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

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

Abstract—**Central bank digital currencies refer to the digitization of central bank money in a way that satisfies the classical requirements of privacy and regulation compliance, in addition to *first-of-a-kind* requirements of transparency, interoperability, and resilience that extends beyond failures.**

In this paper, we introduce a novel framework for central-bank digital currency settlement that yields a trusted system of record with high performance and resilience against partial compromise. This system of record acts as a truth anchor serving the purposes of interoperation, dispute resolution, and fraud detection. Our framework is also agnostic to the transaction validation logic running on top, in particular, to the cryptographic protocols used to meet the objectives of privacy, compliance and transparency.

We evaluated the performance of our framework using an enhanced version of Hyperledger Fabric with Byzantine-fault tolerance, and observed throughput that is comparable to crash-fault tolerant systems. Our results show how transaction validation scales horizontally and how a throughput of more than $100,000$ transactions per second can be achieved even with computation-heavy privacy-preserving protocols.

1. Introduction

In recent years, Central Bank Digital Currency (CBDC for short) 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, drastically decrease counter-party risk, and speed up cross-border payments. Retail CBDC is expected to reduce transaction fees, break the existing monopoly of today's payment service providers, enhance financial inclusion, and trigger innovation in digital payments. In light of these advantages, more than 130 central banks started exploring CBDC and publishing periodic reports on the requirements and evolving architectural considerations of a CBDC system [1], [2], together with the outcomes of their experimentations. Moreover, a handful of central banks have launched CBDC pilots [2], while the European central bank has launched a legislation proposal for the adoption of the *digital euro* [3].

While CBDC systems will be regulated differently depending on the jurisdiction, they all agree on the following aspects. First of all, the critical impact of CBDC infrastructure on the money supply mandates that its governance be in the hands of the central bank only (i.e., centralized). Despite the centralized governance of CBDC systems, they are nonetheless required to operate in a highly *robust and resilient* manner. That is, to function properly even if parts of the system crash or get compromised due to a successful cyber or insider attacks. Given the critical nature of CBDC, this is a reasonable requirement that imposes a distributed if not a decentralized deployment of the system.

In addition, there is the need for *interoperation* and *programmability*. In terms of interoperation, a CBDC system should interoperate with existing payment and settlement infrastructure, other CBDC systems, and the emerging digital asset landscape. In terms of programmability, the system should offer enough flexibility to add new capabilities as the needs of the financial ecosystem evolves. Programmability is to be understood as a stepping stone to "*foster innovation in payments*" within CBDC systems.

Then there is regulatory compliance, which requires *efficient dispute resolution*, *fraud detection* and *auditability*. For example, Anti Money Laundering (AML) and Combating the Financing of Terrorism (CFT) regulations stipulate that suspicious payment transactions be detected, attributed to their origin and reported to the relevant authorities, while PSD2 [4] emphasizes the importance of fraud detection and dispute resolution, which in turn, further accentuate the importance of *accountability*. Accountability ensures that the various CBDC participants cannot repudiate their actions. This includes CBDC consumers, which should not be able to repudiate their payments, and CBDC providers, which should not be able to repudiate their decisions. Furthermore, dispute resolution and interoperation also call for *transparency* regarding both decision making and transaction processing. Transparency is optimally served by a (trusted) *ledger of (processed) transactions* that acts as a single point of reference and truth.

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 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 accommodate different privacy levels. Note that as CBDC technology evolves, so will privacy regulations and requirements, and agility should be built in. However, guaranteeing privacy should not be at the detriment of the other requirements, for example, accountability and fraud detection.

Finally, retail CBDC systems should be able to compete with existing payment solutions and accommodate transactions of millions of users. This translates into processing tens of thousands of transactions per second at peak time.

Prominent efforts to address the CBDC requirements [5], [6] 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. These solutions assume that the entities processing transactions will never arbitrarily misbehave, i.e., they are crash tolerant but not fault tolerant. Given the critical nature of CBDC and the potential geographic distribution of its transaction processing, its correct operation in the event of compromise becomes mandatory. Notice that often data centres located in different geographic areas belong to different trust domains.

To circumvent single points of control, financial institutions [7] and the research community (e.g., [8]) have been exploring decentralized consensus-based transaction processing systems such as distributed ledger technology (DLT). In fact, DLTs can facilitate building a trusted system of record, and supporting programmability. Unfortunately, current DLT implementations penalize throughput. Work-arounds based on off-chain exchanges, which are only occasionally anchored on-chain, have become popular in DLT systems (a.k.a *layer-2 solutions*), yet, they fall short in the context of CBDC, as they lack *fast* finality and full transparency for all payment transactions.

In this paper, we introduce a transaction processing framework for (fungible) financial assets, and more specifically, CBDCs, that addresses the aforementioned requirements. 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 overhead compared to centralized systems 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 system-wide, transaction ledger called *system ledger*.
- *Resilience to compromised nodes* by extending Hyperledger Fabric DLT to support byzantine fault tolerance in all phases of transaction processing.
- *Pluggable transaction format and transaction security checks* that decouple application-related logic from the DLT layer that builds and maintains the system ledger. This is a byproduct of DLT smart contracts that allow easy integration of different privacy mechanisms and functionalities into the system.
- *Throughput and latency comparable to the throughput and latency observed in today's (ledger-enabled) centralized solutions* (e.g., [6]). This is accomplished thanks to (i) the execute-order-validate transaction processing model introduced by Hyperledger Fabric 1.0 [9], (ii) a highly-performant *byzantine fault tolerant consensus* protocol (similar to [10]) for order, and (iii) leveraging 2-phase commit principles.
- *Horizontal scalability of all application-specific logic* introduced in transaction processing. This is particularly important for applications that employ zero-knowledge proofs to offer privacy.

We provide a prototype implementation of our framework as an *evolved version* of Hyperledger Fabric [9], coupled with four transactional privacy models: (i) UTXO support using standard PKI and no privacy; (ii) UTXO support with accountable pseudonymity/anonymity, (iii) UTXO support with anonymity and exchanged amount confidentiality, and (ii) unlinkable UTXO utilizing the cryptographic mechanisms in [11] for full privacy (with accountability). We further evaluate our system's performance using three consensus protocols: (i) a crash fault tolerant consensus protocol, i.e., Raft [12], (ii) a byzantine fault tolerant consensus protocol in the wild, i.e., BFTSmart [13] and (iii) a new byzantine-fault tolerant architecture inspired by [10] with state of the art performance and scalability.

Our results show that for the standard UTXO pseudonymity model, our prototype can process up to $80,000$ transactions per second (TPS) in the case of Raft and BFTSmart and more than $150,000$ TPS in the case of the new BFT consensus. Our results further show that the same numbers *can be reached* with stronger privacy guarantees (unlinkability) if more powerful equipment is available.

Layout. The paper is organized as follows. Section 2 describes the system entities, its trust assumptions and requirements. In Section 3, we extend Hyperledger Fabric to meet the demands of the (retail) CBDC use-case. Section 4 briefly presents UTXO-based token systems and their various privacy models, whereas Section 5 describes our framework that builds on our extension to Hyperledger Fabric and the UTXO model. Section 6 benchmarks our system's performance and Section 7 covers the related work. Finally, Section 8 concludes the paper.

2. System Overview

A CBDC system involves a **central bank**, which decides the monetary policy, manages the overall liquidity, and validates CBDC operations via a **settlement engine**. It also involves **users**, who hold CBDC and exchange it for goods and services. Each user is equipped with a *digital wallet* to store her secret key, track her CBDC holdings and authorize CBDC operations. Intermediated CBDC comprises, in addition, *intermediaries*, which correspond to the commercial banks, usually tasked with Know-Your-Customer (KYC) checks and the management of user accounts, both deposits and CBDC.

