FabZK: Supporting Privacy-Preserving, Auditable Smart Contracts in Hyperledger Fabric

Hui Kang* *IBM Research* *kangh@us.ibm.com*
Ting Dai*† *NC State University* *tdai@ncsu.edu*
Nerla Jean-Louis *IBM Research* *nerla.jeanlouis@ibm.com*
Shu Tao *IBM Research* *shutao@us.ibm.com*
Xiaohui Gu *NC State University* *xgu@ncsu.edu*

***Abstract*—On a Blockchain network, transaction data are exposed to all participants. To preserve privacy and confidentiality in transactions, while still maintaining data immutability, we design and implement FabZK. FabZK conceals transaction details on a shared ledger by storing only encrypted data from each transaction (e.g., payment amount), and by anonymizing the transactional relationship (e.g., payer and payee) between members in a Blockchain network. It achieves both privacy and auditability by supporting verifiable Pedersen commitments and constructing zero-knowledge proofs. FabZK is implemented as an extension to the open source Hyperledger Fabric. It provides APIs to easily enable data privacy in both client code and chaincode. It also supports on-demand, automated auditing based on encrypted data. Our evaluation shows that FabZK offers strong privacy-preserving capabilities, while delivering reasonable performance for the applications developed based on its framework.**

***Keywords*-Blockchain; privacy; auditability; zero-knowledge proofs**

I. Introduction

As distributed and immutable digital ledger, Blockchain offers significant business benefits, such as greater transparency, enhanced security, improved traceability and efficiency in business settlement. While these benefits have motivated a myriad of Blockchain applications, a significant subset of these application scenarios require Blockchain systems to provide additional guarantees on data privacy and confidentiality. Several recent data breach incidents [1], [2], [3], [4], [5], [6], exemplified the importance of meeting such requirements. In addition, many applications also demand auditability of transactions in the underlying Blockchain systems, so that transactions on a Blockchain network can be audited without infringing data privacy. For example, in a stock exchange market, sellers and buyers may not want to reveal trading details to others, yet auditors need to be able to independently verify all transactions.

Blockchain networks can be permissioned or permissionless. Both types of systems can preserve data privacy to some extent. Some permissionless systems, such as Bitcoin [7], Ripple [8], Digital Asset [9], and Stellar [10], preserve privacy by keeping the hashes of transaction data on chain, while storing plain transaction data off chain. Lacking support of auditing on-chain data, these systems have to allow external auditor to access their private off-chain data and risk exposing sensitive information. Other permissionless systems, such as Zcash [11], [12], Ethereum [13], and Confidential Transactions and Assets [14], [15] use cryptographic commitment schemes to obscure transaction information. However, these systems either require a trusted setup or reveal the transaction graph (sending and receiving parties in a transaction) to all network members. In general, none of the existing permissionless blockchain systems preserves full data privacy and support auditability at the same time.

Permissioned blockchains, such as Hyperledger Fabric [16] and Quorum [17], preserve data confidentiality and privacy via private channels (or peer networks) by enforcing access control, so that only admitted channel participants can access its resources (e.g., chaincodes, transactions, and ledger states). Such private channels do not support privacy-preserving audit by default.

Auditable privacy-preserving transaction, also known as zero-knowledge asset transfer [18], is a model designed for the aforementioned application scenarios. It allows members to exchange assets and to record transactions in the shared ledger, without revealing the fact that they are transacting, with whom they are transacting, or the transaction amount. With zero-knowledge asset transfer, each user can assign auditors to access all of their transactions. An auditor can validate the legitimacy of the user's transactions, without violating the user's privacy.

Existing solutions to this problem, such as Solidus [19] and zkLedger [20], support auditability by using either publicly-verifiable oblivious RAM machines (PVORM) or audit tokens. Solidus only works in bank-intermediated systems where a modest number of banks maintain a large number of user accounts. It exposes the transaction graph between users and their affiliated banks, as well as the transaction graph among the banks. Moreover, Solidus enables *public* auditing by revealing all keys used in the system to an auditor, therefore it does not fully protect user privacy. zkLedger supports *private* auditing, but in an inefficient manner: it requires auditors and all participants to actively validate and commit each transaction sequentially, which

*Ting Dai and Hui Kang contributed equally to this work.
†Part of the work was done during an internship at IBM Research.

significantly reduces the overall throughput of transaction.

