000 TPS over the years. Latency is expected to vary between 3 and 5 seconds, to not impact the user experience at a point of sale.

2.4 Interactions Overview

Figure 1 depicts the various components of a CBDC system and their interactions.

First a user must initiate an *on-boarding* process with the registration authority. The authority assigns the user a *unique enrollment identifier*, and keeps track of the mapping between the physical identity of the user, their unique enrollment identifier (henceforth denoted by eid), and their public key (henceforth denoted by pk), plus any other additional attribute that might be needed. This would signify the initialisation of the user's *digital wallet*, that helps the user to track their CBDC holdings and authorise transactions.

A user can receive CBDCs via a withdrawal or a payment operation. In the first case, the user submits a withdrawal request to the CBDC Issuer (typically via their intermediary), who evaluates the request, approves it, and notifies the settlement engine to finalize the issuance of CBDCs. Once the issuance is completed, the user's wallet is updated with the freshly issued currency. In the second case, the user is the payee in a payment. To perform a payment, the payer uses their digital wallet to prepare the corresponding transaction by specifying the amount of CBDC to transfer and the intended payee. Once ready, the wallet submits the transaction to the settlement engine to get it settled. Upon finalisation, the holdings/wallets of the payer and the payee will get updated accordingly.

A user can redeem CBDCs by sending a deposit request to the Issuer, triggering a redemption process on the user provided funds.

If active auditing is in place the user's digital wallet requires an approval from auditor before submitting the transaction to the settlement engine. If passive auditing is in place, the auditor does not need to actively participate in the payment process but can analyze transactions using their secret keys and access to the settlement engine's ledger.

3 EXTENDING HYPERLEDGER FABRIC

Hyperledger Fabric (HLF) is a permissioned DLT system with strong identity management for system participants, and built-in mechanisms to enforce non-repudiation, and with it, accountability. HLF [10] came with the premise to overcome the inherent performance and scalability limitations of DLTs, by introducing a DLT architecture that departs from the traditional *order-execute* DLT

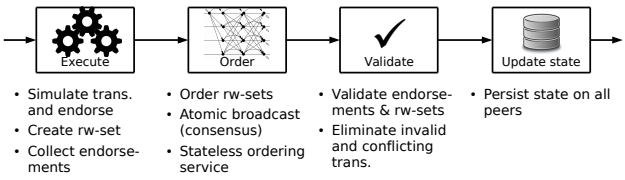

Figure 2: Execute-Order-Validate Model leveraged in Hyperledger Fabric.

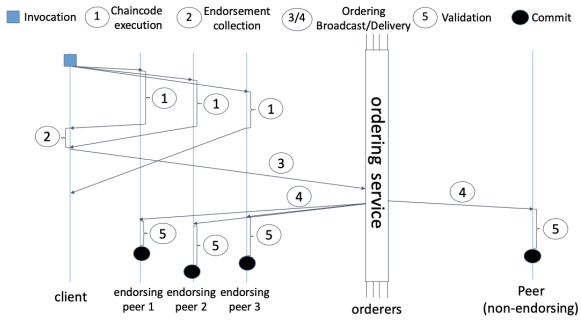

Figure 3: Transaction Lifecycle in Hyperledger Fabric.

paradigm. In particular, HLF introduced an *execute-order-validate* model for transaction processing.

We recall that in *order-execute* transactions are first ordered and batched into blocks, and then executed in the order in which they appear. To reach a consistent state across all DLT participants, the transactions must be executed sequentially and the execution logic must be deterministic. This evidently limits the performance and the flexibility (in terms of either the code that can be executed or the languages that can be supported) of the system. In contrast, the *execute-order-validate* allows transactions to be first *speculatively* executed before being ordered. A transaction that reads stale data during its speculative execution is rejected and skipped during the validate phase, which consists of simple and deterministic checks. An overview of the *execute-order-validate* model is reflected in Figure 2.

The separation between the execute and validate has two advantages: (1) developers can write the code of the execute phase in their language of choice (more flexibility), and (2) the execute phase can run in parallel (higher throughput). Figure 3 shows the transaction flow in HLF. We, therein, find **chaincodes**, which are the smart contracts invoked during the *execute phase*, and which are written in traditional programming languages. For each chaincode, there are **endorsers**, which are a subset of network participants tasked with executing the chaincode and sending the result of the execution. Moreover, each chaincode is assigned a *namespace*, which identifies the partition in the shared ledger that the chaincode is allowed to update. Each namespace comes with an *endorsement policy* that specifies which endorsers should come together and execute the chaincode for the resulting transaction to be accepted during the *validate* phase.

A **client** triggers the execution of a chaincode by sending a transaction proposal to the endorsers of the chaincode, and awaits the response. The endorsers then execute the chaincode speculatively and output a *read-set* which describes the dependencies of the execution (i.e., the entries in the ledger that have been read during the execution), and a *write-set* which are the changes to be applied to the ledger state should the transaction be committed. Together the read-set and the write-set form what we call the *read-write set*. The endorsers then sign and send the results of the chaincode's execution to the client, within a proposal response. Finally, the client attaches the signatures of the endorsers to the Fabric transaction and submits the result to the orderers.

The **orderers** are authorized network participants responsible for the *order phase*, during which they agree on the order in which transactions will be processed in the *validate phase*, batch them into a series of blocks, and deliver the blocks to the committers.

The **committers** are the subset of network nodes responsible for the *validate* phase and applying updates to the ledger. They therefore check each transaction against the ledger state and the endorsement policy of its corresponding chaincode. More specifically, each committer verifies if the entries identified in the read-set correspond to the current state, and if the signatures in the transaction satisfy the endorsement policy of the namespace being updated. If so, then the committer updates its local copy of the ledger in accordance with the transaction's write-set. In HLF, endorsers are also committers, and they are both alternatively known as **Fabric peers**.

HLF provides a default implementation of the endorsers, that handles the chaincode lifecycle, its execution, and the generation of the endorsement signatures. However, when business application logic demands it, one need not follow this default implementation to produce a transaction that will be accepted by the committers. In fact, the committers do not care how a valid transaction has been produced, whether by the default HLF endorser or a custom implementation.

For a pictorial summary of the lifecycle of a standard HLF transaction, refer to Figure 2.

3.1 HLF Relevance and Limitations for CBDC

HLF architecture offers a relevant foundation for CBDC settlement.

- *A chaincode can encode the CBDC issuance/transfer security/validity checks (execute phase).* Chaincode execution is horizontally scalable—allowing more nodes to be added independently to load balance client requests—and can avoid single control points with the appropriate choice of endorsement policy—e.g., requiring sign-offs from several parties. In alignment with the continuously evolving nature of CBDC ecosystems, a chaincode can be upgraded (e.g., if different privacy rules need to be honored, or additional security checks need to be performed) in a transparent manner upon a certain (multi-or single-party) chaincode administration policy is satisfied.
- *Transaction ordering can compile a system-wide ledger, needed for dispute resolution.* Transaction ordering is served by the ordering service in HLF, that, given its modular architecture can easily adjust to the system's threat model and scalability requirements. For example, one can easily change between protocols that tolerate node crashes (Crash Fault Tolerant or CFT) and consensus protocols tolerating a number of compromised nodes (Byzantine Fault Tolerant of BFT) without changing other parts of the system.
- *All layers of transaction processing can tolerate byzantine behavior by an upper bound of nodes.* This is achieved by the appropriate choice of chaincode administration and endorsement policies, the use of BFT consensus for ordering, and the validate phase that takes place in a deterministic manner on any committer independently.
- *Transaction validation can be performed by any member of the network that has access to the output of the ordering phase, strengthening the transparency and resilience of the overall system.* Indeed, a new party that joins the network can easily replay the history of transactions and be assured of the correctness of the reported system state.
- *Chaincodes can be used to accommodate additional logic/applications as per the required system's extensions.*

Unfortunately, the current HLF implementation fails to meet performance, and scalability requirements of CBDC systems, for the following reasons:

- *Tightly coupled endorser and committer.* In HLF, each peer functions as an endorser and a committer on a single node. However, this close coupling limits the performance for several reasons:
  (1) *Competition for CPU Resources:* The endorser concurrently signs the results of multiple executed transactions, while the committer verifies the signatures of multiple transactions within a block, all in parallel. Unfortunately, this performance is capped by the number of CPU cores available on a single node.
  (2) *Sequential execution in committer*: The committer sequentially checks the read-set of each transaction in a block against the committed state to ensure serializability. Only after this validation process can the committer proceed to commit all valid transactions to the ledger at once and move to the next block. This sequential and non-pipelined execution adds a significant overhead to the commit path.
  (3) *Disk Write Bandwidth Constraint*: The committer's performance is further constrained by the available disk write bandwidth on a single node.
  (4) *Exclusive Access to Ledger State*: The endorser and committer vie for exclusive access to the ledger state. This competition arises because executing transactions (endorser operation) and accessing/writing to the system's state (committer operation) are mutually exclusive operations, necessitated by the limited concurrency control implemented in the system's state database.
- *Limited flexibility on the endorser functionality.* As explained in the previous sections, HLF achieves high degrees of resilience in chaincode execution by protecting state updates of the latter with multi-party endorsement policies. As a result, transactions typically include multiple signatures (corresponding to the various endorsers involved) resulting into

unnecessarily complex transaction format and storage overhead. Notice that the transaction size has an immediate impact on the performance of the ordering service, as consensus protocols are typically communication bandwidth sensitive. Moreover, the default implementation of HLF endorser enforces the execution of the same logic across different endorsers of the chaincode, and prevents the use of chaincode- and endorser-specific secrets throughout the endorsement process. Departing from the default implementation of the HLF endorser will allow endorsers to use application-specific secrets, to communicate with each other, and even have them submit the transaction directly to the orderers.

- *Limited throughput by existing Byzantine Fault Tolerant Consensus Algorithms.* As mentioned before, HLF employs a consensus protocol in the *order* phase, that outputs the system's transaction ledger. Traditional consensus protocols that resist arbitrary behavior from a subset of the participants (aka byzantine fault tolerant consensus) demonstrate low transaction throughput as the number of participating nodes increases. In fact, in leader based consensus protocols [16–19] the primary quickly saturates its egress bandwidth as it is the one that broadcasts the block to all other participants. While [20] tackles this problem by running multiple instances of consensus across several nodes and thus distributing the leader network bandwidth load across several participants, further research is required to assess the feasibility of incorporating such a protocol into HLF.