The central bank issues new CBDC upon *withdrawal requests* from the users[1]. Now equipped with CBDC, the users issue payment requests, which are, for compliance purposes, subjected to AML and CFT checks. The checks detect suspicious operations and hold the originators accountable. Each user is consequently provided with a *long-term identity* that binds her physical identity to a public key. The corresponding secret key is stored in the user's wallet to authenticate and authorize subsequent CBDC operations. This process is facilitated by a trusted **registration authority** that guarantees the correctness of the mapping between the physical identity and the digital identity.

If privacy against the central bank is desired, then access to payment data should be restricted. In particular, the central bank and the settlement engine should validate CBDC payments without learning their value or the identities of the parties involved. This complicates AML and CFT checks whose accuracy hinges upon the access to payment information. To meet both the privacy and the compliance requirements, we introduce **auditors**, which are entities independent of the central bank, that are authorized to inspect withdrawal and payment requests and judge their compliance based on historic data. An example of such auditors are the intermediaries.

For simplicity, we restrict the description to *non-intermediated* CBDC, in which users submit their requests directly to the central bank.

2.1. System Participants

Now we describe the participants of a typical CBDC system. The **registration authority** produces and registers the credentials of the users. During user onboarding, the authority first runs KYC checks on the user to validate her physical identity. If the checks are successful, then it verifies if the user knows the secret key sk underlying the advertised public key pk. The authority then maps the physical identity to a *unique enrollment identifier* eid, and produces a credential cred that binds eid to pk, and returns to the user the pair $(\mathsf{eid}, \mathsf{cred})$. Note that the registration authority can be operated by the central bank with the help of the commercial banks to perform KYC.

The **users** hold CBDC and issue withdrawal and payment requests. A user maintains a wallet that stores her secret key sk matching her credential cred, tracks her holdings, and authorize CBDC operations.

The **central bank** issues new CBDC following valid withdrawal requests and runs the settlement engine.

1. The withdrawal request is submitted to the central bank via the intermediary in intermediated CBDC.

The **settlement engine** validates the issuance of new CBDC and settles user payments against the global state maintained in a *ledger* and pre-defined system rules (e.g., issuance is only valid if initiated by the central bank). Each valid issuance and payment result in updates in the global state. Given the criticality of this component, we recommend its distribution/decentralization to withstand failures and attacks. In this paper, we base its implementation on an extension of Hyperledger Fabric [9], on which we further expand in Sections 3 and 5.

Auditors monitor user transactions and determine their compliance. We say that an auditor is *active* if it inspects CBDC operations prior to their validation by the settlement engine, and *passive* otherwise.

2.2. Trust Assumptions

The **central bank** is trusted to issue CBDC only upon a valid withdrawal request, and in accordance with its monetary policies. Trust in the central bank can be relaxed through its *distribution*. For example, the signing key that authorizes central bank operations, can be shared among relevant and independent stakeholders, which now must reach agreement on central bank operations before executing them. **Users**, on the other hand, are not trusted, as they are incentivized to increase their holdings unlawfully and/or spend CBDC they do not own. We trust the **registration authority** to assign each user a unique enrollment identifier and a unique credential; this trust though can be distributed. Finally, the **settlement engine** is distributed and trusted as a whole, in that it will operate correctly in the presence of a dishonest minority. We consider two failure models for the participants of the settlement engine: *Crash* and *Byzantine*. The former implies that the dishonest minority may only crash and not respond, whereas the latter entails that this minority may behave *arbitrarily*.

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 information, whereas the **auditors** are trusted not to share the information they learn with third parties, unless required by the law. However, all participants (excluding the auditors) may collude to undermine the privacy of honest users.

2.3. Requirements

Security. A CBDC system is said to be secure if it meets the following requirements:
Transaction Authorization. The settlement engine accepts a CBDC operation only if it originates from lawful parties: the owner of the funds being transferred in case of payments, and the authorized issuer(s) in case of issuance.
Balance Preservation. A payment should preserve the total amount of CBDC in circulation. In other words, the holdings of the payee must be increased by the same amount by which the holdings of the payer are decreased.
Resilience. The CBDC system must correctly operate and provide expected services even in the face of errors, failures,

and cyber-attacks. Resilience often requires redundancy, replication, backup and recovery strategies, load balancing, and automated failover mechanisms. It can also leverage *distribution of trust* to reduce infrastructure cost.

Privacy. The privacy requirements of CBDC vary depending on the jurisdiction, but are usually covered by the following:
Anonymity. A CBDC operation is anonymous if the identities of the involved parties are unknown to all others. As we only consider CBDC systems with one central bank, the identity of the central bank need not be anonymized during issuance.
Confidentiality. This refers to the confidentiality of the content of the CBDC operations (i.e., currency/value) against all participants except the transacting parties.
Unlinkability. Unlinkability ensures that, excluding the auditors, no one can tell if two CBDC operations involve the same user, either as an issue beneficiary, a payer or a payee.

Note that anonymity does not imply confidentiality and vice versa, whereas unlinkability implies both.

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. In the presence of a single auditor, the latter inspects the content of all transactions. In multi-auditor settings, audit policies will define which transactions an auditor can access.

Performance. To accommodate retail payments CBDC should enjoy the same (or better) performances as existing payment solutions. According to various ECB reports [14], a retail CBDC system must serve a few tens of thousands transactions per second (TPS) in the first years of its operation, and gradually scale to $150,000$ TPS, whereas latency must vary between 3 and 5 seconds, not to impact the user experience at points of sale. Note that wholesale CBDC is characterized by more flexible performance requirements.

2.4. Interactions Overview

A user first engages in an *on-boarding* process with the registration authority. A successful onboarding sees the user assigned a unique enrollment identifier eid and a credential cred binding eid to the user's public key pk. The corresponding secret key sk is stored in the user's wallet to authorize and authenticate future CBDC operations.

A user receives CBDC by instructing her wallet to send a withdrawal request to the central bank. In turn, if the central bank approves the request, it submits an issuance request to the settlement engine to finalize the creation of the new CBDC. If the settlement engine accepts the issuance request, the user's wallet updates the user's holdings accordingly. The user can also receive CBDC as a result of a CBDC payment, where the user is the *payee*. In this case, the *payer* instructs their wallet with the payment information (i.e., value, payee) and the wallet prepares and sends the payment request to the settlement engine for validation. If

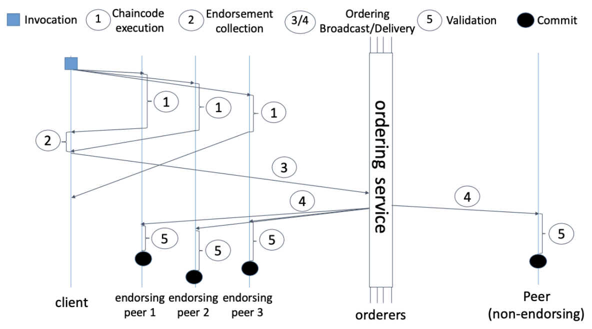


Figure 1: Transaction Lifecycle in Hyperledger Fabric.

the settlement engine accepts the payment, then the wallets of the payer and the payee are updated correspondingly.

In the case of active audit, the user wallets seek additional approval from the auditors before submitting the CBDC requests. In the case of passive audit, no additional approvals are needed.