In this paper, we present FabZK, an extension to the Hyperledger Fabric that enables complete protection of data integrity, privacy, and confidentiality. FabZK realizes efficient, privately auditable, and privacy-preserving peer-to-peer transactions by designing a set of Non-Interactive Zero-Knowledge (NIZK) proofs on Pedersen commitments [21]. To improve throughput, we introduce a two-step validation approach to support concurrent transactions. In FabZK, each participant conducts active and lightweight auto-validation, when a transaction is appended to the public ledger. An auditor periodically monitors ledger activities and validates transactions based only on the encrypted data and proofs.

Our work makes the following contributions:

- We develop a theoretical model via Pedersen commitments and refined NIZK proofs (Section III). The augmented NIZK proofs provide strong transaction privacy, public verifiability, and provable auditing.
- We design an application development framework, including APIs, on top of Fabric (Section IV). This allows application developers to easily create auditable and privacy-preserving blockchain applications.
- We implement the proposed model into a real-world solution, i.e., FabZK, and introduce various optimizations to achieve reasonable performance (Section V).
- We compare FabZK with state-of-the-art approaches such as zk-SNARKs, native Fabric, and zkLedger (Section VI). Our evaluation shows that FabZK offers superior performance trade-offs. FabZK is more efficient than zk-SNARKs in generating and verifying proofs. Compared to the native Fabric system, it enables auditable privacy-preserving transactions at the cost of 3% to 32% throughput loss and less than 10% latency increase. Compared to zkLedger, FabZK's throughput is up to 180× higher.

We discuss background in Section II, present related work in Section VII, and finally conclude in Section VIII.

II. Background

In this section, we provide some background of the Hyperledger Fabric, and discuss the key concepts in FabZK design, i.e., confidential transactions and anonymity schemes.

A. Hyperledger Fabric

Hyperledger Fabric is a permissioned blockchain system for recording transactions between organizations. In Fabric, organizations form consortia and transact with each other on private channels. Fabric provides access control mechanism, so that the data and resources on a private channel can only be accessed by admitted organizations.

Business logic shared by a consortium of organizations is programed as *chaincode*, also known as *smart contract*. Chaincode enables different parties to automate tasks that

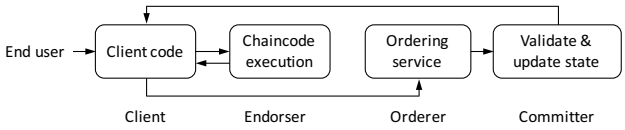

Figure 1: Components and data flow in Hyperledger Fabric.

are traditionally performed through an external intermediary [22]. Organizations in the same consortium execute identical chaincode to process transactions, produce and store data on an *immutable shared ledger*.

Unlike the traditional *order-execute* architecture adopted in systems such as Tendermint [23] and Chain [24], Fabric introduces the *execute-order-validate* blockchain architecture. This architecture supports concurrent execution and post-ordering determinism through pluggable consensus algorithms [25], [26]. Specifically, transaction data are computed by peers concurrently, forwarded to the ordering service, disseminated to all other peers and appended to their immutable ledgers. Figure 1 illustrates the above data flow, along with following key components in a Fabric network:

- *Client* invokes chaincode execution by submitting a transaction *proposal* to the endorser nodes. It then collects the signed *endorsements* in the proposal, assembles a transaction, and broadcasts it to the orderer. A client can subscribe to a channel to receive updates from the ledger (e.g., a new transaction block being committed).
- *Endorser* runs chaincode and creates transaction endorsement, which includes a *write set* with state updates produced by simulating the transaction proposal, a *read set* capturing the version dependencies of the proposal simulation, and the endorser's signature.
- *Orderer* establishes the total order of transactions in a channel, batches transactions into blocks, and distributes them to all committers in the channel.
- *Committer* validates each transaction in a block by checking its compliance to endorsement policy and any read-write conflicts. Then it appends the transaction to the ledger.

Since the endorser, orderer, and committer are deployed on the Fabric platform, they are considered on-chain components. In contrast, the client is not part of the deployment and is thus running off chain. Client code interacts with the Fabric platform through its SDK [27], [28], [29]. The shared ledger is replicated on each peer node which can contain an endorser, a committer, or both.

Although the consortium-based Fabric contains a certain degree of knowledge about each user, member organizations may still want to keep the actual transactions private, due to business or privacy concerns. This underlies the necessity to enable confidential transactions in Fabric with anonymity.

B. Confidential Transaction and Anonymity Schemes

On a private channel, transaction details are visible to all members, regardless of their involvement in the transaction. We aim to develop schemes to prevent non-involving organizations (which we call hereafter *non-transactional organizations*) from accessing transaction details, such as transaction amount and *transactional organizations* (i.e., sender and receiver), from the ledger. Meanwhile, we also need these schemes to be auditable. To achieve this goal, we adopt and extend three existing techniques in FabZK: tabular structured ledger [20], Pedersen commitments [21], and NIZK proofs [30].