3.2 Optimising HLF for CBDC applications

As current instances of HLF [21] do not meet the performance and message flow needs of retail CBDC, we enhanced HLF in four ways: (1) we simplified the transaction format of HLF making it easier for the committers to unpack the relevant data (read-write sets and signatures) and for clients and endorsers to create it. (2) we reworked the HLF endorser to i) produce more compact transactions via the use of threshold signature schemes, ii) operate chaincodes in a more flexible manner, allowing application logic in them to leverage endorser- and chaincode-specific secrets, and establish communication channels with other endorsers; (3) we reworked the HLF committer to allow high degree of parallelism in the validate phase of transaction processing; (3) we reworked the HLF peer to i) decouple endorser and committer component to reduce contention for CPU resources; ii) make committer a distributed transaction processing spanning several nodes; iii) simultaneously execute read-set validation with high degree of parallelism by using read-write dependency analysis; iv) employ a sophisticated database that supports high degree of concurrency control so that the endorsers and committers can simultaneously access the state; (4) we reworked the HLF ordering service to accommodate a State of the Art BFT algorithm.

3.2.1 Simplifying Transaction Format. HLF transaction is a nested data schema of opaque byte blobs of serialized protobuf messages (cf. Fig 4a). Each layer in the data schema identifies the type of the message to be opened in the next layer. This design facilitates extendability as new data fields can be introduced without impacting the overall data schema. On the downside, this nested hierarchy, incurs significant computational overhead, as processing each layer involves memory allocation and copying. In contrast, a flat data schema ensures that the different fields of the transaction can be directly accessed in constant time, without memory copying or allocation.

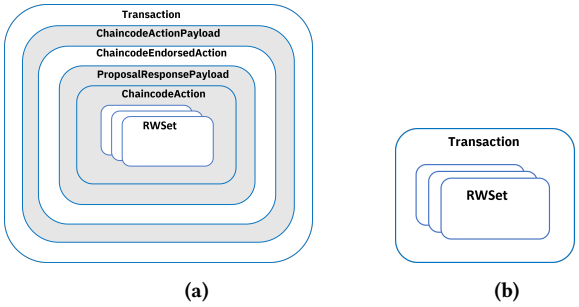

Figure 4: Transaction format in current and new version of Hyperledger Fabric: (a) HLF nested transaction format. Gray indicates a protobuf message encoded as opaque bytes; (b)New transaction format. Each transaction contains a read-write set for several namespaces. Each such namespace can contain reads, writes or read-writes.

To simplify HLF transaction and decrease the overhead of its processing, we introduce a new transaction format, which unlike the old one, is flat, enabling hence, direct and fast access to each element in the transaction, (i.e., without nested deserialization).

The new transaction is comprised of a unique transaction identifier, a list of signatures that correspond to the endorsements of the executed chaincodes, and multiple lists of read-write sets organized by namespace. Each namespace identified in the transaction corresponds to a ledger partitions that will be read and/or updated during the validate phase.

Similarly to the old HLF transaction, the new transaction defines a read-write set as follows: (1) each entry in a read-set consists of a key in the corresponding namespace and a version number. The version number allows the committers to efficiently check if the chaincode has read the latest value of the key. On the other hand, (2) each entry in the write-set is a pair ⟨key, value⟩ that will be persisted in the system state if the transaction is successfully committed.

We stress here that this transaction format is versatile, albeit flat: recall that transaction validation consists of checking if the transaction (1) reads fresh states, and (2) satisfies the endorsement policies of the namespaces being updated. If both checks succeed, then the committers (3) write the new state following the write-sets in the transaction. Including only read-write sets, the identifiers of the namespaces, and the signatures of the endorsers should suffice to perform all of these checks.

3.2.2 Reworking the HLF Endorser. Fig. 7 depicts the new endorser architecture.

The new endorser can host and execute multiple chaincodes, whereby, each **chaincode** is associated with a signing key, which is stored within a **key management component**. This component maps the chaincode to its key and ensures that the key is available to the **signer component**, which in turn, signs the result of the chaincode execution. The endorser also comes with a **coordinator**

NameSpace	"NS_ID"
ReadSet	[]<key, version>
WriteSet	[]<key, value>

Figure 5: Each read-write set (referred to as RWSet in Figure 4) identifies a namespace in the ledger and describes transaction dependencies (read-set) and state updates (write-set).

that receives transaction proposals from clients and dispatches them to the right chaincode for execution.

In contrast with the standard endorser in HLF, the new endorser is extended with

- **peer-to-peer communication** that accommodates interactions between endorsers when the application demands it.
- an **off-chain database** that records application-specific data required for chaincode execution but never stored in the ledger.
- **threshold signing**, thanks to which, a transaction will cary only one signature per chaincode execution, as opposed to a multi-signature; reducing hence, its size and the number of signature verifications to be performed at time of its validation. The new endorser supports three threshold signature algorithms BLS [22], ECDSA [23], and EdDSA [24].

Finally, the new endorser is not required to be a **committer**. For example, an endorser that only executes chaincodes that do not require frequent access to the latest state of the ledger for correct execution can do without being a committer. Instead, it can access the state of the ledger through a lightweight **query service** on demand.

3.2.3 Increasing concurrency control in HLF Committer. To address the limitations of HLF discussed in 3.1, we have proposed a new committer architecture composed of various services spanning several nodes. The new committer architecture is depicted in Figure 7. To have high degree of parallelism in validation of signatures against the endorsement policy, we introduced a new service called signature validator, whose sole purpose is to utilize all available CPU cores to validate signatures. Further, to optimize disk read and write bandwidth utilization for both committer and endorsers, we have introduced a new service called shard server. Each shard server maintains a partition of the ledger state, validates read-set freshness, applies write-set to the ledger, and also directly provides query service to the endorser. Finally, we introduced a coordinator which is responsible for processing transactions by communicating with signature validators and shard servers. The coordinator and shard servers execute 2-phase commit protocol[25] between them to perform distributed validation and commit of transactions.

Furthermore, to increase the degree of parallelism in checking the freshness of the read-set, we introduced transaction dependency analysis in the coordinator. In existing committer, each transactions read-set is validated sequentially because the validity of a prior transaction may affect the validity of a later transaction. Let's

Table 1: The read-write set of transactions present in block B_1 and B_2.

Block	Tx.	Read Set (key, version)	Write Set (key, value)	valid
B_1	T_1	$(k_6, 1)$	(k_1, v_1)	✓
	T_2	$(k_1, 2)$	(k_6, v_6)	✗
	T_3	$(k_3, 1)$	(k_6, v_7)	✓
	T_4	-	(k_5, v_5)	✓
B_2	T_1	(k_5, nil)	(k_7, v_1)	✗
	T_2	$(k_4, 3)$	(k_4, v_4)	✓

consider an example to understand the necessity of serial validation of read-set. Consider two blocks of transactions, as shown in Table 1. The table shows states read by each transaction under *Read Set* column and states written by each transaction under *Write Set* column. Note that the version is a monotonically increasing number associated with each state and is updated whenever the state is modified. The transaction T_1 being valid would render T_2 invalid. This is becaues T_2 read the state k_1 at version 2, while the T_1 updates the state k_1 to new version. Consequently, read-set freshness check for T_2 would fail. If we were to process both T_1 and T_2 in parallel, it is possible to get T_2 as valid and T_1 as invalid or to get both transactions as valid, thereby violating serializability. However, T_1 and T_4 are independent of each other and can be processed in parallel. To ensure correctness, the existing committer processes each transaction sequentially.

In the new committer architecture, we track dependency between transactions using a dependency graph and process independent transactions in parallel. This graph contains a node per transaction. We use the term node and transaction interchangeably. Let's assume we have two transactions, T_i and T_j, with T_i appearing in a block before T_j (where T_j can be in the same block or any subsequent block). An edge from T_j to T_i denotes that the transaction T_i's read-set must be validated before validating T_j's read-set. The following are the three types of dependencies that can create an edge from T_j to T_i, where $i < j$:

- ***read-write dependency*** ($T_i \xleftarrow{rw(k)} T_j$): T_i writes a new value to state k and updates its version. T_j reads the previous version of state k. If T_i is valid, T_j must be invalid because the read version is not the latest version. This read-write dependency is also called fate dependency as the fate of T_j is decided by T_i.
- ***write-read dependency*** ($T_i \xleftarrow{wr(k)} T_j$): T_j writes a new value to state k and updates its version. T_i reads the previous version of state k. Regardless of the validity of T_i, T_j can be validated. We use this dependency to ensure T_j is not committed before T_i. Otherwise, T_i can be marked invalid as it reads the previous version of state k.
- ***write-write dependency*** ($T_i \xleftarrow{ww(k)} T_j$): Both T_i and T_j write to the same state k. Regardless of the validity of T_i, T_j can be validated. We use this dependency to ensure T_j is not committed before T_i. Otherwise, the write made by T_j would be lost forever.

Note that an edge can only go from a newer transaction to an older transaction, i.e., if $T_i \leftarrow T_j$ then $i < j$, because the commit order is determined in the *ordering* phase. Thus, there are no cycles in the dependency graph. The dependency graph of transactions in Table 1 is shown in Figure 6. The transactions T_1, T_4 in block B_1 and T_2 in block B_2 are dependency-free and can be processed in parallel. Other transactions have to wait for their dependencies to be validated and committed or aborted.

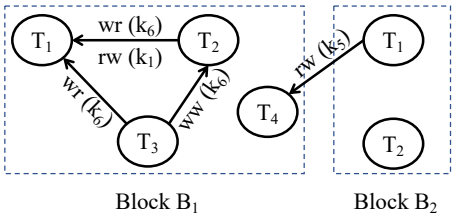

Figure 6: Dependency graph of transactions in Table 1.

Transaction Flow. Blocks that arrive from the ordering service first reach the coordinator, which verifies the signature of the orderers on each block, and inserts the ordered set of transactions into its *transaction dependency graph*. Transactions which do not depend on any other transactions are processed in *parallel*, and once the validation of a transaction concludes (either by having the transaction committed or rejected), the transaction is removed from the graph and dependents of this transaction are updated. If there are any transaction that have fate dependency, i.e., *rw-dependency*, on the committed transaction, they are marked as invalid and removed from the graph.

More specifically, the coordinator sends the ordered transactions to one of the signature validators, which verifies *in a stateless manner* whether the transaction complies with the endorsement policies of the namespaces identified in the transaction. The failure to pass these checks results in the early the rejection of the transaction. Meanwhile, the 2-phase commit protocol is executed between the coordinator and shard servers. The coordinator dispatches the transaction to the relevant shard servers to check the freshness of the read-sets in the transaction against the (committed) ledger state. That is, for each key in the read-sets, the coordinator calls the associated shard server *in parallel*, whereas upon call, each shard server checks if their assigned read-set entries are fresh or not. If that is the case, the shard server sends back OK; else, it returns NOK. If one of the shard servers responds with NOK, the coordinator marks the transaction as *invalid*. Concurrently, the coordinator sends each entry in the write sets to its respective shard server, while also verifying if the transaction has passed the signature validation. If the signature validation succeeds, the coordinator considers the transaction *valid*; otherwise, *invalid*. Finally, if the transaction is valid, the coordinator notifies the shard servers with entries in the write-sets to apply the updates, and we say that the transaction is *committed*. Else, the shard servers are asked to discard the updates.