3. Extending Hyperledger Fabric

Hyperledger Fabric (HLF) is a permissioned Distributed Ledger Technology (DLT) system emphasizing robust identity management and built-in mechanisms for non-repudiation and accountability. It introduces an innovative *execute-order-validate* [9] model for transaction processing, departing from the traditional *order-execute* paradigm. In the *execute-order-validate* model, transactions can be *speculatively* executed before ordering. Stale data during speculative execution leads to rejection during the validate phase. This model enables parallel execution for higher throughput and allows developers to write the execute phase code in their language of choice, providing flexibility.

Figure 1 depicts HLF's transaction flow involving **chaincodes**, smart contracts in traditional languages, executed during the *execute phase*. Each chaincode has **endorsers**, a subset of network participants tasked with execution. Chaincodes are assigned a *namespace* indicating the ledger partition they can update, governed by an *endorsement policy* for validation. A **client** triggers chaincode execution by proposing a transaction to endorsers. Endorsers speculatively execute, producing a *read-set* (dependencies) and a *write-set* (changes). These form the *read-write set*, signed and sent to the client, whose submission to orderers includes endorser signatures. **Orderers**, authorized network participants, manage the *order phase*, agreeing on transaction order, batching into blocks, and delivering to **committers**. **Committers**, a subset of nodes, validate transactions against ledger state and chaincode endorsement policies. Valid transactions update the ledger, and in HLF, endorsers also serve as committers, known as **Fabric peers**.

3.1. HLF Relevance and Limitations for CBDC

The HLF architecture provides a foundation for CBDC settlement due to the following. A chaincode can handle

CBDC issuance/transfer and security/validity checks in the execute phase, with horizontally scalable execution. Scalability allows independent addition of nodes for load balancing. Avoiding single control points is achieved through an endorsement policy, requiring sign-offs from multiple parties. Chaincodes are adaptable to evolving CBDC ecosystems, enabling transparent upgrades based on a specified chaincode administration policy (multi or single-party).

Transaction ordering, essential for dispute resolution, is facilitated by the adaptable ordering service in HLF. The modular architecture allows easy adjustments to the system's threat model and scalability requirements. For instance, switching between Crash Fault Tolerant (CFT) and Byzantine Fault Tolerant (BFT) protocols can be done seamlessly without altering other system components.

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.

Unfortunately, the current HLF implementation fails to meet performance and scalability requirements of CBDC systems for the following reasons. The endorser and committer roles are tightly coupled within each peer node, which results in performance limitations. The concurrent execution of transaction endorsement and commitment on a single node leads to a competition for CPU resources, where the performance is constrained by the available cores. Additionally, the committer's sequential validation of transactions and the subsequent commit process introduce overhead due to non-pipelined execution. Further constraints emerge from the disk write bandwidth on a single node, impacting the committer's performance. The exclusive access requirement to the ledger state intensifies the competition between the endorser and committer, as transaction execution and state access/writing operations are mutually exclusive, driven by limited concurrency control in the system's state database.

The endorser functionality has limited flexibility. Transactions with multiple signatures create a complex format and storage overhead, impacting the ordering service performance due to consensus protocol sensitivity to communication bandwidth. The default HLF endorser enforces consistent logic across endorsers, limiting the use of chaincode and endorser-specific secrets. Departing from this default allows endorsers to use application-specific secrets, communicate directly, and submit transactions to orderers independently.

HLF utilizes a consensus protocol in the *order* phase for transaction ledger output. Traditional byzantine fault tolerant consensus protocols are designed to resist arbitrary behavior in a subset of participants. However, they exhibit low transaction throughput with an increasing number of nodes. Leader-based consensus protocols, such as [15], [16], [17], [18], face saturation of the primary's egress bandwidth due to block broadcasting.

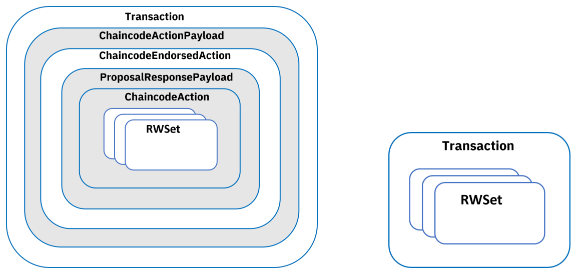


Figure 2: Transaction format in current and new version of Hyperledger Fabric: (a) HLF nested transaction format: Gray indicates a protobuf message encoded as opaque bytes; (b) New transaction format: Each transaction contains several namespaces, where each namespace can contain reads, writes or read-writes.

3.2. Optimising HLF for CBDC applications

As current instances of HLF [19] do not meet the performance and message flow needs of retail CBDC, we enhanced HLF in four ways as described in this section.

Simplifying Transaction Format. The current HLF transaction employs a nested data schema using serialized protobuf messages (refer to Figure 2a), allowing for extensibility but incurring computational overhead. To address this, we introduce a flat transaction format (refer to Figure 2b), eliminating nested deserialization, enabling direct access to transaction elements, and reducing the overall transaction size. The new format includes a unique identifier, endorsements' signatures, and multiple read-write sets organized by namespace. Each namespace corresponds to a ledger partition for reading and/or updating during validation. The read sets define entries with keys and version numbers for validation, while successful commits persist key-value pairs present in write-set to the corresponding namespace.

Reworking the HLF Endorser. Figure 4 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 that receives transaction proposals from clients and dispatches them to the right chaincode for execution.

The new endorser extendeds the original HLF endorser with the following features. *Peer-to-peer communication* to accommodate interactions between endorsers when applications 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

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)	✓

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

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 [20], ECDSA [21], and EdDSA [22].

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 4. 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 a two-phase commit protocol [23] between them to perform distributed validation and commitment 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 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.

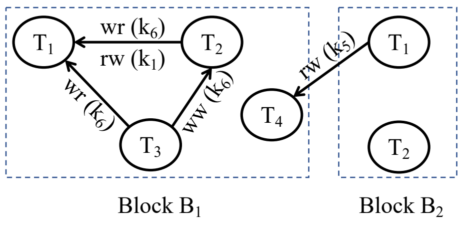


Figure 3: Dependency graph of transactions in Table 1.

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 3. 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.

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

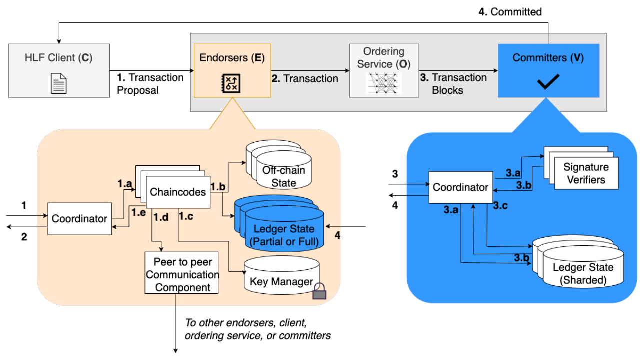


Figure 4: Architecture and transaction flow of Hyperledger Fabric with the new endorser and committer. (1) Client submits a transaction proposal to the endorsers of the chaincode to be invoked; the endorsers trigger the corresponding chaincode's execution (1a), by accessing an off-chain state database (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).

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 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.

Reworking HLF ordering service. To order transactions, we leverage an improved version of the Narwhal and Tusk BFT consensus [10], called ARMA. ARMA (as Narwhal and Tusk) replicates transaction batches in parallel and uses a BFT protocol to order 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 [10]), overcomes the network bandwidth bottleneck described in Section 3.1. Further distinguishing itself from closely-related work like [10], 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), and defined as a triple $(\mathrm{owner}, \mathrm{type}, \mathrm{value})$. In systems with a single asset type, a token is a pair $(\mathrm{owner}, \mathrm{value})$. For simplicity, we restrict the paper to token systems with a single type.