A tabular structured ledger is an anonymity scheme proposed in zkLedger [20] to conceal the transaction graph and to prevent an organization from hiding assets on the ledger. It maintains a two-dimensional table, where each row represents a single transaction and each column represents the transaction history of an organization. For example, a tabular structured ledger for an N-organization channel has N columns, and has M rows if M transactions in total have occurred on this channel.

To hide the transaction amount u, a Pedersen commitment [21] is computed for u with a random number r by

$$\text{Com} = com(u, r) = g^u h^r, \tag{1}$$

where g and h are two random generators of a cyclic group $\mathbb{G}$ with $s = |\mathbb{G}|$ elements and prime order p, $\mathbb{Z}_p = \{0, 1, \ldots, s-1\}$, $u \in \mathbb{Z}_p$, and $r \in \mathbb{Z}_p$.

An outsider cannot tell the transaction amount from a Pedersen commitment, or whether it is positive, negative, or 0. In addition, as long as Pedersen commitments are computed for both transactional and non-transactional organizations, it is impossible to identify the sender and receiver of a transaction, so that the transaction graph is also concealed. To support audit, an audit token is assigned to a Pedersen commitment $com(u, r)$:

$$\text{Token} = \text{pk}^r, \tag{2}$$

where pk is the public key of an organization, $\text{pk} = h^{\text{sk}}$, sk is the private key of the organization, r and h are the same as in Equation (1).

With transactions encrypted as Pedersen commitments, our system need to allow validation and auditing over the encrypted data. This is achieved by a set of NIZK proofs. We describe these proofs in detail in Section III.

III. Preliminaries

In this section, we design NIZK proofs required for auditing encrypted transaction data, and introduce their usage in private and public ledgers, in our FabZK system.

A. NIZK Proofs

In FabZK, the spending organization (i.e., sender) is responsible for creating commitments, tokens, and NIZK proofs for other organizations or an auditor to verify. Without loss of generality, we assume that each transaction has one spending and one receiving organization throughout the paper.[1]

Proof of Balance is a known method for verifying the overall balance in a single transaction row, i.e., $\sum_{i=1}^{N} u_i = 0$ (N is the number of columns). It validates the commitments in a row by leveraging the homomorphism of Pedersen commitment, i.e., $\prod_{i=1}^{N} \text{Com}_i = (g^{\sum_{i=1}^{N} u_i}) \cdot (h^{\sum_{i=1}^{N} r_i})$. The prover chooses r_i that satisfies $\sum_{i=1}^{N} r_i = 0$ to generate the commitments in Equation (1). The verifier checks whether $\prod_{i=1}^{N} \text{Com}_i = 1$ in that row. If this condition holds, then the ledger is proved to be balanced.

Proof of Correctness prevents an organization from making an incorrect or fraudulent transaction to steal assets from others.

To verify the correctness of the amount of a transfer tx_m (m is the index of current transaction), each organization uses the audit token to check if

$$\text{Token}_m \cdot g^{\text{sk} \cdot u_m} = (\text{Com}_m)^{\text{sk}}, \tag{3}$$

where sk is the organization's private key and u_m is its transaction amount. For non-transactional organizations, they are aware of the existence of tx_m, but are not involved in tx_m, thus their transaction amount is 0. If any organization fails to verify Equation (3), it indicates incorrect tx_m.

Equation (3) shows that verifying the *Proof of Correctness* can be achieved with only the data in the current transaction tx_m; so does *Proof of Balance*. This is an important feature that we will leverage in the two-step validation in Section IV.

Proof of Assets ensures that a spending organization has enough assets to execute the transaction. In a tabular structured ledger, a column represents all assets an organization has received or spent [20]. *Proof of Assets* verifies that the sum of all committed values in a column, including that of the current transaction, is non-negative.

We let the prover generate a range proof for the spending organization's remaining balance:

$$\text{RP} = \text{ZK}(u_{\text{RP}}, r_{\text{RP}} : \text{Com}_{\text{RP}} \wedge l_0 \leqslant u \leqslant l_p), \tag{4}$$

where $\text{ZK}(u_{\text{RP}}, r_{\text{RP}} : \text{Com}_{\text{RP}})$ is a zero-knowledge proof of u_{RP} such that $\text{Com}_{\text{RP}} = g^{u_{\text{RP}}} h^{r_{\text{RP}}}$, g and h are known to the verifier, $u_{\text{RP}} = \sum_{i=0}^{m} u_i$, r_{RP} is a random number different from the r in Equation (1), l_0 and l_p are two bound values.