3.2.4 Reworking HLF ordering service. To order transactions, we leverage an improved version of the Narwhal and Tusk BFT consensus [11], called ARMA. ARMA (as Narwhal and Tusk) replicates transaction batches in parallel and uses a BFT protocol to order

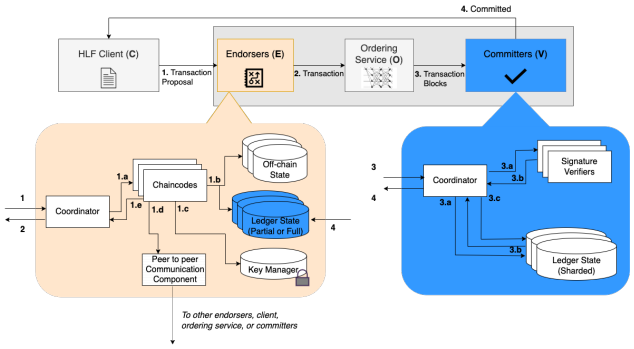

Figure 7: Hyperledger Fabric Architecture, and the new endorser and committer architecture within it. (1) Client submits a transaction proposal to the endorsers of the chaincode to be invoked; the endorsers trigger the corresponding chaincode's execution (1a), by accessing an off-chain state database (1b) and/or the ledger state (1c); after the read/write set is compiled, the chaincode results are signed by the endorser using the endorser's key manager (1c), while depending on the endorser's configuration, this could trigger communications with other endorsers, e.g., for a collaborative threshold signature generation; notice that the ledger state can be implemented as a replica of the actual system's ledger (that would require endorser to run a committer), or via a mechanism that connects to a committer, and only retrieves (endorser/chaincode-relevant) parts of the ledger's state. (2) Transaction is submitted to the ordering service for ordering and from there (3) the transaction is delivered within a block to the committer's coordinator component; the coordinator exchanges with the signature verifiers and the shards to ensure validity of the signatures and read-set freshness (3.a/3/b), to finally commit state updates (3.c) and respond to the client (4).

the headers of the corresponding transaction batches, instead of the transactions themselves. Since the size of the headers is much smaller than the size of the batch of the transactions (it is in the order of a few hundreds of bytes per transaction *batch*), ARMA (just like its predecessor [11]), overcomes the network bandwidth bottleneck described in Section 3.1. Further distinguishing itself from closely-related work like [11], ARMA is censorship resistant. That is, it ensures that any transaction originating from an honest participant will be eventually ordered.

4 TOKEN SYSTEMS FOR CBDC

In this paper, a token is a digital representation of a fungible financial asset (e.g., CBDC, tokenized deposits, or securities). It is defined as a triple (owner, type, value), and in systems with a single asset type, a token can be reduced to the pair (owner, value). For simplicity, in the remainder of this paper we focus on token systems with a single type.

The state of a token is maintained in a ledger, and the updates it undergoes are governed by three operations: Issue, Transfer and Redeem. An Issue introduces new tokens into the system, and it is a privileged operation that only a select few (or one) can initiate. A Transfer, on the other hand, allows a user to send a token to another user, whereas a Redeem removes tokens from the system. Similar to Issue, Redeem is a privileged operation.

We note here that the operation that the ledger receives to update the state of a token is wrapped in a message, commonly known as a *transaction*. For transactions we adopt the Unspent Transaction Output (UTXO) model, because of the advantages on the privacy and concurrency control front[7].

For ease of exposition, we conflate the transaction with the operation it carries, for e.g., the transaction that carries an Issue operation is called an Issue transaction.

4.1 UTXO-based Token Transactions

A UTXO-based transaction is defined by a set of inputs and a set of outputs. Inputs are references to tokens that have been generated in the past and are in the possession of the transaction originator (payer or issuer), while outputs are new tokens that are created as a result of the transaction. If the transaction is valid, then the inputs are deleted from the ledger (i.e., marked as spent), whereas the outputs are added, ready to be used in subsequent transactions.

Assume that user A would like to transfer a token of value v to another user B. To that end A builds a Transfer transaction as follows: (1) selects from the tokens they own a subset $(\tau_0, ..., \tau_{n-1})$ whose sum value $\geq v$; (2) defines the inputs of the transaction as $\mathrm{id}_0, ..., \mathrm{id}_{n-1}$, whereby id_i is the identifier of τ_i in the ledger; (3) computes a token τ_B of value v whose owner is B, and a second token τ_A intended for A whose value is $(\sum_{i=1}^{n} v_i) - v$, where v_i is the value of τ_i; (4) sets the outputs of the transaction to τ_B and τ_A.

To guarantee that only the owner of a token can spend it, the Transfer transaction must be signed by the owners of the inputs in the transaction. Following the previous example, user A will sign the Transfer transaction prior to submitting it for validation.

At settlement time, a Transfer transaction is subjected to the following checks, and all need to succeed for the transaction to be accepted:

(1) All the inputs identified in the transaction exist in the ledger. This corresponds to the **double spending** check.
(2) The sum value of the inputs equals the sum value of the outputs. This is the **equality of sum** check. Note that successful checks of double spending and equality of sum reflect that the total balance of tokens is preserved after the transfer. [7]
(3) The rightful owners of the inputs signed the transaction. This is the **authorization** check.

Each output of an accepted transaction is assigned a unique identifier in the ledger. Usually, this is computed as a function of the unique identifier of the transaction it creates it and its index in that transaction (e.g., index of τ_B is 0 and index of τ_A is 1).

Prior to any Transfer, tokens must be first created via an Issue transaction. In the UTXO model, an Issue is a transaction without inputs, and the sum value of its outputs stands for the value being injected into the system. The validation of an Issue checks if it was signed by one of the parties authorized to create the tokens.

Finally, a Redeem in the UTXO model is a Transfer that contains an output whose owner is empty. We call the output with empty owner a *redeemed output*, and it is non-transferable: namely, no one can sign a Transfer that includes as an input a redeemed output. This de-facto reduces the value of tokens in circulation.

[7]Notice that if multiple types of tokens are supported in the system, the equality of sum should ensure that the types of inputs and outputs are identical.

In the following sections, we describe how varying degrees of transaction privacy can be achieved in the UTXO model: starting from systems that only achieve user anonymity, to systems that also ensure transaction confidentiality, concluding with systems that guarantee transaction unlinkability. Transaction confidentiality refers to the property that a transaction does not leak the values of the inputs and the outputs, whereas transaction unlinkability refers to the property that no one can link two transactions together (i.e., link the input of one transaction to the output of a previous one).

For the sake of simplicity, we only consider token systems with a single issuer. Furthermore, since the validation of Issue transactions, in single-issuer settings, does not change from one token system to another, we omit it from the description. In fact, an Issue is valid if it was signed by the authorized issuer. Finally, we focus on Transfer transactions, as details pertaining to Redeem transactions can be inferred from the description of the former.

4.2 Anonymous Token Systems

In the UTXO model, an anonymous Transfer hides the *long-term identities* of the owners of the inputs and the outputs. In permissionless systems like Bitcoin, anonymity is achieved by having as token owners ephemeral public keys that are not tied to any real-world identity. In permissioned systems where only registered users are allowed to transact, ephemeral public keys are not suitable. Instead, anonymous credentials [26, 27] are recommended. More specifically, a registration authority grants each enrolled user a credential that binds their unique enrollment identifier $\mathrm{eid} \in \mathbb{Z}_p$ with their secret key $\mathrm{sk} \in \mathbb{Z}_p$. $\mathbb{Z}_p$ is the set $\{0, 1, ..., p-1\}$, where p is prime. Roughly speaking, the user credential cred is a signature of the registration authority on the pair (sk, eid). If pk_R denotes the public key of the registration authority and Verify the algorithm to verify its signatures, then $\mathrm{Verify}(\mathrm{pk}_R, \mathrm{cred}, (\mathrm{sk}, \mathrm{eid})) = 1$.

A token τ is consequently defined as a pair (c, v)[8], with $c = \mathrm{Commit}(eid, t)$ is the hiding commitment of the enrollment identifier of the owner eid using some randomness $t \in \mathbb{Z}_p$, and $v \in \mathbb{Z}$ is the value of the token. This assures that the identity of the owner of τ is not leaked to those with access to the ledger.

Let $\mathrm{ttx} = (in_0, in_1, out_0, out_1)$ be the Transfer transaction that spends token $\tau_0 = (c_0, v_0)$ and token $\tau_1 = (c_1, v_1)$, i.e, in_0 (resp. in_1) identifies τ_0 (resp. τ_1) in the ledger. To sign ttx anonymously, the owner of τ_0 and τ_1 produces two signatures of knowledge σ_0 and σ_1 on ttx such that each σ_b, $b \in \{0, 1\}$ proves the following statement:

$$\begin{aligned} &\exists\ \mathrm{cred}_b, \mathrm{sk}_b, \mathrm{eid}_b, t_b \text{ s.t.} \\ &c_b = \mathrm{Commit}(eid_b, t_b) \\ &1 = \mathrm{Verify}(\mathrm{pk}_R, \mathrm{cred}_b, (sk_b, \mathrm{eid}_b)) \end{aligned}$$

We recall that a signature of knowledge is a non-interactive zero-knowledge proof that can also be used to sign messages.

The user then submits for validation the signed transaction

$$\mathrm{ttx} = (in_0, in_1, out_0, out_1, \sigma_0, \sigma_1).$$

Validation of Anonymous Transfer *Transactions.* ttx is validated by checking that (1) the tokens τ_0 and τ_1 identified by in_0 and in_1 exist in the ledger, (2) the sum value of τ_0 and τ_1 equals the sum

[8]In systems without any privacy, τ is pair (eid, v), i.e., the token reveals the identity of its owner and its value.

value of out_0 and out_1, and (3) σ_0 and σ_1 are valid signatures of knowledge relative to tokens τ_0 and τ_1 and the public key pk_R of the registration authority. If ttx passes all the checks, then τ_0 and τ_1 are deleted, while out_0 and out_1 are added to the ledger.

Now that we showed how anonymity is achieved when users are compelled to use their long-term identities to authorize Transfer transactions, we describe next how to hide the values of the inputs and the outputs of the transactions.

4.3 Anonymous and Confidential Token Systems

In [28], Poelstra et al. show how to hide the values and still enable the balance preservation check. A token τ in [28] is a hiding commitment to pair (eid, v), i.e, $\tau = \text{Commit}(\text{eid}, v, t)$[9] for $v \in \mathbb{Z}_p$ and some randomly-chosen $t \in \mathbb{Z}_p$.

Let $\text{ttx} = (in_0, in_1, out_0, out_1)$ be the Transfer transaction that spends token τ_0 and token τ_1.