The state of a token is maintained in a ledger, which can be updated using two operations. An Issue operation creates new tokens in the system and a $\mathrm{Transfer}$ operation changes the owner of a token. The ledger receives these operations as *transactions*, and we conflate the transaction with the operation it carries, for e.g., a transaction with an Issue operation is called an Issue transaction. In this paper, these transactions follow the Unspent Transaction Output (UTXO) model, as it offers better privacy and concurrency control [6] than the account model.

4.1. UTXO-based Token Transactions

A UTXO-based transaction is defined by a set of inputs and a set of outputs. The inputs are references to tokens in the ledger, whereas the outputs are new tokens to be created. Valid UTXO transactions result in deleting the inputs, and adding to the ledger the outputs, now ready to be used in subsequent transactions.

Assume that user A wishes to transfer a token of value v to user B. To that end, A builds a $\mathrm{Transfer}$ transaction as follows: (1) she selects from her tokens a subset $(\tau_0, ..., \tau_{n-1})$ whose sum value $v^* \geq v$; (2) defines the inputs of the transaction as $\mathrm{key}_0, ..., \mathrm{key}_{n-1}$, where key_i is the key 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 $v^* - v$; (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 $\mathrm{Transfer}$ transaction must be signed by the owners of its inputs. Considering the previous example, user A signs the $\mathrm{Transfer}$ transaction prior to submitting it for validation, during which the Transfer transaction is subjected to the following *validity checks*:

1) The inputs of the transaction already exist in the ledger. This checks for **double spending**.
2) The sum value of the inputs equals the sum value of the outputs. This checks for **balance preservation**.
3) The rightful owners of the inputs signed the transaction. This checks **authorization**.

If all checks succeed, then the transaction is accepted and the state of the ledger is updated. Namely, the inputs of the transaction are deleted, and each transaction output is assigned a unique key in the ledger. This key is usually computed as a function of the unique identifier of the transaction and the index of the output in the transaction.

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 verifies if it was signed by one of the parties authorized to create the tokens. If the transaction is accepted, then its outputs are added to the ledger.

4.2. Privacy-preserving Token Systems

UTXO-based token systems can accommodate varying degrees of transaction privacy: starting from systems that only achieve user anonymity, to systems that also ensure transaction confidentiality, concluding with systems that guarantee transaction unlinkability. We recall that 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). Both transaction confidentiality and unlinkability are achieved using *zero-knowledge proofs*, whereas transaction anonymity can be guaranteed using one-time keys, however, for accountability purposes, *anonymous credentials* [24] are preferred. These allow users to sign transaction using their long-term identities without leaking any information about who they are. Due to space limitations, we defer the detailed description of these token systems to Appendix B.

4.3. Audit in Token Systems

Financial applications often require audit capabilities, and token systems are no exception. Token systems without privacy protections support audit by default: 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 contains partially or fully obfuscated transactional information. Audit in such a setting is facilitated by one of two approaches:

Active Audit: The initiator of a transaction, be it an Issue or Transfer, provides the information of the transaction to the auditors in the clear. The auditors then 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 it.

Passive Audit: A transaction in this approach contains ciphertexts that can be decrypted only by the authorized auditors. To guarantee that the auditors successfully retrieve the transactional information without external help, the ciphertexts carry additional proofs that confirm that they encrypt the correct information.

We omit audit functionalities in what follows, however, we note that the transaction model and the architecture we introduce next can be easily enhanced with audit.

5. Scalable Architecture for CBDC

Our CBDC architecture involves two main components: the *user wallets* and the *settlement engine*. A user wallet could be held by an issuer, a payer or a payee, and is tasked with translating the instructions of its holder into transactions, which are validated by the settlement engine. The latter builds on HLF in the following way:

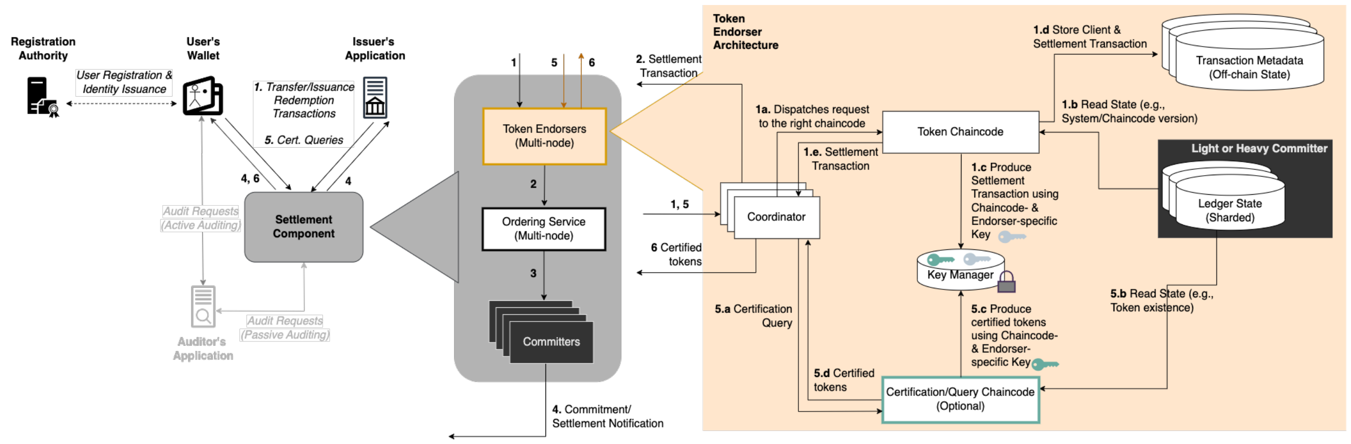
A chaincode, that we call the *token chaincode*, implements the validity checks as described in Section 4.1. The token chaincode verifies the signature(s) of the issuers and/or the token owners, and the zero-knowledge proofs whenever transaction privacy is supported. These checks are performed with minimal access to the ledger state: the chaincode only reads the chaincode version or the system configuration (e.g., identity rules), which seldom change.

Upon receiving a 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 transaction, including the most recent versions of the system configuration; and (2) the write-sets reflect the state updates that the transaction applies to the ledger if committed. The token endorsers then sign the transaction's execution results using their (threshold) signing key. They can additionally be configured to deliver the endorsed transaction to the ordering service on behalf of the user wallets, playing hence the role of an HLF client.

The *ordering service* totally orders the endorsed transactions and batches 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 latter accordingly. It is 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



Registration Authority
User's Wallet
Issuer's Application
User Registration & Identity Issuance
1. Transfer/Issuance Redemption Transactions
5. Cert. Queries
4, 6
4
Audit Requests (Active Auditing)
Settlement Component
Auditor's Application
Audit Requests (Passive Auditing)
1
5
6
Token Endorsers (Multi-node)
2
Ordering Service (Multi-node)
3
Committers
4. Commitment/ Settlement Notification
2. Settlement Transaction
1, 5
6 Certified tokens
Token Endorser Architecture
1a. Dispatches request to the right chaincode
1.e. Settlement Transaction
Coordinator
Token Chaincode
1.d Store Client & Settlement Transaction
Transaction Metadata (Off-chain State)
1.b Read State (e.g., System/Chaincode version)
Light or Heavy Committer
Ledger State (Sharded)
1.c Produce Settlement Transaction using Chaincode- & Endorser-specific Key
Key Manager
5.a Certification Query
5.b Read State (e.g., Token existence)
5.c Produce certified tokens using Chaincode- & Endorser-specific Key
5.d Certified tokens
Certification/Query Chaincode (Optional)

Figure 5: Scalable CBDC Architecture. (1) The wallet submits a client transaction to the settlement engine; (2) the token endorsers, through a front-end component called the coordinator, receive the client transaction (1.a) and execute the token chaincode, which occasionally accesses the ledger state (1.b); the token endorsers then produce and sign the settlement transaction (1.c), store the client transaction in a dedicated database referred to a transaction metadata database (1.d), and finally submit via the coordinator the signed settlement transaction to the ordering service (1.e & 2); the ordering service delivers the transaction in an ordered block to the committers for validation (3); if the settlement transaction is valid, the committers apply the corresponding state updates, and send back the final status of the transaction (committed or not) to the transaction submitter (4). When transaction unlinkability is supported, users request the certification of their tokens (5) by invoking the query/certification chaincode (5.a); if the tokens exist in the ledger (5.b), the chaincode's endorsers certify the requested tokens (5.c) and return the result (5.d & 6).

of the token chaincode, or in a dedicated chaincode with its own endorsement policy.