We use the inner-product range proof from BulletProofs [31] to prove that an account's remaining assets $\sum_{i=0}^{m} u_i \geqslant 0$, in encrypted form. The details are described in appendix.

[1] Our approach can be adapted to more complex scenario, such as multiple senders, which will be addressed in our future work.

Proof of Amount guarantees that the transaction amount is within a certain range. In a scenario with a single spending and a single receiving organization, the prover generates an inner-product range proof to the transaction amount of the receiving organization $u_{\text{recv}} \in [0, 2^t)$, with a random number r_{recv}.

In each transaction, the prover also needs to generate indistinguishable cryptographic primitives for all non-transactional organizations, in order to conceal the transaction graph. These primitives are a Pedersen commitment of 0 with a random number r_i, an audit token, a range proof to 0 with another random number r_{RP_i}, and a disjunctive proof (discussed later in this section).

Besides the four aforementioned NIZK proofs, each organization needs to further check whether these proofs are consistent with each other, i.e., the same parameters are used in different proofs for the same organization. To achieve this goal, we introduce *Proof of Consistency*.

Proof of Consistency ensures that 1) for a spending organization, the generated range proof is consistent with its remaining assets $\sum_{i=0}^{m} u_i$, instead of some arbitrary $u_{\text{arb}} \in [0, 2^t)$, 2) for other organizations, the generated range proofs are consistent with their current transaction amounts. We also need to make sure that the verifier can validate the *Proof of Consistency* without knowing the identity of the spending organization.

To provide such a *Proof of Consistency*, we use a non-interactive variant of the Chaum-Pedersen zero-knowledge proofs [32] to construct a disjunctive zero-knowledge proof (DZKP) for a transaction tx_m. A DZKP takes the spending organization's private key sk, or random numbers r and r_{RP} (for other organizations), as input and generates two non-interactive Σ-protocols [33].

In addition, we generate two tokens Token' and Token'' paired with the DZKP:

$$\text{Token}' = \begin{cases} \text{pk}^{r_{\text{RP}}}, & \text{for the spending org.,} \\ t \cdot (\text{Com}_{\text{RP}}/s)^{\text{sk}}, & \text{otherwise.} \end{cases} \quad (5)$$

$$\text{Token}'' = \begin{cases} \text{Token} \cdot (\text{Com}_{\text{RP}}/s)^{\text{sk}}, & \text{for the spending org.,} \\ \text{pk}^{r_{\text{RP}}}, & \text{otherwise.} \end{cases} \quad (6)$$

where r_{RP} is the random number used in Equation (4), $s = \prod_{i=0}^{m} \text{Com}_i = g^{\sum_{i=0}^{m} u_i} h^{\sum r_i}$ is the product of an organization's commitments from row 0 to row m, and $t = \prod_{i=0}^{m} \text{Token}_i = h^{\text{sk} \sum_{i=0}^{m} r_i}$ is the product of the organization's audit tokens from row 0 to row m. Detailed cryptographic primitives about DZKP and how it works are described in appendix.

B. Private and Public Ledgers

In a blockchain network, each organization maintains two ledgers: a private, off-chain ledger and a public, on-chain ledger. Public ledger is for recording and auditing transactions on the Fabric channel, while private ledger is an organization's private datastore. Both ledgers have a tabular structure, as shown in Figure 2.

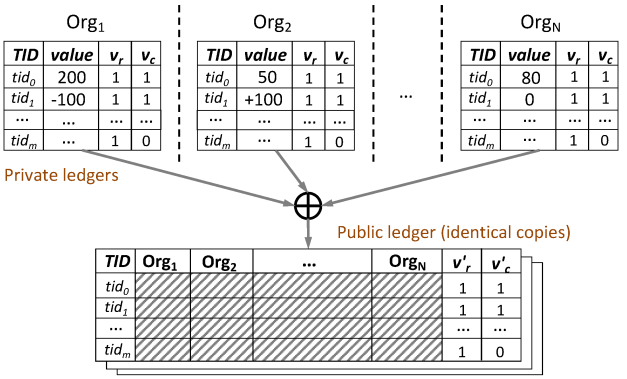

Figure 2: Private ledgers and the public ledger. The public ledger has many identical replicas on the Fabric peer nodes, owned by participating organizations.