Since the values of the inputs and outputs are now hidden, ttx contains a zero-knowledge Π_{Sum} that proves that the sum of the inputs equals the sum of the outputs. That is:

$$\begin{aligned}\forall b \in \{0,1\} : \exists\ &\text{eid}_{in,b}, v_{in,b}, \text{eid}_{out,n}, v_{out,b}, t_{in,b}, t_{out,b} \text{ s.t.}\\ \tau_b &= \text{Commit}(\text{eid}_{in,b}, v_{in,b}, t_{in,b})\\ out_b &= \text{Commit}(\text{eid}_{out,b}, v_{out,b}, t_{out,b})\\ v_{in,0} + v_{in,1} &= v_{out,0} + v_{out,1} \mod p\\ v_{out,b} &< max.\end{aligned}$$

max refers here to the maximum value a token can hold. By restricting the range of the individual values of the outputs to $[0, max($, where max $<< p$, we ensure that there is no field wrap-arounds when one adds up the values of the outputs.

Finally, to authorize ttx, the owner of τ_0 and τ_1 produces two signatures of knowledge σ_0 and σ_1 on $(in_0, in_1, out_0, out_1, \Pi_{\text{Sum}})$ such that each σ_b, $b \in \{0,1\}$, proves the following:

$$\begin{aligned}&\exists\ \text{cred}_b, \text{sk}_b, \text{eid}_b, v_b, t_b \text{ s.t.}\\ &\tau_b = \text{Commit}(eid_b, v_b, t_b)\\ &1 = \text{Verify}(\text{pk}_R, \text{cred}_b, (sk_b, \text{eid}_b))\end{aligned}$$

The user then submits for validation

$$\text{ttx} = (in_0, in_1, out_0, out_1, \Pi_{\text{Sum}}, \sigma_0, \sigma_1).$$

Validation of Anonymous and Confidential Transfer *Transactions.* ttx is validated by checking that (1) the tokens τ_0 and τ_1 identified by in_0 and in_1 exist in the ledger, (2) Π_{Sum} is a valid zero-knowledge proof that proves that the sum value of τ_0 and τ_1 equal the sum value of out_0 and out_1, and (3) σ_0 and σ_1 are valid signatures of knowledge relative to tokens τ_0 and τ_1, and the public key pk_R of the registration authority. If ttx passes all the checks, then τ_0 and τ_1 are deleted, and out_0 and out_1 are added to the ledger.

Following these steps, one can produce Transfer transactions that are both anonymous and confidential. Next, we describe how these transactions can be made unlinkable.

[9] If anonymity is not required, then the token τ can be defined as pair (eid, c), where $c = \text{Commit}(v, t)$.

4.4 Unlinkable Token Systems

In order to prevent double spending attacks, token transactions identify the inputs, and when deemed valid, result in the deletion of said inputs. This however, undermines transaction unlinkability, and with user privacy. In particular, by revealing the inputs of a transaction, anyone with access to the ledger can infer that the owner of an output in one transaction (whether Issue or Transfer) is the initiator of the Transfer spending that output.

Hence, to guarantee transaction unlinkability, it is crucial not to reveal the tokens being spent in a Transfer. The challenge in this case is to show that an input to a Transfer (1) is actually the output of a valid transaction; and (2) hasn't been spent before.

In Zerocash [29], Bensasson et al. demonstrated how to efficiently achieve both goals. The first is realized by relying on zero-knowledge proofs of membership, which are leveraged to prove that a token is in the ledger without revealing the location of the token. The second is met by assigning unique serial numbers to tokens, which are revealed at time of spending.

We now describe how to enhance anonymous and confidential transactions with unlinkability.

We start from the credential cred given to a user in the system. cred is now defined as a signature by the registration authority on triple $(\text{sk}, K, \text{eid})$, where K is a secret key for a pseudo-random function PRF. Moreover, a token τ becomes a hiding commitment $\text{Commit}(\text{eid}, v, r, t)$, with r and t being two random numbers in $\mathbb{Z}_p$. A Transfer that spends τ, includes as input a pair $in = (\tau', sn)$. $\tau' = \text{Commit}(\text{eid}, v, r, t')$ is a randomization of τ and sn is τ's serial number, computed as $sn = \text{PRF}(K, r)$.

Let ttx be the transfer transaction that spends two tokens (τ_0, τ_1) and creates two outputs out_0, out_1. ttx, correspondingly, carries tuple $(in_0, in_1, out_0, out_1)$ such that $in_0 = (\tau'_0, sn_0)$ and $in_1 = (\tau'_1, sn_1)$. ttx also carries the zero-knowledge proof Π_{Sum} that proves that out_0 and out_1 sum up to the same value as in_0 and in_1, i.e, Π_{Sum} shows that:

$$\begin{aligned}\forall b \in \{0,1\} :\ \exists\ &\text{eid}_{in,b}, v_{in,b}, \text{eid}_{out,n}, v_{out,b}, r_{in,b}, r_{out,b}, t_{in,b}, t_{out,b} \text{ s.t.}\\ \tau_b &= \text{Commit}(\text{eid}_{in,b}, v_{in,b}, r_{in,b}, t_{in,b})\\ out_b &= \text{Commit}(\text{eid}_{out,b}, v_{out,b}, r_{in,b}, t_{out,b})\\ v_{in,0} + v_{in,1} &= v_{out,0} + v_{out,1} \mod p\\ v_{out,b} &< max.\end{aligned}$$

ttx also includes a zero-knowledge proof Π_{sn} that shows that sn_0 and sn_1 were computed correctly as a function of τ'_0 and τ'_1. Namely, Π_{sn} ascertains that:

$$\begin{aligned}\forall b \in \{0,1\} :\ \exists\ &\text{cred}_b, \text{sk}_b, K_b, \text{eid}_b, v_b, r_b, t'_b \text{ s.t.}\\ \tau'_b &= \text{Commit}(\text{eid}_b, v_b, r_b, t'_b)\\ sn_b &= \text{PRF}(K_b, r_b)\\ 1 &= \text{Verify}(\text{pk}_R, \text{cred}_b, (sk_b, K_b, \text{eid}_b))\end{aligned}$$

So far ttx only shows that the sum of inputs matches the sum of outputs and that sn_0 and sn_1 are computed correctly with respect to the information encoded in τ'_0 and τ'_1. ttx does not guarantee, however, that τ'_0 and τ'_1 are the randomization of two tokens that already exist in the ledger. To tackle this issue, we leverage zero-knowledge membership proofs. These are proofs that allow one

to prove that an element is in a set without revealing any information about the said element. These proofs can be implemented using various techniques including Merkle trees, accumulators, or signatures. For efficiency purposes and similar to [12], we rely on signature-based membership proofs. More specifically:

- We introduce *certifiers* whose majority is assumed to be honest. The certifiers crawl the ledger, and upon *users' requests* jointly sign the outputs of valid transactions, using threshold signatures. We call such an operation a *certification* of an output. The certifiers store the resulting signatures and return them upon query.
  We denote hereafter pk_C the public key associated with the threshold signature and γ the output of an execution of the threshold signature.
- A user then fetches the signatures of the tokens they own, and stores them for later use. We call tokens that are signed by the certifiers, **certified** tokens.
- ttx is enhanced with a third zero-knowledge proof Π_{Exist} that proves the following:

$$\begin{aligned}\forall b \in\{0,1\}: \ \exists\, \gamma_b, \text{eid}_b, v_b, r_b, t_b, t'_b \text{ s.t.}\\ \tau'_b = \text{Commit}(\text{eid}_b, v_b, r_b, t'_b)\\ \tau_b = \text{Commit}(\text{eid}_b, v_b, r_b, t_b)\\ 1 = \text{Verify}(\text{pk}_C, \gamma_b, \tau_b)\end{aligned}$$

At this point, $\text{ttx} = (in_0, in_1, out_0, out_1, \Pi_{\text{Sum}}, \Pi_{sn}, \Pi_{\text{Exist}})$, and the owner(s) of τ_0 and τ_1 generate two signatures of knowledge (σ_0, σ_1) on ttx for the following statement:

$$\begin{aligned}\forall b \in \{0,1\}: \ \exists\, \text{cred}_b, \text{sk}_b, K_b, \text{eid}_b, r_b, t_b \text{ s.t.}\\ \tau'_b = \text{Commit}(\text{eid}_b, v_b, r_b, t'_b)\\ 1 = \text{Verify}(\text{pk}_R, \text{cred}_b, (sk_b, K_b, \text{eid}_b))\end{aligned}$$

The resulting transaction

$$\text{ttx} = (in_0, in_1, out_0, out_1, \Pi_{\text{Sum}}, \Pi_{sn}, \Pi_{\text{Exist}}, \sigma_0, \sigma_1), \text{ where}$$
$$in_0 = (\tau_0, sn_0)\ ;\ in_1 = (\tau_1, sn_1)$$

is then submitted for validation.

Validation of Unlinkable Transfer *Transactions.* ttx is validated by checking that (1) sn_0 and sn_1 do not appear in ledger, (2) Π_{Exist} is a valid zero-knowledge proof that τ'_0 and τ'_1 are the randomization of certified tokens, (3) Π_{sn} is a valid zero-knowledge that shows that sn_0 and sn_1 were correctly computed, (4) Π_{Sum} is a valid zero-knowledge proof that shows that τ'_0 and τ'_1 sum up to the same value as out_0 and out_1, and (5) σ_0 and σ_1 are valid signatures of knowledge by the owner(s) encoded in τ'_0 and τ'_1. If ttx passes all the checks, then $(sn_0, sn_1, out_0, out_1)$ are added to the ledger.

4.5 Audit in Token Systems

Financial applications often require audit capabilities. Token systems, as a result, must accommodate such capabilities by design. In a token system that does not offer any privacy protections, audit is supported automatically: anyone with access to the ledger can check the identities of the transacting parties and the values of the transactions. In contrast, token systems that preserve user privacy make it impossible for auditors to inspect transactions just by crawling the ledger, which now depending on the implementation, only contain partially or fully obfuscated transactional information. Audit in such a setting is facilitated by one of two approaches:

(1) **Active Audit:** The initiator of a transaction, be it Issue or Transfer, provides the transaction together with some metadata to the authorized auditors. The metadata contains the information of the transaction in the clear. The auditors check if the transaction information matches the transaction and if it complies with the audit rules. If both checks succeed, then the auditors sign off the transaction and return the signature to the transaction initiator. When the transaction is submitted for validation, it undergoes an additional signature verification that ascertains if the auditors inspected and accepted the transaction.
(2) **Passive Audit:** A transaction in this approach is augmented with ciphertexts that can only be decrypted by the authorized auditors. To guarantee that the auditors can retrieve the transactional information successfully without any external help, the ciphertexts carry proofs that confirm that they encrypt the correct information and can be decrypted by the auditors.