5.1. Transaction Model and Lifecycle

In our architecture, there are two types of transactions, Issue and $\mathrm{Transfer}$, and they come in two forms: *client transactions* and *settlement transactions*. The client transaction is the message submitted to the settlement engine by a user wallet, and whose content depends on the type of the transaction (Issue vs. $\mathrm{Transfer}$) and the underlying token exchange (e.g., anonymous vs. confidential). According to HLF, the client transaction is the *proposal*, whereas the user wallet is the HLF client. The 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. 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, validate the settlement transaction and commit its write-set if valid. This concludes the lifecycle of a transaction in our architecture.

For clarity, we distinguish between token systems with transaction unlinkability (i.e., full privacy) and those without. Due to space limitations, we only describe here token systems with unlinkability. We defer the description of token systems without unlinkability to Appendix C.

5.2. Token Systems with Transaction Unlinkability

Token systems that assure unlinkability hide the relation between the inputs and the outputs of transactions in the ledger. This is achieved by ensuring transaction anonymity and confidentiality, and enabling payers to spend inputs in a $\mathrm{Transfer}$ transaction without revealing their identifiers in the ledger. The main challenge here is to show that the input of a transaction (1) is actually the output of a previously-committed transaction; and (2) hasn't been spent before, without disclosing its identifier. Similar to [9], we rely on signature-based zero-knowledge proofs of membership to implement (1). More specifically, we introduce the *certifiers*, which are tasked with producing threshold signatures that certify that a token exists in the ledger. These signatures will be used subsequently, to produce the relevant zero-knowledge proofs. To achieve (2), we assign each transaction output a *unique serial number*, which is revealed only at time of spending.

Transaction Model. 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 $\langle \mathrm{out\_key}, \mathrm{ser\_out} \rangle$. The key $\mathrm{out\_key}$ is the *unique identifier* of the output in the ledger, defined as the concatenation of the transaction's unique identifier and the index of the output in the transaction, and $\mathrm{ser\_out}$ is its serialization. In the case of a $\mathrm{Transfer}$ transaction, each output of a client transaction is a token, while each input is the randomization of a token in the ledger, and the serial number of that token. The client transaction also contains the signatures of the token owners and the following zero-knowledge proofs, whose details are in Appendix B.3.

- Π_{Bal} proves that the sum value of the tokens in the inputs equals the sum value of the outputs;
- Π_{Exist} shows that the token in each input is the randomization of a token that was signed by the certifiers;
- Π_{SN} ascertains that the serial numbers in the inputs are computed correctly.

The associated settlement transaction carries a read-set where for each serial number sn in the client transaction, there is an entry $\langle sn, \mathrm{nil} \rangle$ signaling that, at the time of transaction commitment, entry with key sn should be empty. The transaction also contains a write-set with entries $\langle \mathrm{out\_key}, \mathrm{ser\_out} \rangle$ writing the outputs, and entries $\langle sn, \mathrm{spent} \rangle$ indicating that the serial numbers should be marked as spent, when the transaction is committed. This enforces that a token can only be spent once at time of validation. 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 $\langle sn, \mathrm{nil} \rangle$ in the read-set is no longer empty, or vice-versa.

User Wallets. A token τ in a system with transaction unlinkability is defined as a commitment $\mathrm{Commit}(\mathrm{eid}, v, r, t)$, 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 threshold signature γ of the certifiers on τ.

Consequently, the wallet securely stores the user's long-term identity $(\mathrm{eid}, \mathrm{sk}, K, \mathrm{cred})$, whereby eid is the unique identifier of the user, sk her secret key, K is a PRF secret key dedicated to computing the serial numbers, and cred is the signature from the registration authority on vector $(\mathrm{eid}, \mathrm{sk}, K)$. The user wallet also stores for each token τ, the information that enables its spending, i.e., entry $\langle \tau, (v, r, t, \gamma) \rangle$.

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 $\sum v_i \geq v'$. Let $\langle \tau_0, (v_0, r_0, t_0), \gamma_0 \rangle$ and $\langle \tau_1, (v_1, r_1, t_1), \gamma_1 \rangle$ be the selected entries and τ'_0 and τ'_1 their randomization.
- Create two outputs $out_0 = \mathrm{Commit}(\mathrm{eid}', v', r'_0, t'_0)$ and $out_1 = \mathrm{Commit}(\mathrm{eid}, v_0 + v_1 - v', r'_1, t'_1)$.
- Compute $sn_0 = \mathrm{PRF}(K, r_0)$ and $sn_1 = \mathrm{PRF}(K, r_1)$.
- Let $\mu = (in_0, in_1, out_0, out_1, \Pi_{\mathrm{Bal}}, \Pi_{\mathrm{SN}}, \Pi_{\mathrm{Exist}})$, where $in_0 = (sn_0, \tau'_0)$, $in_1 = (sn_1, \tau'_1)$, and $(\Pi_{\mathrm{Bal}}, \Pi_{\mathrm{SN}}, \Pi_{\mathrm{Exist}})$ are the zero-knowledge proofs previously described, and compute TxID as the hash of μ.
- Compute two anonymous signatures σ_0 and σ_1 on (TxID, μ) (note that the signatures can also be computed only on TxID) and construct the client transaction $(\mathrm{TxID}, \mu, \sigma_0, \sigma_1)$.
- Communicate to the payee the opening of output out_0 (i.e., $(\mathrm{eid}', v', r'_0, t'_0)$), together with transaction identifier TxID. This identifier allows the payee to track the status of the transaction in the ledger.
- Finally, submit the client transaction to the settlement engine through token endorsers.

Settlement Engine. In this section, we expand on the components of the settlement engine.

Token Endorser and the Token Chaincode. On receiving the client transaction, the token endorsers verify if the zero-knowledge proofs and the signatures within the transaction are valid. If so, then the token endorsers create (1) read-set entries that indicate that ledger 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 5.2, and assembles a settlement transaction with identifier TxID using the produced read-write sets. Note that the token chaincode execution does not require reading any state to check whether the transaction inputs are stored in the ledger. Instead, it only relies on the content of the client transaction.

After producing the content of the settlement transaction, the token endorsers leverage a *threshold signature* to jointly sign the settlement transaction. By using threshold signatures, we reduce the number of signatures to be verified at the committers. The token endorsers then submit the signed settlement transaction to the ordering service and listen for a confirmation, which indicates whether the transaction has been successfully committed or not. If the endorsers do not receive the confirmation after a timeout, they resubmit the transaction.

Committers. After ordering, the settlement transaction reaches the committers, which validate it according to the description in Section 3.2. Thanks to the way we encode the serial numbers in the read-write sets, transactions attempting to double spend a token will always yield a read conflict, and consequently, be rejected.