Private Ledger stores transaction data in plaintext. It is only accessible to and maintained by the possessing organization. As shown in Figure 2, a private ledger table has four columns: (1) a transaction identifier, i.e., *tid*; (2) the transaction amount, i.e., *value*; (3) a validation bit v_r that indicates whether a transaction is valid, verified by *Proof of Balance* and *Proof of Correctness*; and (4) a validation bit v_c that indicates whether a transaction is valid, verified by *Proof of Assets*, *Proof of Amount* and *Proof of Consistency*. We discuss the rationale behind separating the validations of the five NIZK proofs in Section IV.

Public Ledger is maintained by all peer nodes, owned by participating organizations. As shown in Figure 2, the public ledger for an N-organization channel is a table with $N+3$ columns, corresponding to a transaction identifier, N $\langle \text{Com}, \text{Token}, \text{RP}, \text{DZKP}, \text{Token}', \text{Token}'' \rangle$ sextets, and two validation bitmaps. Like private ledgers, rows in the public ledger represent transactions. The bitmaps (i.e., v'_r, v'_c) are composed of N bits representing the validation results from N organizations.

The public ledger is bootstrapped by computing the Pedersen commitments and audit tokens of the initial values for all organizations in the first row with transaction identifier tid_0, denoted by the $\langle \text{Com}, \text{Token} \rangle$ tuples. We require the client of each organization to validate *Proof of Correctness* for itself. The system assumes that all organizations' initial assets are already validated at the bootstrap time.

In Figure 2, the rows with transaction identifier tid_1 exemplifies a transfer of 100 units of assets from Org_1 to Org_2. In their private ledgers, Org_1 and Org_2 set the transaction value in row tid_1 as -100 and $+100$, respectively. The transaction amount for other organizations in tid_1 is set to 0. We store N commitments (i.e., $\langle com(value_i, r_i) \rangle, i = 1, \ldots, N$) at row tid_1 in public ledgers. Each commitment is associated with a $\langle \text{Token}_i, \text{RP}_i, \text{DZKP}_i, \text{Token}_i', \text{Token}_i'' \rangle$

quintet, which is used to validate transaction tid_1 with the five NIZK proofs.

Hyperledger Fabric constructs transactions via chaincode installed on the channel as an agreement of the consortium, so only the transactions made by approved chaincode will be accepted. FabZK follows a similar design principle: the cryptographic primitives are computed by the extended Fabric system, not externally by upper level applications. This guarantees that no malicious user can manipulate the outputs generated by FabZK, hence the validations are trustable.

FabZK writes $\langle \text{Com}, \text{Token}, \text{RP}, \text{DZKP}, \text{Token}', \text{Token}'' \rangle$ sextets in the public ledger for both transactional and non-transactional organizations. Although the extra padding incurs some overhead in storage size, this design allows FabZK to hide the transaction graph and achieve one-to-one mapping between private and public ledgers. Moreover, the one-to-one mapping makes it easier for an organization or auditor to track and validate the transactions.

IV. FabZK Design

In this section, we provide an overview of FabZK's architecture and its enhanced program execution flow. Then, we introduce FabZK's programming interfaces for writing auditable, privacy-preserving blockchain applications.

A. Overview

Figure 3 shows the FabZK architecture. It augments the current Fabric system to allow channel participants to make privacy-preserving transactions with each other, and to allow non-transactional organizations and trusted third-party auditors to audit the results. To do so, a FabZK program runs in four phases: *preparation*, *execution*, *notification*, and *two-step validation*. Preparation and notification phases run on the client nodes, while execution and validation phases run in the chaincode on endorsers. Among the four phases, *execution* and *two-step validation* are specifically designed to support privacy and audit.

B. Program Execution Flow

On a Fabric channel, we suppose that a deal is made privately between a spending and a receiving organization. The transaction is then reflected in both their private ledgers and the public ledger. With FabZK, we formulate the problem as a program execution flow, illustrated in Figure 3.