5 SCALABLE ARCHITECTURE FOR CBDC

The main components of our CBDC architecture are the *user wallets* and the *settlement engine*. Each user wallet is held by either an issuer, a payer or a payee, and its task is to translate the instructions of its holder into token transactions. These transactions are settled by the settlement engine, which builds on HLF in the following way:

- A chaincode, we call the **token chaincode**, implements the validity checks of the token transactions described in Section 4. Notably, the token chaincode verifies the signature(s) of the issuers and/or the token owners, and the zero-knowledge proofs whenever transaction privacy is supported. These checks are performed with minimal access to the ledger state; the chaincode only accesses the chaincode's version or the system configuration (e.g., identity rules), which seldom change.
- Upon receiving a token transaction, the endorsers of the token chaincode, referred to as **token endorsers**, output the results of the transaction execution in the form of read-write sets, such that: (1) the read-sets describe the read-dependencies of the token transaction, including the most recent versions of the system configuration; and (2) the write-sets reflect the state updates that the token transaction will apply to the ledger if committed (e.g., creating new tokens or marking old tokens as spent). The token endorsers then sign the transaction's execution results using their (threshold) signing key. The token endorsers can be additionally configured to deliver the endorsed transaction to the ordering service on behalf of the user wallets that constitute the HLF client in this case.
- The ordering service establishes the total order of the endorsed token transactions by batching them into a sequence of signed blocks.
- The committers validate the transactions against the endorsement policy of the token chaincode and the content of the ledger, and update the state of the ledger accordingly. It is

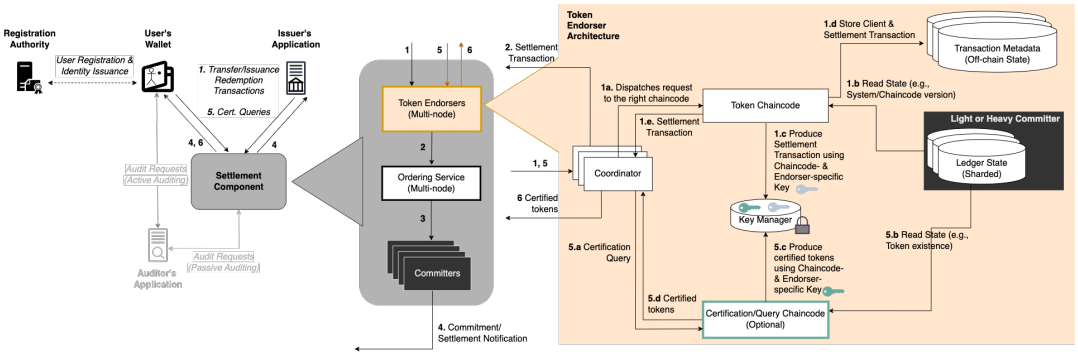

Figure 8: Scalable CBDC Architecture. (1) Client transaction requesting issuance of new CBDCs or transfer of CBDCs is submitted to the system; (2) Token Endorsers (front end component, called Coordinator) receive the Client transaction and execute the token chaincode (1.a) by occasionally accessing the ledger state (1.b); upon completion of the chaincode's execution, the endorser produces a signed version of the settlement transaction (1.c), and stores the associated client transaction to its Transaction Metadata database (1.d); it finally returns the settlement transaction to the Coordinator (1.e) that forwards the former to the HLF ordering service, that delivers the transaction in a block to the committers of the system (including the ones endorsers rely on), where the transaction proposed state updates are stored on the ledger triggering further notification to the transaction client originator (4). In privacy-preservation scenarios, users may request that their tokens are certified via a query submitted to the system's endorsers (5), triggering the execution of the query chaincode or / certification chaincode (5.b), that after consulting with the system's ledger (5.c) leverage the endorser's key manager (5.d) to certify the requested token and return it to the client (5.e and 6).

at this stage that double-spending attempts are caught, and system upgrades are enforced.

The settlement engine also includes a *query service* that responds to user queries on the results of the processed transactions. The query service can be built either on top of the token chaincode, or in a dedicated chaincode with its own endorsement policy.

5.1 Transaction Model and Lifecycle

A CBDC system is governed by three types of *business* transactions: Issue, Transfer, and Redeem. In our architecture, these transactions come in two forms: *client transactions* and *settlement transactions*. A client transaction is the message submitted to the settlement engine by a user wallet, and whose content varies depending on the business transaction it serves and the token exchange mechanism employed. In HLF parlance, this is the *proposal* submitted to the token endorsers by an HLF client, which is the user wallet in this instance. A settlement transaction, on the other hand, is a compact version of the client transaction, submitted by the token endorsers directly to the ordering service of the settlement engine. Using HLF terminology, this is the *HLF transaction*, which includes the read-write sets and the signatures of the endorsers. After ordering, the settlement transaction is handed over to the committers, which following the logic described in Section 3.2.3, validate the settlement transaction and commit its write-set if deemed valid, concluding hence, the lifecycle of the token transaction.

For ease of exposition, we distinguish between token systems with transaction unlinkability (i.e., full privacy) and those without. Notably, Section 5.2 demonstrates that systems without unlinkability share the same client transaction format and endorsement logic, despite offering different privacy guarantees, whereas Section 4.4 describes the token systems with unlinkability and highlights the differences.

5.2 Token Systems without Transaction Unlinkability

Following the description in Section 4, transactions in such systems identify the tokens to be spent, revealing what we call the *transaction graph* (i.e. the relation between the inputs and the outputs in the ledger). However, depending on the desired privacy level, the outputs of the transactions can

(1) reveal all the transaction information (zero privacy);
(2) hide the identities of the transacting parties (anonymity);
(3) hide the values of the transactions (confidentiality);
(4) hide both (anonymity and confidentiality).

Client Transaction	
TxID	String
InputIdentifiers	[]String
Inputs	[][]byte
Outputs	[][]byte
ZKProof	[]byte
Signatures	[][]byte

Figure 9: Client transaction format.

5.2.1 Transaction Model. In settings where transaction unlinkability is not required, the client transaction is a *standard* UTXO transaction, with inputs and outputs, and the signatures of the issuer or/and the token owners.

When transaction confidentiality is desired, the client transaction will additionally carry a zero-knowledge proof that shows that the transaction preserves the value (i.e., the sum of the inputs equals the sum of the outputs), cf. Figure 9.

A client transaction is then translated into a settlement transaction as follows. The settlement transaction of an Issue consists of a write-set in which, there are entries ⟨out_key, ser_out⟩ corresponding to the tokens to be created (i.e., the transaction outputs). The key out_key is the *unique identifier* of the output in the ledger, which is set to the concatenation of the transaction identifier, the hash of the token, and the index of the output in the transaction[10], and ser_out is the serialization of the output.

Similarly, the settlement transaction of Transfer carries a write-set containing two types of entries: the first contains the outputs of the transaction ⟨out_key, ser_out⟩, and the second the delete instructions of the inputs represented by ⟨in_key, isDelete = true⟩, where in_key is the key in the ledger of the input to be deleted. Furthermore, the settlement transaction contains a read-set with entries ⟨in_key, 0⟩. This guarantees that at time of transaction validation, double spending attempts are thwarted. Actually, if two transactions try to spend the same input, this will generate a version conflict in the read sets. If one of these two transactions gets committed, then it will delete the input, incrementing hence the version of the corresponding key, leading the other transaction to fail due to an outdated read-set entry.

Next, we explain in more details the functionalities of the components of our architecture focusing on the flow of Transfer transactions. Issue transactions adopt a similar processing flow.

5.2.2 User Wallets. The wallet securely stores the user's credential (i.e., long-term identity), defined as tuple (eid, sk, cred), whereby eid is the unique identifier of the user, sk her secret key, and cred is the signature from the registration authority on pair (eid, sk). The user wallet also stores for each token τ_i, the information that enables its spending. For example, in a token system without any privacy protections, $\tau_i = (\text{eid}, v_i)$ and the wallet stores $\langle \tau_i, \text{in\_key}_i \rangle$, where in_key_i is the identifier of τ_i in the ledger. In a token system with anonymity, $\tau_i = (c_i, v_i)$, where $c_i = \text{Commit}(\text{eid}, t_i)$ and the wallet stores $\langle \tau, t_i, \text{in\_key}_i \rangle$. Finally, in a system that also ensures transaction confidentiality $\tau_i = \text{Commit}(\text{eid}, v_i, t_i)$ and the wallet stores $\langle \tau_i, v_i, t_i, \text{in\_key}_i \rangle$.

Assuming that the user instructs the wallet to transfer a token of value v' to a payee identified by eid$'$, the user wallet assembles the corresponding transaction as follows:

- Select tokens τ_i such that the sum value of these tokens exceed v'. Without loss of generality, we assume that the wallet picked two tokens τ_0 and τ_1.
- Create two outputs out_0 and out_1, such that out_0 is intended for the payee and has value v', and out_1 is the change to be returned to the payer, and accordingly, is of value $v_0 + v_1 - v'$.
- Produce tuple $(in_0, in_1, out_0, out_1)$, where $in_0 = (\tau_0, \text{in\_key}_0)$ and $in_1 = (\tau_1, \text{in\_key}_1)$.
- When transaction confidentiality is required, compute Π_{Sum} to prove that out_0 and out_1 sum up to the same value as τ_0 and τ_1. Let $\mu = (in_0, in_1, out_0, out_1, \Pi_{\text{Sum}})$

[10]This identifier is unique, given that HLF enforces uniqueness of transaction identifiers.

1: **Input**
   $(\text{TxID}, (\tau_0, \text{in\_key}_0), (\tau_1, \text{in\_key}_1), out_0, out_1, \Pi_{\text{Sum}}, \sigma_0, \sigma_1)$;
2: **Output**
   A client transaction ctx
3: ctx ← NewClientTx()
4: ctx.TxID ← TxID
5: **for** $b \in \{0, 1\}$ **do**
6: ctx.InputIdentifiers$[b]$ ← in_key_b
7: ctx.Inputs$[b]$ ← τ'_b.Serialize()
8: ctx.Outputs$[b]$ ← out_b.Serialize()
9: ctx.Signatures$[b]$ ← σ_b.Serialize()
10: **end for**
11: ctx.ZKProof ← Π_{Sum}.Serialize()
12: return ctx

Figure 10: Building a client transaction.

- Compute TxID as the concatenation of the hash of μ and some randomness[11]. TxID is the unique identifier of the client transaction.
- Subsequently, compute two signatures σ_0 and σ_1 on (TxID, μ), as depicted in Section 4, and assemble a client transaction, as shown in Figure 10.
- Then communicate to the payee the identifier TxID of the transaction. This identifier allows the payee to track the status to the transaction in the ledger. In anonymous token systems the payer also communicates to the payee the randomness t' used to compute $out_0 = (\text{Commit}(\text{eid}', t'), v')$. When transaction confidentiality is supported, the payer additionally communicates v'.
- Finally, submit the client transaction to the settlement engine, more specifically, to the token endorsers for processing.

5.2.3 Settlement Engine. In this section, we expand on the components and functionalities of the settlement engine, illustrated in Figure 8.

Token Endorsers & Token Chaincode. In our architecture, the token endorsers support a threshold signing algorithm ThresholdSign to sign the results of the token chaincode execution (i.e, the settlement transactions). Correspondingly, each endorser is equipped with a secret share denoted tsk. Without loss of generality, we assume that there are n token endorsers and that the threshold of the signature is t. Consequently, a settlement transaction is successfully signed only if $t + 1$ *honest* token endorsers come together.