Certifiers. Once a settlement transaction is committed, its outputs are ready to be signed by the certifiers, which each runs a dedicated application on top of the committer's logic. The application continuously tracks ledger updates, and produces threshold signatures for the new outputs. 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, whose endorsers correspond to the certifiers. Each certifier leverages the chaincode to (1) confirm the existence of the outputs in the ledger, and (2) compute a threshold signature for the confirmed outputs with the other certifiers. Notice that in contrast with token endorsers whose response is used to update the ledger state, the certifiers treat execution requests as queries.

5.3. 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 therefore, 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; 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.

6. Experimental Evaluation

In this section we evaluate the performance characteristics of our system's architecture as described in Section 5. We benchmark and analyze the various phases of transaction settlement in terms of transaction throughput and latency, with respect to the requirements presented in Section 2.3.

More specifically, in Section 6.1 we evaluate the transaction *execute* phase capturing stateless checks of transaction validity for different privacy configurations. As the execute phase scales horizontally, we focus its performance evaluation on the introduced latency. We proceed in Section 6.2 with the performance evaluation of the *order* phase with three different consensus algorithms. Finally, in Section 6.3 we evaluate the *validate* phase with our enhanced HLF committers, as well as the *certification* phase that applies solely in the case where privacy-preserving protocols presented in [11] are adopted.

Experimental Setup. We implemented the user wallets, the endorsers, and the certifiers using the Fabric Smart Client [25] and Token SDK [26], and extended the HLF ordering service [27] to support the ARMA consensus protocol. We implemented the committers as a distributed service in Go [28] using GRPC [29] for network communication, and yugabyteDB [30] as shard database to persist committed transactions. To evaluate various aspects of the settlement engine, we implemented a workload generator that can produce synthetic workloads to simulate millions of users and stress individual components of the settlement engine.

Performance metrics (i.e., throughput and 99th percentile latency) are collected via Prometheus [31] with a sample rate of one second. Each reported data point is the average of at least two minutes (120 samples) running time after a warm-up phase. Error bars report the standard deviation.

Finally, we deployed the components of the settlement engine on IBM Cloud [32] in three different regions, namely, London, Paris, and Milan. The bare-metal 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 with Ubuntu Linux 20.04 LTS Server.

6.1. Transaction Execution

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

Table 2 summarizes our results for the execution latency of the endorsement 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 using threshold BLS [20] 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 4.2) anonymity techniques are being leveraged. Notice that the impact of threshold signing mechanisms is negligible to the overall execute phase latency, even for thresholds that go above a few tens of nodes.

6.2. Transaction Ordering

Next, we investigate the settlement engine's ordering efficiency using three different consensus algorithms, RAFT [12], SmartBFT [17] and ARMA (variant of [10]).

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 [15] protocol, where each consensus round carries a single batch of transactions. ARMA is similar in its architecture to the Narwhal-HotStuff version of [10], but substitutes HotStuff with SmartBFT [17].

Figure 6 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 nodes ($f = 1$), and up to $9,000$ tx/s with 16 nodes ($f = 5$). The latency is about 0.7 seconds with four nodes before saturation, whereas with 10 and 16 nodes the latency goes up to two seconds before saturation. Other works [17] reported comparable numbers of around

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 B.1), Confidential Exchange (Section B.2), Unlinkable (Section B.3), the endorser's threshold signing (Signing) and signature share aggregation (Aggregation).

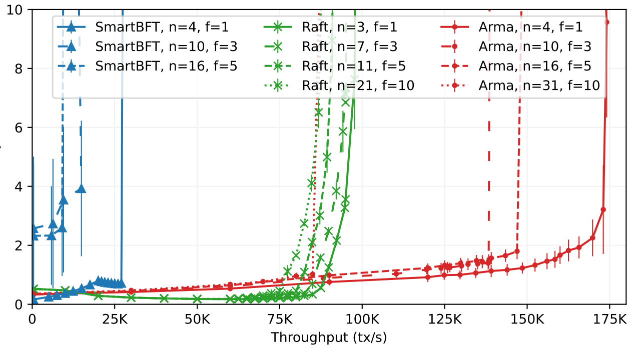


Figure 6: Ordering service throughput-latency for different consensus protocols, namely, SmartBFT, Raft, and ARMA.

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 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 throughput we observe is 80,000 tx/s with three nodes ($f = 1$) and up to 65,000 tx/s with 21 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 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 [17].

ARMA. We observe the best throughput up to 165,000 tx/s with a latency below two seconds with four orderer nodes ($f = 1$). With 21 nodes ($f = 10$), we still achieve 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 nodes ($f = 5$) saturates before the deployment with 10 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

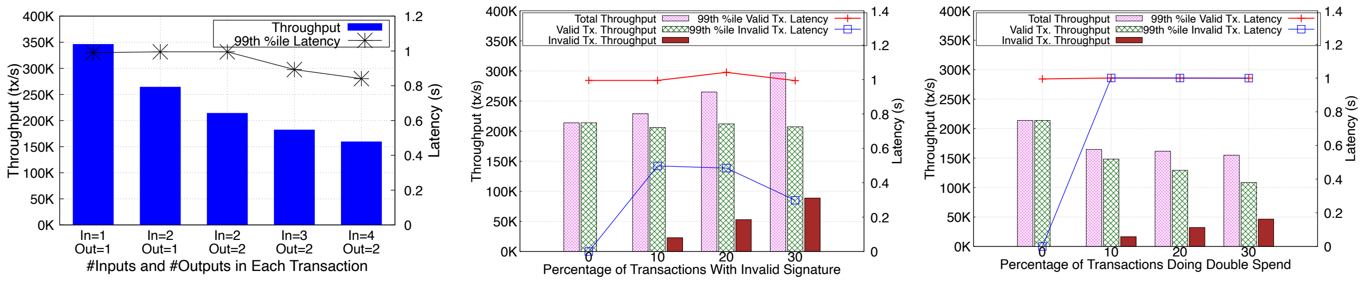
We evaluate the validation phase using a prototype implementation of our enhanced committer, as outlined in Section 3.2. Our evaluation focuses on different paramaters, such as transaction size, invalid signatures, and double spend, impacting the transaction throughput and latency.

We deployed the components of the committer and the workload generator on multiple servers within the same region. Unless otherwise specified, we deployed the committer with four signature validators, six shards, and one coordinator. Throughout all experiments, we shaped the workload to keep the latency below one second. Note that when we overload the committer beyond its capacity, a queue begins to form, leading to an increase in latency.

Impact of transaction size. To evaluate the impact of transaction size, we conducted an experiment where we submitted transactions with varying numbers of inputs and outputs to the coordinator. Figure 7a shows that as the number of inputs and outputs in transactions increased, we observed a decrease in transaction throughput, declining from 345,000 tx/s to 160,000 tx/s. This 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 additional experiments to understand the performance characteristics of the shards and signature validators, both components running in isolation. We found that increasing the number of shards or signature validators resulted in a scalable increase in the throughput of the respective components, surpassing 450,000 tx/s. We conclude that the bottleneck is not with the shards or signature validators but with the coordinator.

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%.



(a) (b) (c)

Figure 7: Impact on transaction throughput and latency of (a) varying number of inputs and outputs per transactions; (b) varying mix of valid and invalid transaction signatures; and (c) varying number of double spend transactions.

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 7b illustrates the changes in throughput and latency as we varying the mix of invalid transactions. Notably, we observed an intriguing trend: the overall throughput increased from $213,000$ tx/s to $296,000$ tx/s 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 $213,000$ tx/s to $207,000$ tx/s over the same range.