Preparation: At the beginning of an execution, FabZK requires the spending and receiving organizations to first determine the transfer amount (u), outside of the blockchain network. Then, the spending organization's client code constructs the transaction, which consists of N tuples corresponding to the N columns of the public ledger. Each tuple contains the transaction amount ($\pm u$ for transactional organizations and 0 for non-transactional organizations), a random number, and the organization's public key. These tuples reflect the involvement of individual organizations

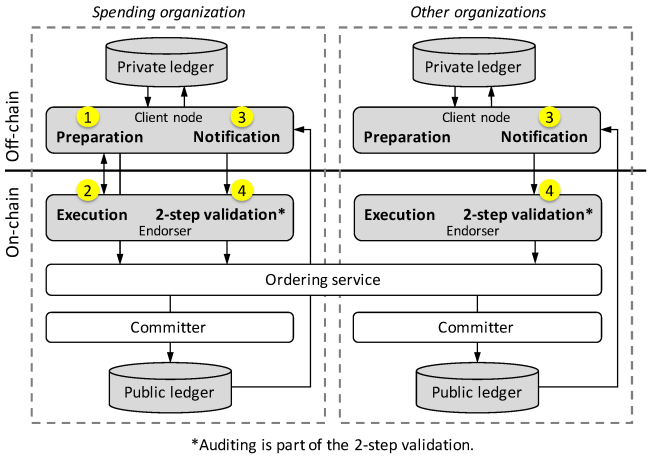

Figure 3: System architecture and program execution flow. The grey boxes denote FabZK's four major stages: preparation, execution, notification and two-step validation. Dashed boxes represent isolated organizations.

in this transaction. After this preparation, the transaction specification is sent by the spending organization's client code to its endorsers to invoke the *transfer* execution, in the Fabric network.

Execution: On receiving the transaction specification, the transfer chaincode written with FabZK's API is executed to convert the plaintext specification to N $\langle \text{Com}, \text{Token} \rangle$ tuples. The tuples represent the transfer amount of individual organizations and form a new row on the public ledger. The execution results are returned to the client code as an endorsement. The client code assembles the endorsement, and broadcasts it to the ordering service. As shown in Figure 3, the transfer chaincode is executed only by the spending organization.

Notification: Once executed, all organizations on the channel are informed of the transaction output (i.e., N $\langle \text{Com}, \text{Token} \rangle$ tuples) through the standard Fabric notification mechanism. Specifically, the orderers order transactions from different organizations, batch them into blocks, and deliver the blocks to the committers. The committers validate endorser signatures inside each transaction, check the read-write set conflicts, and append the transaction to the public ledger. Meanwhile, a notification is sent to each organization's client code. With the FabZK API, each client code retrieves information from its private ledger and invokes the two-step validation process to verify the change on the public ledger.

Two-step Validation: Provided a transaction's N tuples constructed by the spending organization, all other organizations need to verify whether these tuples embody a valid transaction using the five NIZK proofs described in Section III-A. To improve performance, we design the validation process as two steps to enable parallel execution (Section V-B).

Step one ensures that no asset is created or destroyed

Table I: FabZK's client code and chaincode APIs.

Client code APIs

Name	Description
PvlGet	Retrieve transaction content from private ledger
PvlPut	Append transaction content to private ledger
Validate	Invoke the *validation* chaincode to validate a transaction
GetR	Return a list of random numbers which sum to 0

Chaincode APIs

Name	Description
ZkPutState	Compute commitments and audit tokens
ZkAudit	Compute range proofs and disjunctive proofs
ZkVerify	Verify the proofs against the input from client code

during the transaction and no organization steals assets from others. Each organization checks whether a row on the public ledger satisfies *Proof of Balance* and whether its corresponding cell in the row satisfies *Proof of Correctness*.

Step two ensures that the spending organization owns enough assets to execute the transaction, and the transaction amount is within the predefined upper and lower bounds. This step is usually activated by a trusted third-party auditor. To start this step, auditor asks the spending organization to generate range proofs and disjunctive proofs. For the m-th row, the spending organization's client code constructs an audit specification, which includes its remaining balance (i.e., $\sum_{i=0}^{m} u_i$), a set of the transaction amounts for the rest of the organizations, three sets of random numbers (i.e., r_{RP} in Equation (4) and w_1, w_2 in Equation (7)), the commitment product set (the product of an organization's commitments from row 0 to row m), the token product set (the product of an organization's audit tokens from row 0 to row m), all organizations' public keys, and the spending organization's private key.

It is safe for the spending organization to provide its private key to the chaincode, because chaincode runs on the organization's own endorsers. The audit specification is sent to the organization's endorsers to invoke the *audit* chaincode execution. The chaincode then converts the plaintext audit data to $\langle \text{RP}, \text{DZKP}, \text{Token}', \text{Token}'' \rangle$ quadruples for each organization. The auditor checks *Proof of Assets*, *Proof of Amount* and *Proof of Consistency* for all organizations. A transaction is considered valid only when all checks are positive. Finally, the validation result is updated on the public ledger, which results in another notification to all organizations, who will then update their private ledgers.