Upon receiving a client transaction (Step 1 Fig 8), the token endorsers run the token chaincode, which validates the transaction following the description in Section 4. More specifically, the chaincode checks if

- TxID contains the hash of $(in_0, in_1, out_0, out_1, \Pi_{\text{Sum}})$;
- identifiers in_key_0 and in_key_1 include the hash of τ_0 and τ_1 respectively;
- Π_{Sum} is a valid zero-knowledge proof;

[11]The randomness is only necessary when the token system does not provide any privacy protections.

```
1: Input
   ctx                                          ▷ a client transaction
   tsk                                ▷ a threshold signature secret share
2: Output
   stx                                          ▷ a settlement transaction
3: stx ← NewSettlementTX()
4: stx.TxID ← ctx.TxID
5: stx.NameSpace ← "token"
6: for b ∈ {0, 1} do
7:     RSEntry ← ⟨ctx.InputIdentifiers[b], 0⟩
8:     stx.ReadSet ← stx.ReadSet ∪ RSEntry
                                              ▷ Delete the inputs
9:     WSEntry ← ⟨ctx.InputIdentifiers[b], isDelete⟩
10:    stx.WriteSet ← stx.WriteSet ∪ WSEntry
11: end for
                          ▷ Add the created outputs to the write-set
12: for b ∈ {0, 1} do
13:    WSEntry ← ⟨stx.TxID||hash(out_b)||b, out_b.Serialize()⟩
14:    stx.WriteSet ← stx.WriteSet ∪ WSEntry
15: end for
   ▷ The token endorsers jointly generate a threshold signature
   on stx
16: φ ← ThresholdSign(tsk, stx)
17: stx.Signature ← φ.Serialize()
18: return stx
```

Figure 11: Building a settlement transaction.

- σ_0 and σ_1 are valid signatures by the owners of τ_0 and τ_1 respectively over tuple $(\text{TxID}, in_0, in_1, out_0, out_1, \Pi_{\text{Sum}})$.

If any of these checks fail, then the chaincode rejects. Otherwise, it produces a read-write set as described in Section 5.2.1, and assembles a settlement transaction with identifier TxID using the produced read-write sets.

After the token chaincode produces the content of the settlement transaction, the token endorsers jointly sign it using the threshold signing algorithm ThresholdSign and their shares tsk.

We stress here that the token chaincode execution does not require reading any state, that is, the token endorsers do not access any information from their local storage, for example, to check whether the inputs are stored in the ledger.

The token endorsers then submit the signed settlement transaction to the ordering service (Step 2, Figure 8) and listen for the transaction confirmation, which indicates its final status, i.e., whether it is valid and has been committed or not. If the endorsers do not receive the confirmation after a timeout, they resubmit the settlement transaction.

Committers. Committers validate the ordered transactions as described in Section 3.2.3. Notice that thanks to the way we encode the read-write sets, attempts to double spend a token will result in a read conflict (i.e., the state of the spent inputs is outdated), invalidating the double-spending transaction.

5.3 Token Systems with Transaction Unlinkability

We recall that token transactions that assure unlinkability hide the *transaction graph* (i.e. the relation between the inputs and the outputs in the ledger).

5.3.1 Transaction Model. Following the privacy-preserving protocol detailed in Section 4.4, we describe the content of client transactions and the corresponding settlement transactions. In the case of an Issue transaction, the client transaction comprises the outputs and the signature of the issuer. The corresponding settlement transaction contains a write-set that includes the issued outputs as ⟨out_key, ser_out⟩.

In the case of a Transfer transaction, each output is a token as usual, and each input is defined as the randomization of a token in the ledger, and the serial number of that token. The transaction also contains the zero-knowledge proofs that show that the transaction is well-formed, and the signatures of the token owners (cf. Figure12).

The associated settlement transaction carries a read-set where for each serial number sn in the client transaction, there is an entry $\langle sn, \text{nil}\rangle$ signaling that, at the time of transaction commitment, entry with key sn should be empty. The transaction also contains a write-set with entries ⟨out_key, ser_out⟩ writing the outputs, and entries $\langle sn, \text{spent}\rangle$ indicating that the serial numbers should be marked as spent, when the transaction is committed. In this way, the validation phase enforces that a token can only be spent once. In fact, two transactions that spend the same token will result in a conflict. Either the first gets committed and the second gets rejected on the ground that one of the entries $\langle sn, \text{nil}\rangle$ in the read-set is no longer empty, or vice-versa.

Client Transaction	
TxID	String
SerialNumbers	[]String
RandInputs	[][]byte
Outputs	[][]byte
ZKProofs	[][]byte
Signatures	[][]byte

Figure 12: Client transaction format for a token system with transaction unlinkability.

We now describe the user wallets and the settlement engine for a token system that guarantees transaction unlinkabiliy.

5.3.2 User Wallets. We first recall that a token τ in such a system is defined as a commitment $\text{Commit}(\text{eid}, v, r, t)$, where eid is the unique identifier of the owner, v is the value, r is the randomness used to compute the serial number of the token, and t is the randomness of the commitment. Furthermore, to be able to spend τ, its owner must have access to the signature γ of the certifiers on τ. The certifiers sign a token only if that token is committed in the ledger.

Accordingly, the wallet securely stores the user's credential $(\text{eid}, \text{sk}, K, \text{cred})$, whereby eid is the unique identifier of the user, sk

```
1: Input
   (TxID, (sn_0, τ'_0), (sn_1, τ'_1), out_0, out_1, Π_Sum, Π_SN, Π_Exist, σ_0, σ_1);
2: Output
   A client transaction ctx
3: ctx ← NewClientTx()
4: ctx.TxID ← TxID
5: for b ∈ {0, 1} do
6:     ctx.SerialNumbers[b] ← sn_b.String()
7:     ctx.RandInputs[b] ← τ'_b.Serialize()
8:     ctx.Outputs[b] ← out_b.Serialize()
9:     ctx.Signatures[b] ← σ_b.Serialize()
10: end for
11: ctx.ZKProofs[0] ← Π_Sum.Serialize()
12: ctx.ZKProofs[1] ← Π_SN.Serialize()
13: ctx.ZKProofs[2] ← Π_Exist.Serialize()
14: return ctx
```

Figure 13: Building a client transaction for token systems with transaction unlinkability.

her secret key, K is her secret key dedicated to computing the serial numbers[12], and cred is the signature from the registration authority on vector $(\text{eid}, \text{sk}, K)$. The user wallet also stores for each token τ, the information that enables its spending. Namely,the wallet stores entries $\langle \tau, (v, r, t, \gamma) \rangle$.

As in Section 5.2.2, we assume that the user instructs the wallet to transfer value v' to payee with identifier eid'. The user wallet assembles the corresponding client transaction as follows:

- Select the first entries $\langle \tau_i, (v_i, r_i, t_i), \gamma_i \rangle$ such that $\sum v_i \geq v'$. Without loss of generality, we assume that the wallet picked entries $\langle \tau_0, (v_0, r_0, t_0), \gamma_0 \rangle$ and $\langle \tau_1, (v_1, r_1, t_1), \gamma_1 \rangle$.
- Create two outputs $out_0 = \text{Commit}(\text{eid}', v', r'_0, t'_0)$ and $out_1 = \text{Commit}(\text{eid}, v_0 + v_1 - v', r'_1, t'_1)$.
- Next, following the steps described in Section 4.4, produce the tuple: $\mu = (in_0, in_1, out_0, out_1, \Pi_{\text{Sum}}, \Pi_{\text{SN}}, \Pi_{\text{Exist}})$, where $in_0 = (sn_0, \tau'_1)$ and $in_1 = (sn_1, \tau'_1)$.
- Compute TxID as the hash of μ.
- Compute two signatures σ_0 and σ_1 on (TxID, μ) as described in Section 4.4.
- Construct the client transaction, as illustrated in Figure 10.
- Communicate to the payee the opening of output out_0 (i.e., $(\text{eid}', v', r'_0, t'_0)$), together with the transaction identifier TxID.
- Finally, submit the client transaction to the *token endorsers* for chaincode execution.

5.3.3 Settlement Engine. When the client transaction first reaches the settlement engine through the token endorsers, it is processed following the validation logic detailed in Section 4.4. If all the validation checks are successful, the token endorsers create (1) read-set entries that indicate that entries with keys sn_0 and sn_1 in the ledger must be empty, (2) write-set entries that write out_0 and out_1, and (3) write-set entries that mark sn_0 and sn_1 as spent, cf. Section 5.3.1. Next, the token endorsers populate the content of the settlement transaction using the produced read-write-sets (cf. Figure 14) and submit the result for ordering. At the committers, the settlement transaction is validated according to the description in Section 3.2.3. Thanks to the way we encode the serial numbers in the read-write sets, attempts to double spend a token will result in a read conflict (i.e., the serial number state is outdated), thwarting hence the double-spending attempt.

We reiterate that to achieve transaction unlinkability, one leverages zero-knowledge proofs of membership and serial numbers. The former allows a payer to show that a token she is spending is in the ledger, whereas the latter ensures that the payer cannot spend the same token more than once. We implement the zero-knowledge proofs of membership using signatures and we introduce **the certifiers**, which are tasked with signing the tokens in the ledger. In more details, each certifier has a secret share tsk' for (t', n')-threshold signature scheme ThresholdSign, where t' indicates that at most t' certifiers can be malicious among the existing n' certifiers. Each certifier runs a dedicated application that is built on top of the committer logic. The application continuously tracks ledger updates, and executes ThresholdSign to produce a threshold signature for the new created outputs (when invoked). The signatures are then stored by the certifiers, to be retrieved later by the user wallets upon request.

Alternatively, the certification functionality can be provided by a dedicated chaincode. Notably, the endorsers of this chaincode correspond to the certifiers, which each leverages the chaincode to (1) confirm the existence of the outputs in the ledger, and (2) jointly compute a threshold signature for the confirmed outputs with other certification endorsers. Notice that in contrast with token endorsers whose response is used to update the ledger state, the certification endorsers treat execution requests as queries (Steps 5 and 6, Fig 8).

5.4 Dispute Resolution – Reconciling Client Transactions

To improve the performance of the overall system, the settlement transaction only includes the information necessary for validation: read-write sets and the signature of the endorsers. This makes the settlement transactions compact and yields bandwidth gains, and therewith, better performances at the orderers. On the downside, this hinders dispute resolution mechanisms, an example of which is holding the token endorsers' accountable for the transactions they endorse.

It is important thus to make client transactions available to the settlement engine (even if asynchronously). To that end, we (1) extend the token endorser application to store the client transaction in a *transaction metadata database*, right after they complete the chaincode execution (Step 1.d, Fig 8); and (2) implement a reconciliation service on top of the committers, which will monitor freshly-committed settlement transactions and query the endorsers for the corresponding client transactions. Since the settlement transaction identifier includes the hash of the client transaction, the reconciliation service can easily verify if the client transaction it is given is the right one.