Double spendings. In the second scenario involving faulty transactions, we delved into the consequences of double spend incidents. These occure when a transaction is submitted with inputs that have already been recorded in the ledger, meaning that the input has already been spent. Figure 7c shows the impact of double spending on the throughput and latency of the committer. It became evident that the overall throughput decreased compared to the previous exeriment considering invalid transaction signatures. This effect can be attributed to the higher load 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.

6.4. Transaction Certification

We evaluate the transaction *certification* phase as described in Section B.3. 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 focus on the execution throughput and latency of the blind signature, which in the prototype corresponds to Pointcheval-Sanders' [33], [34]. We follow its threshold variation described in [35], [11]. Our prototype implementation uses [36]. With 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. Related Work

Central-bank digital currency. The architecture in [37] is one of earliest efforts to design centrally-governed cryptocurrencies, which are similar to CBDC. The proposed architecture, however, trusts the central bank to preserve transaction confidentiality. Hamilton Project [6] is a prominent solution for high-throughput CBDC settlement that introduces two CBDC architectures, only one of which, called Atomizer, yields a system of record. Although Hamilton is distributed, it is not decentralized, and thus not resilient to partial system compromise. Moreover, Hamilton's privacy is limited to user pseudonymity. In contrast, our decentralized framework reaches performances comparable to Hamitlon's Atomizer, while accommodating various privacy levels. In [38], Wüst et al. describe Platypus which couples the transaction processing model of e-cash with account-based transactions. Platypus addresses compliance requirements by enforcing holding and receiving limits, also and protects user privacy. On the downside, it does not support parallel transactions: a user cannot receive and send payments simultaneously. Recently, Kiayias et al. [39] introduced a framework for privacy-preserving CBDC, with elements of byzantine fault tolerance. Although the authors claim their system outperforms existing solutions, this is not confirmed experimentally. By contrast, our benchmarks demonstrate that our framework is amenable to horizontal scaling regardless of the privacy mechanisms used.

Privacy-preserving transparent payments. Miers et al. [40] introduced Zerocoin, which is a privacy-preserving extension of Bitcoin whose aim is to break the link between the inputs and the outputs of Bitcoin transactions. Zerocoin does not hide the transaction values, though. Poelstra et al. [41] leverages Pedersen commitments and zero-knowledge proofs to hide the value and type of UTXO transactions. Monero enhances the privacy of [41] by partially obfuscating the relation between transaction inputs and outputs using anonymity sets. In [42], Bünz et al. presented Zether, a privacy-preserving payment system tailored for Ethereum. Like Monero, it uses anonymity sets to partially hide the transaction graph. Zerocash [43] is the first **fully** privacy-preserving token management system. Notably, Zerocash guarantees that a transaction does not leak any information about the transacting parties, the transaction information, or the inputs being used in the transaction. By design Zerocash allows users to deny their participation in a transaction. This is clearly against the compliance requirements of CBDC. To address this, Androulaki et al. [9] exploit the properties of permissioned blockchains to build *a more efficient and fully privacy-preserving* token system with audit support. More specifically, the transactions in [9] verifiable ciphertexts that encrypt transaction information for the legitimate auditors. In this paper, we rely on the insights in [9] to build our framework.

8. Conclusion

In this paper, we described a distributed and privacy-preserving transaction processing framework suited for retail CBDC. We evaluated the framework using four token exchange protocols with varying degrees of privacy. Our results show that for a token system with only anonymity, our prototype implementation reaches a throughput of $80,000$ TPS in the case of Raft and SmartBFT, and $150,000$ TPS with our tailored consensus mechanism. Our results further demonstrate that thanks to the horizontal scaling of transaction processing compute, we *can achieve* comparable throughput numbers for token systems with stronger privacy guarantees.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, jun 1981.

[24] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM CCS*, pages 21–30. ACM, 2002.

[25] https://github.com/hyperledger-labs/fabric-smart-client.

[26] https://github.com/hyperledger-labs/fabric-token-sdk.

[27] https://github.com/hyperledger/fabric.

[28] https://go.dev/doc/devel/release\#go1.20.

[29] https://grpc.io.

[30] https://github.com/yugabyte/yugabyte-db.

[31] https://prometheus.io.

[32] https://www.ibm.com/cloud.

[33] 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.

[34] 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.

[35] 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.

[36] Threshold signature scheme library. https://github.com/IBM/TSS/.

[37] George Danezis and Sara Meklejohn. Centrally banked cryptocurrencies. In *Network and Distributed System Security Conference*, 2016.

[38] Karl Wüst, Kari Kostiainen, Noah Delius, and Srdjan Capkun. Platypus: A central bank digital currency with unlinkable transactions and privacy-preserving regulation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2947–2960, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Aggelos Kiayias, Markulf Kohlweiss, and Amirreza Sarencheh. Peredi: Privacy-enhanced, regulated and distributed central bank digital currencies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1739–1752, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411, 2013.

[41] 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.

[42] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 423–443, Cham, 2020. Springer International Publishing.

[43] 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.

[44] 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.

A. Proof Of Consistency Across Scalable Committers

In guaranteeing the uniformity of individual committers deployed across diverse organizations, it is imperative to verify the consistent validation and sequencing of transactions. This becomes particularly critical when transactions conflict with each other. The focal point is to establish and enforce two essential conditions between every pair of transactions T_i and T_j, where T_i precedes T_j in the overall transaction order:

1) **Condition 1.** For the validation of a transaction T_j that reads a state k, a prerequisite is processing, i.e., validating and committing/aborting, all transactions T_i (where $i < j$) that modified k.
2) **Condition 2.** The application of writes to each state must adhere to the correct chronological order. In other words, if transactions T_i and T_j (where $i < j$) both write to a state k, any subsequent transaction reading k should exclusively observe the write from T_j.

Utilizing "rw," "ww," and "wr" dependencies, the dependency graph guarantees the fulfillment of the aforementioned conditions, thereby ensuring consistency across committers.

Meeting Condition 1. When T_j's is added to the dependency graph, edges, i.e., "rw" dependency, are created from T_j to each T_i that writes to any state k read by T_j. Whenever the coordinator queries for dependency free transactions, T_j will not be returned until it has no out-edges. Thus, T_j will not be validated until every transaction T_i it depends upon is removed from the graph. It is important to note that a transaction is only removed from the graph when it is processed, i.e., validation followed by a commit or abort. Therefore, by the time T_j becomes dependency free, all transactions writing to states it reads would have been validated.

Meeting Condition 2. When T_j is introduced to the dependency graph, edges are established from T_j to each T_i that either reads or writes to any state k modified by T_j. The "ww" dependencies guarantee the correct sequencing of writes to any state k. The transaction T_j remains bound by dependencies until T_i is removed from the graph, a process that unfolds only after the database incorporates the writes made by T_i. Similarly, the "wr" dependencies ensure that until all preceding transactions T_i that have read k are validated, T_j remains constrained by dependencies.

Consequently, the updates initiated by T_j are withheld from application to the database until this validation is complete. As a result, no transaction T_i will perceive the writes from a subsequent transaction T_j.

B. Token Systems for CBDC

For the sake of simplicity, we only consider token systems with a single issuer. A generalization to systems with multiple issuers is straightforward. The validation of Issue transactions, in the single-issuer settings is the same[2] regardless of the privacy protections in place, thus we omit it in the following and only focus on Transfer transactions.

B.1. 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 [24], [44] 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 $(\mathrm{sk}, \mathrm{eid})$. 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)[3], 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 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.

B.2. Anonymous and Confidential Token Systems

In [41], Poelstra et al. show how to hide the values and still enable the balance preservation check. A token τ in [41] is a hiding commitment to pair (eid, v), i.e, $\tau = \mathrm{Commit}(\mathrm{eid}, v, t)$[4] for $v \in \mathbb{Z}_p$ and some randomly-chosen $t \in \mathbb{Z}_p$.