C. Programming Interfaces

FabZK provides two sets of programming interfaces to enable an application's interaction with the Fabric system. The client code APIs support the interactions during *preparation* and *notification* stages. The chain code APIs support the interactions during *execution* and *two-step validation*. Table I shows the specifications of these APIs.

Client Code APIs support read from and write to an organization's private ledger. Via these APIs, an application can construct and maintain the private ledger, as well as submit transactions to the blockchain network via the Fabric SDK. The `PvlGet` API is used to retrieve rows by transaction identifier. When a new transaction arrives or a submitted transaction is validated, the `PvlPut` API is called to update the private ledger. Client code uses the `Validate` API to invoke the *validation* chaincode to verify a new transaction.

As discussed in Section III-A, in order to generate a valid and publicly verifiable *Proof of Balance*, the random numbers in a transaction specification must satisfy $\sum_{i=1}^{N} r_i = 0$. These random numbers can either be generated in the chaincode or provided as arguments by the client code. In Fabric, each organization can own multiple peer nodes for fault tolerance. A transaction request can be sent to multiple peers of the initiating organization to get endorsements. To ensure that consistent random numbers are used by independent peers for the same transaction, we provide the `GetR` API to the client code, so that the same random numbers can be distributed to all the endorsing peers.

Chaincode APIs are used to read/write data from/to the public ledger. The `ZkPutState` API is called during the execution stage, when the spending organization initializes a transfer. It converts a transaction specification to the commitments and audit tokens, serializes them into a byte stream, and invokes the native Fabric API, `PutState`, to generate a *write* set, which is stored in a transient data store on the endorsing peer and returned to the spending organization.

The `ZkVerify` and `ZkAudit` APIs are called collaboratively during the two-step validation phase to verify transactions in the public ledger. In step one, `ZkVerify` is invoked by individual organizations to check *Proof of Balance* and *Proof of Correctness* for a given row of transaction. In step two, all organizations invoke `ZkAudit` in the chaincode to create range proofs, disjunctive proofs, and the two tokens in Equation (5) and (6), i.e., $\langle \text{RP}, \text{DZKP}, \text{Token}', \text{Token}'' \rangle$ quadruples for each transaction. Finally, `ZkVerify` is called again to check *Proof of Assets*, *Proof of Amount*, and *Proof of Consistency*. A transaction is considered valid if `ZkVerify` has successfully validated all five proofs.

V. System Implementation

In this section, we present the implementation details of FabZK. We specifically focus on two practical design aspects: data structure on the public ledger and parallelizing the computation. We also explain how to write FabZK applications.

A. Data Structure of Public Ledger

Recall that, in a tabular structured public ledger (Figure 2), each row represents a single transaction, containing three types of transaction data: a $\langle \text{Com}, \text{Token} \rangle$ tuple, a $\langle \text{RP}, \text{DZKP}, \text{Token}', \text{Token}'' \rangle$ quadruple, and validation state. We implement the schema of a FabZK row with the `zkrow`

```
//zkrow represents a row in the public ledger
message zkrow {
    map<string, OrgColumn> columns = 1;
    bool isValidBalCor = 2;
    bool isValidAsset = 3;
}
//OrgColumn represents one organization
message OrgColumn {
    // transaction content
    bytes commitment = 1;
    bytes auditToken = 2;
    // two step validation state
    bool isValidBalCor = 3;
    bool isValidAsset = 4;
    // auxiliary data for proofs
    bytes TokenPrime = 5;
    bytes TokenDoublePrime = 6;
    RangeProof rp = 7;
    DisjunctiveProof dzkp = 8;
}
```

Figure 4: Data structure for a row in FabZK's public ledger in protobuf language [34]. Due to space limitations, details of `RangeProof` and `DisjunctiveProof` are omitted.

data structure, shown in Figure 4. A `zkrow` constructs all organizations' data as multiple `columns` and holds the validation state of that row. Each column is a key/value pair, where the key is an organization's name (or ID) and the value is typed as an `OrgColumn`, storing the three types of transaction data.

The contents of the ledger data structures are filled by the chaincode APIs. The $\langle \text{Com}, \text{Token} \rangle$ tuple and $\langle \text{RP}, \text{DZKP}, \text{Token}', \text{Token}'' \rangle$ quadruple are created by the `ZkPutState` and `ZkAudit` APIs, respectively. The two validation states, i.e., `OrgColumn.isValidBalCor` and `OrgColumn.isValidAsset` are set by the `ZkVerify` API during the two-step validation. After all `OrgColumns`' validation states are set, the result of the logical AND operation of these states are assigned to `zkrow.isValidBalCor` and `zkrow.isValidAsset`, respectively.