[12] In some implementations, K is not required to compute the serial number. Instead the randomness r of the token is only known to the recipient and r becomes the serial number.

```
1: Input
   ctx                                   ▷ a client transaction
   tsk                          ▷ a threshold signature secret share
2: Output
   stx                                   ▷ a settlement transaction
3: stx ← NewSettlementTX()
4: stx.TxID ← ctx.TxID
5: stx.NameSpace ← "token"
      ▷ Add the serial numbers as read dependencies, and mark
   them as "Spent"
6: for b ∈ {0, 1} do
7:     RSEntry ← ⟨ctx.SerialNumbers[b], nil⟩
8:     stx.ReadSet ← stx.ReadSet ∪ RSEntry
9:     WSEntry ← ⟨ctx.SerialNumbers[b], "Spent"⟩
10:    stx.WriteSet ← stx.WriteSet ∪ WSEntry
11: end for
                          ▷ Add the created outputs to the write-set
12: for b ∈ {0, 1} do
13:    WSEntry ← ⟨stx.TxID||b, out_b.Serialize()⟩
14:    stx.WriteSet ← stx.WriteSet ∪ WSEntry
15: end for
   ▷ The token endorsers jointly generate a threshold signature
   on stx
16: φ ← ThresholdSign(tsk, stx)
17: stx.Signature ← φ.Serialize()
18: return stx
```

Figure 14: Building a settlement transaction for token systems with transaction unlinkability.

6 EXPERIMENTAL EVALUATION

In this section we evaluate from a performance perspective our system's architecture as described in Section 5. For our evaluation we benchmark and analyze the various phases of transaction settlement in terms of transaction throughput and latency, with respect to the retail CBDC requirements presented in Section 2.3. Our evaluation is extended to the system's performance in cases of malformed settlement transactions or double-spending attacks.

More specifically, in Section 6.1 we evaluate the transaction *execute* phase capturing stateless checks of client transaction validity for different privacy configurations—including any expensive zero-knowledge proof verification—and the generation of the threshold token endorsement (Section 5) signature. As the execute phase scales horizontally, we restrict its performance evaluation to measuring the introduced latency. We proceed in Section 6.2 with the full performance evaluation of the *order* phase with three different consensus algorithms integrated into the HLF ordering service, RAFT [13], SmartBFT [18] and the mentioned (Section 3.2.4) variant of Narwhal and Tusk BFT algorithm [11], ARMA. Finally, in Section 6.3 we evaluate the throughput and latency observed in the optimized HLF committers delivering on the *validate* phase, as well as the certification (Section 4.4) phase that applies solely in the case where privacy-preserving protocols presented in [12] are adopted.

Experimental Setup. For our performance evaluation, we implemented the user wallets, the endorsers, and the certifiers using the Fabric Smart Client[13] and Token SDK[14] libraries. We used the HLF ordering service[15] which we extended to support SmartBFT and ARMA. We implemented the committers as a distributed service in Go[16] using GRPC[17] bidirectional streams for network communication, and yugabyteDB[18] as shard database to persist committed transactions.

Performance metrics (i.e., throughput and 99^{th} percentile latency) are collected via Prometheus[19] with a sample rate of 1 second. During our experiments each reported data point is the average of at least 2 minutes (120 samples) running time after a warm-up phase. Error bars report the standard deviation.

To evaluate various aspects of the settlement engine, we implement a workload generator that can produce synthetic workloads to simulate millions of users. This includes workloads to stress individual components of the settlement engine, such as the HLF ordering service or the enhanced HLF committer.

Finally, we deployed the various components of the settlement engine on IBM Cloud[20], using bare-metal servers in three different regions, namely, London, Paris, and Milan. The servers are equipped with dual 48 core CPUs (Intel(R) Xeon(R) 8260 CPU @ 2.40 GHz), 64 GB RAM, 1 TB SSD (Raid 0), and 10 Gbps network running Ubuntu Linux 20.04 LTS Server.

6.1 Transaction Execution

In this section we analyze the transaction latency observed on the token endorser end when a variety of privacy-preserving token exchange techniques are utilized. Notice that as the operations performed in this phase are horizontally scaleable, throughput can be amortized as per the utilized compute resources.

Recall, that a token endorser routes the transaction received by a client to the token chaincode that i) performs the ZKP verification checks, ii) compiles a read/write-set that corresponds to the considered inputs and outputs, and iii) produces a (threshold) signature by reaching out to and receiving signature shares from the other token endorsers and combines the signature shares into a threshold signature. Table 2 summarizes our results for the latency of the execution of the token chaincode for different privacy configurations, including the ZKP verification checks where applicable and the threshold signature generation by a single endorser employing the techniques described in Section 4. For our experiments we considered the combination of ($t = 3, 5, 10$) endorser shares into one transaction using threshold BLS [22] signing.

Our results show that, even when ZKPs are utilised for anonymity and unlinkability in payment transactions, the *execute* phase does not go above 20 milliseconds, while latency is brought down to a few milliseconds where accountable (Section 1) anonymity techniques are being leveraged. Notice that the impact of threshold signing

[13] https://github.com/hyperledger-labs/fabric-smart-client
[14] https://github.com/hyperledger-labs/fabric-token-sdk
[15] https://github.com/hyperledger/fabric
[16] https://go.dev/doc/devel/release#go1.20
[17] https://grpc.io
[18] https://github.com/yugabyte/yugabyte-db
[19] https://prometheus.io
[20] https://www.ibm.com/cloud

Number of Endorsers	No Privacy	Anonymous	Anonymous & Confidential Exchange	Unlinkable	Signing	Aggregation
$t = 3$	0.3 ±2%	4.2 ±1%	13.6 ±0%	19.7 ±0%	0.147 ±4%	0.124 ±0%
$t = 5$	0.3 ±2%	4.2 ±1%	13.6 ±0%	19.7 ±0%	0.147 ±4%	0.241 ±1%
$t = 10$	0.3 ±2%	4.2 ±1%	13.6 ±0%	19.7 ±0%	0.147 ±4%	0.624 ±1%

Table 2: Transaction execution latency breakdown of various ZKP verifications: No Privacy, Anonymous (Section 4.2), Confidential Exchange (Section 4.3), Unlinkable (Section 4.4), the endorser's threshold signing (Signing) and threshold signature share aggregation (Aggregation).

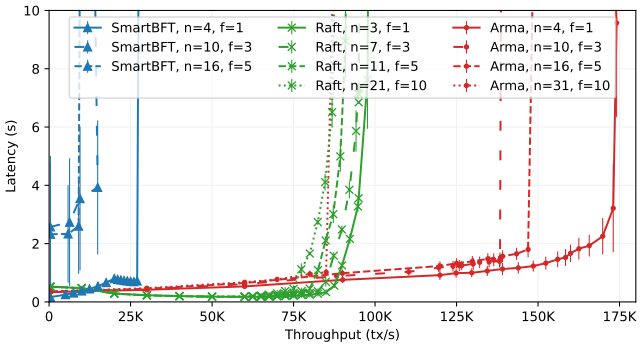

Figure 15: Ordering service throughput-latency for different consensus protocol instantiations, namely, SmartBFT, Raft, and Arma with varying degree of byzantine fault tolerance.

mechanisms is negligible to the overall execute phase latency, even for thresholds that go above a few tens of nodes.

6.2 Transaction Ordering

As mentioned before, we evaluate the settlement engine's ordering efficiency assuming three different consensus algorithms, RAFT [13], SmartBFT [18] and ARMA (variant of [11]).

SmartBFT as well as Raft are integrated in the official open source Hyperledger Fabric release, and slight modifications were made to it to accommodate our customized transaction format. The SmartBFT protocol can be thought of as a non-pipelined version of the seminal PBFT [16] protocol, where each consensus round carries a single batch of transactions. ARMA is similar in its architecture to the Narwhal-HotStuff version of [11], but substitutes HotStuff with SmartBFT [18].

Figure 15 illustrates the throughput and latency of different consensus protocols instantiated with varying number of orderer nodes (supporting up to f node failures), which are deployed across the regions of our testbed. For our experiments, we colocated a workload generator in the same region as the leader node, and used blocks of 3500 settlement transactions, where each transaction contains two serial numbers (SNs) and two outputs. In contrast to classical Fabric transactions, which require a size of approximately 3.5 KB, the compact form of the settlement transaction only requires less than 300 B to perform the same operation, thereby reducing the network load by a factor of 10.

SmartBFT. The maximum throughput we observe is 27,000 tx/s with four orderer nodes ($f = 1$), and up to 9,000 tx/s with 16 nodes ($f = 5$). The latency is round 0.7 seconds with four orderer nodes before saturation, whereas with 10 and 16 nodes the latency goes up to two seconds before saturation. Other works [18] reported comparable numbers of around 2,000 tx/s with four nodes ($f = 1$) and a block size of 1 MB comprising 250 transactions each size of 4 KB. This shows that the transaction size directly impacts the overall throughput of the ordering service as the network load is reduced.

RAFT. In contrast to SmartBFT, we observe much more stable latency with varying number of orderer nodes around 200 ms before saturation. In fact, the latency decreases slightly until saturation. The maximum throughout we observe is 80,000 tx/s with three orderer nodes ($f = 1$) and up to 65,000 tx/s with 21 orderer nodes ($f = 10$). Even though Raft performs better than SmartBFT, recall Raft only supports crash faults. For this reason, we were able to modify the implementation of the Hyperledger Fabric ordering service to authenticate the submitting client already during the TLS handshake and disabled signature validation on the transactions submitted by the client. This modification and the compact settlement transaction size help to improve the overall performance compared to the upstream implementation as reported in [18].

ARMA. We observe the best throughput up to 165,000 tx/s with a latency below 2 seconds with four orderer nodes ($f = 1$). With 21 orderer nodes ($f = 10$) still achieves up to 80,000 tx/s. Interestingly, we observe that the latency increases linear with higher load whereas we had expected a stable latency before saturation. Additionally, it is unclear why Arma with 16 orderer nodes ($f = 5$) saturates before the deployment with 10 orderer nodes ($f = 3$). We did not further investigate this unusual behavior since the current code is considered to be prototype and under active development. Updates to this report will present the new results we obtain.

6.3 Transaction Validation

We evaluated the validation phase of our settlement engine using the prototype implementation of a committer, as outlined in Section 3.2. Our evaluation delved into three parameters impacting the throughput and latency of scalable committer. These parameters are transaction size, invalid signatures, and double spend.