Let $\mathrm{ttx} = (in_0, in_1, out_0, out_1)$ 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_{Bal} that proves that the sum of the inputs equals the sum of the outputs. That is:

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

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 $\mathrm{max} << p$, 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 $(in_0, in_1, out_0, out_1, \Pi_{\mathrm{Bal}})$ such that each σ_b, $b \in \{0, 1\}$, proves the following:

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

The user then submits for validation

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

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_{Bal} 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

2. An Issue is valid if it was signed by the authorized issuer.

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

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

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.

B.3. 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 [43], 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 $(\mathrm{sk}, K, \mathrm{eid})$, where K is a secret key for a pseudo-random function PRF. Moreover, a token τ becomes a hiding commitment $\mathrm{Commit}(\mathrm{eid}, v, r, t)$, with r and t being two random numbers in $\mathbb{Z}_p$. A Transfer that spends τ, includes as input a pair $in = (\tau', sn)$. $\tau' = \mathrm{Commit}(\mathrm{eid}, v, r, t')$ is a randomization of τ and sn is τ's serial number, computed as $sn = \mathrm{PRF}(K, r)$.

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_{Bal} that proves that out_0 and out_1 sum up to the same value as in_0 and in_1, i.e, Π_{Bal} shows that:

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

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:

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

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 [11], 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:

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

At this point, $\mathrm{ttx} = (in_0, in_1, out_0, out_1, \Pi_{\mathrm{Bal}}, \Pi_{sn}, \Pi_{\mathrm{Exist}})$, 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:

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

The resulting transaction

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

is then submitted for validation.

Validation of Unlinkable $\mathrm{Transfer}$ 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_{Bal} 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.

C. Token Systems without Transaction Unlinkability

Transactions in systems without unlinkability 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 transactions can

1) reveal all the transaction information (zero privacy);
2) hide the identities of the transacting parties (anonymity), using for example *anonymous credentials*;
3) hide the values of the transactions (confidentiality);
4) hide both (anonymity and confidentiality).

C.1. Transaction Model

In systems without transaction unlinkability, the client transaction is a *standard* UTXO transaction, with inputs and outputs, and the signatures of the issuer or the input owners. When transaction confidentiality is desired, the client transaction will additionally carry a zero-knowledge proof that shows that the sum of transaction's inputs equals the sum of its outputs.

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 $\langle \mathrm{out\_key}, \mathrm{ser\_out} \rangle$ corresponding to the transaction outputs. The key $\mathrm{out\_key}$ is the *unique identifier* of the output in the ledger, which is set to the concatenation of the transaction's unique identifier, the hash of the output, and the index of the output in the transaction, and $\mathrm{ser\_out}$ is its serialization. Similarly, the settlement transaction of a $\mathrm{Transfer}$ carries a write-set with two types of entries: the first contains the outputs of the transaction $\langle \mathrm{out\_key}, \mathrm{ser\_out} \rangle$, and the second the delete instructions of the inputs represented by $\langle \mathrm{in\_key}, \mathrm{isDelete} = \mathrm{true} \rangle$, where $\mathrm{in\_key}$ is the key of the input in the ledger. Furthermore, the settlement transaction contains a read-set with entries $\langle \mathrm{in\_key}, 0 \rangle$. 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. Committing one of these transactions results in deleting the input and incrementing the version of its key from 0 to 1. This leads the other transaction to fail due to an outdated read-set entry.

Next, we explain the functionalities of the components of our architecture focusing on $\mathrm{Transfer}$ transactions. Issue transactions adopt a similar processing flow.

C.2. User Wallets

The wallet securely stores the user's long-term identity defined by triple $(\mathrm{eid}, \mathrm{sk}, \mathrm{cred})$, whereby eid is the unique identifier of the user, sk her secret key, and cred is the signature from the registration authority on pair $(\mathrm{eid}, \mathrm{sk})$. 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, $\tau_i = (\mathrm{eid}, v_i)$ and the wallet stores $\langle \tau_i, \mathrm{in\_key}_i \rangle$, where $\mathrm{in\_key}_i$ is the key of τ_i in the ledger. In a token system with anonymity, $\tau_i = (c_i, v_i)$, where $c_i = \mathrm{Commit}(\mathrm{eid}, t_i)$ is a *hiding commitment* of eid computed with randomness t_i, and the wallet stores $\langle \tau, t_i, \mathrm{in\_key}_i \rangle$. Finally, in a system with transaction confidentiality, $\tau_i = \mathrm{Commit}(\mathrm{eid}, v_i, t_i)$ is a hiding commitment to pair (eid, v_i) using randomness t_i, and the wallet stores $\langle \tau_i, v_i, t_i, \mathrm{in\_key}_i \rangle$.

Assume that the user instructs its wallet to transfer a token of value v' to a payee identified by eid'. The user wallet correspondingly assembles the client transaction as follows:

- Select tokens τ_i such that the sum value of these tokens exceed v'. Let τ_0 and τ_1 denote the selected tokens.
- Create two outputs out_0 and out_1, such that out_0 is intended for the payee with value v', and out_1 is the change to be returned to the payer, and hence, of value $v_0 + v_1 - v'$.
- Produce tuple $(in_0, in_1, out_0, out_1)$, where $in_0 = (\tau_0, \mathrm{in\_key}_0)$ and $in_1 = (\tau_1, \mathrm{in\_key}_1)$.
- When transaction confidentiality is required, compute a zero-knowledge proof Π_{Bal} to prove that out_0 and out_1 sum up to the same value as τ_0 and τ_1. Let $\mu = (in_0, in_1, out_0, out_1, \Pi_{\mathrm{Bal}})$.
- Compute the unique transaction identifier TxID as the concatenation of the hash of μ and some randomness. The randomness is only necessary if the token system offers no privacy protections.
- Subsequently, compute two signatures σ_0 and σ_1 on (TxID, μ) and define the client transaction as tuple $\mathrm{ctx} = (\mathrm{TxID}, \mu, \sigma_1, \sigma_0)$.
- Then communicate to the payee the identifier TxID of the transaction. In anonymous token systems, the payer also communicates the randomness used to compute out_0. When transaction confidentiality is supported, the payer additionally transmits v'.
- Finally, submit ctx to the token endorsers.

C.3. Settlement Engine

In this section, we expand on the components of the settlement engine.

Token Endorsers & Token Chaincode. Upon receiving a client transaction, the token endorsers run the token chaincode, which checks if:

- TxID contains the hash of $(in_0, in_1, out_0, out_1, \Pi_{\mathrm{Bal}})$;
- $in_0 = (\mathrm{in\_key}_0, \tau_0)$ and $in_1 = (\mathrm{in\_key}_1, \tau_1)$, and $\mathrm{in\_key}_0$ and $\mathrm{in\_key}_1$ include the hash of τ_0 and τ_1 respectively;
- Π_{Bal} is a valid zero-knowledge proof;
- σ_0 and σ_1 are valid signatures by the owners of τ_0 and τ_1 over $(\mathrm{TxID}, in_0, in_1, out_0, out_1, \Pi_{\mathrm{Bal}})$.

If any of these checks fail, then the chaincode rejects. Otherwise, it produces a read-write set as described in Section C.1, and assembles a settlement transaction with identifier TxID using the produced read-write sets and submits the result to the ordering service.

Committers. Committers validate the ordered transactions as described in Section 3.2. Notice that thanks to the way we define the read-write sets (see Section C.1), any attempts to double spend a token will result in a read conflict, invalidating the double-spending transaction.