We conducted comprehensive performance benchmarking by deploying the various components of the committer and the workload generator on multiple machines within the same region of our testbed. Unless otherwise specified, we deployed the committer

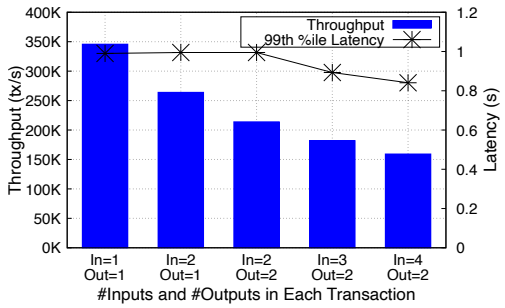

Figure 16: Impact of the number of inputs and outputs in transactions on the throughput and latency of committer.

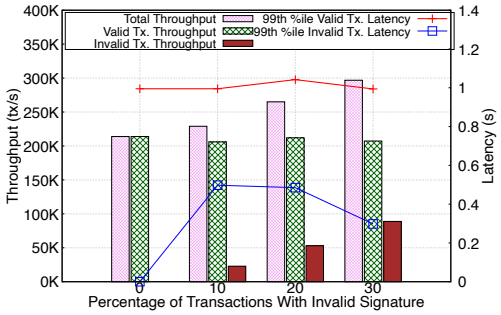

Figure 17: Impact of the percentage of transactions with invalid signature on the throughput and latency of the committer. Each transaction had two inputs and two outputs.

with a total four signature validators, six shards, and one coordinator. We also deployed the workload generator on a separate machine to generate the load. Throughout all our experiments, we consistently kept the latency below 1 second as our benchmark. However, it's crucial to note that when we overload the committer beyond its capacity, a queue begins to form, leading to an increase in latency. Therefore, we carefully generated load in a manner that ensured the latency remained below the 1-second threshold.

6.3.1 Impact of transaction size. To evaluate the impact of transaction size on the scalability of the committer, we conducted experiments involving the submission of transactions with varying numbers of inputs and outputs to the coordinator. The results, as depicted in Figure 16, provided valuable insights. As the number of inputs and outputs in transactions increased, we observed a decrease in transaction throughput, declining from 345,000 tps to 160,000 tps.

The decrease in throughput can be attributed to the growing complexity of the dependency graph maintained by the coordinator. With more inputs and outputs, the number of nodes in the graph increases, leading to longer times for constructing and updating the dependency graph. Additionally, the data structure employed to store the dependency graph is protected by a synchronization primitive, preventing concurrent access. This safeguard, while crucial for data integrity, contributes to the reduction in overall throughput. Thus, we can conclude that the throughput of the committer is limited by the performance of the coordinator service.

To validate this claim, we conducted experiments to understand the performance characteristics of shard servers (Section 3.2.3) with no other services running and signature validators, also in isolation. We found that increasing the number of shard servers or signature validators resulted in a scalable increase in the throughput of the respective components, surpassing 450k tps. Consequently, we can conclude that the bottleneck is not with the shard servers or signature validators but with the coordinator.

6.3.2 Impact of faulty transactions. To analyze the performance of the committer when handling faulty transactions, we conducted two experiments, focusing on the submission of a mix of valid and invalid transactions. Our objective was to investigate how this mix influenced transaction throughput and latency. All transactions included in these experiments had two inputs and two outputs. We varied the percentage of invalid transactions from 0% to 30%. These experiments were conducted with the committer configured with six shards and four signature validators.

Invalid signatures. In the first scenario, we examined the impact of invalid threshold signatures, which occur when a settlement transaction lacks the necessary endorsements or has malformed content. Figure 17 illustrates the changes in throughput and latency as we introduced a varying mix of invalid transactions. Notably, we observed an intriguing trend: the overall throughput increased from 213,000 tps to 296,000 tps as the percentage of invalid transactions rose from 0% to 30%. This increase can be attributed to the sooner rejection of transactions with invalid signatures by the signature validators as opposed to valid transaction. Consequently, the coordinator did not consider these transactions for inclusion in the dependency graph. As a result, other transactions were processed in their place, leading to the overall throughput boost. However, it's worth noting that the throughput of valid transactions experienced a slight decrease, dropping from 213,000 tps to 207,000 tps over the same range.

Double spendings. In the second scenario involving faulty transactions, we delved into the consequences of double spend incidents. These occurrences take place when a transaction is submitted with inputs that have already been recorded in the ledger, meaning that the input has already been spent. In Figure 18, we can observe the impact of double spending on the throughput and latency of the committer. It became evident that the overall throughput was reduced in comparison to the previous experiment, where we submitted transactions with invalid signatures to the committer.

The decline in the overall throughput can be attributed to the heightened workload imposed on the committer. When a transaction doesn't involve double spending, the read-set freshness check doesn't entail reading any data from the disk since the relevant input doesn't exist in the ledger. Conversely, in cases where a double spend occurs, the input is indeed present in the ledger, necessitating the verifier to read specific bytes from the disk. Consequently, this additional workload causes a reduction in the overall throughput, amounting to an 27% decrease.

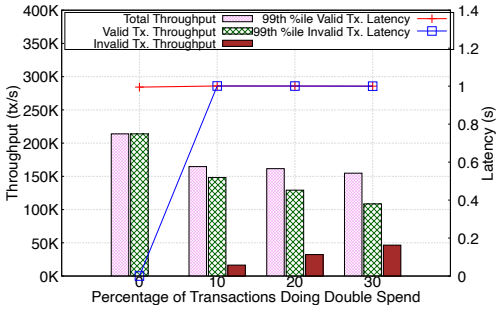

Figure 18: Impact of the percentage of transactions doing double spend on the throughput and latency of the committer. Each transaction had two inputs and two outputs.

6.4 Transaction Certification

As mentioned earlier, transaction certifiers confirm the existence of a token in the ledger by collaboratively generating a threshold signature on the token. In our prototype, a user sends to the certifiers a certification request, which identifies a token in the ledger and carries some auxiliary information that allows the certifier to sign the token data (i.e., owner's enrollment id, value, type and serial number) without accessing them. This process is referred to in the literature as *blind signature*. Each certifier then checks if the token exists in the ledger and *blindly threshold signs* it. The user receives the blind signature shares, un-blinds and combines them to get the certifiers' signatures.

We evaluate the transaction certification as described in Section 4.4. We focus on the execution throughput and latency of the blind signature, which in the prototype corresponds to Pointcheval-Sanders' [30, 31]. We follow its threshold variation described in [12, 32]. The implementation we use is the open source Pointcheval-Sanders (PS) implementation of [33].

Using a single certifier node, we observed a maximum throughput up to 13,300 certifications per seconds with stable latency around 7.69 milliseconds. The overall throughput of the certifier scales horizontally with the number of available compute resources.

7 CONCLUSION

In this paper, we described a distributed and privacy-preserving transaction processing framework tailored for tokenized assets, and suited, in particular, for retail CBDC. We evaluated the framework using various consensus mechanisms, and a variety of literature asset exchange protocols exhibiting different degrees of privacy. Our results show that for the standard UTXO pseudonymity model, our prototype implementation can process up to 80,000 TPS in the case of Raft and SmartBFT and more than 150,000 TPS in the case of emerging consensus algorithms. Our results further demonstrate the horizontal scalability of transaction processing compute. In fact, we show that the same numbers *can be accomplished* in stronger privacy scenarios where the exchanged amounts are concealed, and / or activity of individual users concealed at the cost of more powerful equipment. These results demonstrate that BFT protocols and zero-knowledge proofs can be leveraged to implement strong resilience and flexible privacy guarantees in CBDC systems without sacrificing performance.

REFERENCES

[1] Gaining momentum – results of the 2021 bis survey on central bank digital currencies. https://www.bis.org/publ/bppdf/bispap125.pdf.

[2] https://cbdctracker.org/.

[3] European Central Bank EuroSystem. Ecb welcomes european commission legislative proposals on digital euro and cash, 2023. https://www.ecb.europa.eu/press/pr/date/2023/html/ecb.pr230628~e76738d851.en.html#:~:text=The%20European%20Commission%20has%20published,as%20a%20means%20of%20payment.

[4] European central bank mission. https://www.ecb.europa.eu/ecb/orga/escb/ecb-mission/html/index.en.html.

[5] Second payment services directive. https://edpb.europa.eu/sites/default/files/files/file1/edpb_guidelines_202006_psd2_afterpublicconsultation_en.pdf.

[6] The people's republic of china's digital yuan: Its environment, design, and implications. https://www.adb.org/sites/default/files/publication/772316/adb-wp1306.pdf.

[7] James Lovejoy, Madars Virza, Cory Fields, Kevin Karwaski, Anders Brownworth, and Neha Narula. Hamilton: A High-Performance transaction processor for central bank digital currencies. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 901–915, Boston, MA, April 2023. USENIX Association.

[8] Wholesale central bank digital currency experiments with the banque de france. https://www.banque-france.fr/sites/default/files/media/2021/11/09/821338_rapport_mnbc-04.pdf.

[9] Alin Tomescu, Adithya Bhat, Benny Applebaum, Ittai Abraham, Guy Gueta, Benny Pinkas, and Avishay Yanai. Utt: Decentralized ecash with accountable privacy. Cryptology ePrint Archive, Paper 2022/452, 2022. https://eprint.iacr.org/2022/452.

[10] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[11] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 34–50, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Elli Androulaki, Jan Camenisch, Angelo De Caro, Maria Dubovitskaya, Kaoutar Elkhiyaoui, and Björn Tackmann. Privacy-preserving auditable token payments in a permissioned blockchain system. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, AFT '20, pages 255–267, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[14] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[15] Publications on central bank digital currencies (cbdc). https://www.ecb.europa.eu/home/search/html/central_bank_digital_currencies_cbdc.en.html.

[16] Miguel Castro. Practical byzantine fault tolerance. 04 2001.

[17] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[18] Artem Barger, Yacov Manevich, Hagar Meir, and Yoav Tock. A byzantine fault-tolerant consensus library for hyperledger fabric. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2021, Sydney, Australia, May 3-6, 2021*, pages 1–9, 05 2021.

[19] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[20] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput bft for blockchains, 06 2019.

[21] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 264–276, 2018.

[22] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, sep 2004.

[23] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63, Aug 2001.
[24] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sep 2012.
[25] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, jun 1981.
[26] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM CCS*, pages 21–30. ACM, 2002.
[27] Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and modular anonymous credentials: Definitions and practical constructions. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT*, volume 9453 of *LNCS*, pages 262–288. Springer, 2015.
[28] Andrew Poelstra, Adam Back, Mark Friedenbach, Gregory Maxwell, and Pieter Wuille. Confidential assets. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 43–63, Berlin, Heidelberg, 2019. Springer Berlin Heidelberg.
[29] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474. IEEE Computer Society, 2014.
[30] David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016*, pages 111–126, Cham, 2016. Springer International Publishing.
[31] David Pointcheval and Olivier Sanders. Reassessing security of randomizable signatures. In Nigel P. Smart, editor, *Topics in Cryptology – CT-RSA 2018*, pages 319–338, Cham, 2018. Springer International Publishing.
[32] Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, and George Danezis. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. *ArXiv*, abs/1802.07344, 2018.
[33] Threshold signature scheme library. https://github.com/IBM/TSS/.