B. Parallelizing Computation

Because of the computation overhead of the cryptographic algorithms, we need to further optimize program execution to improve FabZK's performance. We focus on parallelizing the computation during the *execution* and *two-step validation* phases, since these two phases account for most of the computation overhead.

In the execution phase, we observe that the computations of $\langle \text{Com}, \text{Token} \rangle$ tuples for different organizations are independent of each other. These computations also do not require accessing historical data. Therefore in our implementation, a spending organization creates multiple threads to compute $\langle \text{Com}, \text{Token} \rangle$ tuples for all organizations concurrently.

In the two-step validation phase, we realize that the computations of *Proof of Balance* and *Proof of Correctness* have no dependency upon historical data or dependency across different organizations. However, the other three proofs have to be computed sequentially, due to the two constraints of Equation (4). First, computing a range proof for the m-th row requires data from row 0 to row m, e.g., $u_{\text{RP}_m} = \sum_{i=0}^{m} u_i$. Thus, `ZkAudit` cannot be invoked until all previous results are computed. Second, range proofs and disjunctive proofs can only be computed by the spending organization, because other organizations are not aware of the sender's available assets or the transaction detail. These two constraints, together with the fact that the spending organization varies by transaction, dictate that the computations of range proofs and disjunctive proofs have to be performed sequentially.

In our implementation, the first step of validation is fully parallelized, i.e., the computations of *Proof of Balance* and *Proof of Correctness* are distributed to all peer nodes. The second step of validation is partially parallelized: the spending organization can launch multiple threads to verify the range proofs and disjunctive proofs for all organizations, but these two proofs are computed sequentially.

C. Writing FabZK Applications

Writing applications in FabZK is similar to that in the Hyperledger Fabric. A FabZK application is comprised of application chaincode and client code: the former is installed on the endorser nodes, and the latter on off-chain nodes.

When an application chaincode is instantiated on a channel, its *init* function initializes the tabular structure of the public ledger for each organization. Values such as organization name (or ID), public key and initial asset amount can be loaded from the channel's genesis block. The *init* function calls the `ZKPutState` API to create the first row on the public ledger.

The application chaincode needs to support three chaincode methods: *transfer*, *audit* and *validation*. All of them accept input parameters from the client code. The *transfer* method calls the `ZkPutState` API to create a row with columns of $\langle \text{Com}, \text{Token} \rangle$ tuples on the public ledger. The *validation* and *audit* methods invoke the two-step validation through their underlying FabZK APIs. The *validation* method runs twice to call the `ZkVerify` API, validating two sets of NIZK proofs respectively. The *audit* method calls the `ZkAudit` API to compute $\langle \text{RP}, \text{DZKP}, \text{Token}', \text{Token}'' \rangle$ quadruples. Note that the *audit* chaincode method can be invoked periodically (e.g., once a week) to provide automated auditing.

Developers write client code to access the private ledger. To prepare the input transaction specification for the *transfer* chaincode method, the client code retrieves the current assets on the private ledger via the `PvlGet` API, and calls the `GetR` API to obtain a set of random numbers. After being notified of a new arrived transaction *tid*, the client code retrieves information from its private ledger to check

whether its organization is involved. If involved, it appends a new transaction row in the private ledger with *tid* and the transfer amount via the `PvlPut` API. The client code can also invoke the *validation* chaincode method with the transfer amount, the organization's secret key, and remaining assets as input. Based on the returned result, the client code updates the valid fields for that row of its private ledger.

A Sample Application: We build an over-the-counter stock trade application to demonstrate the methodology above. This application allows organizations to exchange assets between each other on a Fabric channel [35]. Client code of this application contains about 1200 lines of code in NodeJS, while chaincode contains about 1000 lines of code in Go.

A single asset exchange transaction requires two chaincode invocations. First, the sender informs the receiver of the upcoming transaction's unique identifier out of band and uses the client code to invoke the *transfer* chaincode method on its endorsing peer. Next, all organizations invoke the *validation* chaincode method to verify *Proof of Balance* and *Proof of Correctness* of the incoming transaction.

While the transactions are being submitted to Fabric, they are being audited with the other three NIZK proofs: Each organization scans the rows in its private ledger. If a row is verified during the asset exchange phase and the organization is the spending transaction, it invokes the `audit` chaincode method to create range proofs and disjunctive proofs for the transaction. Then, the auditor and other organizations can verify *Proof of Assets* (or *Proof of Amount*) and *Proof of Consistency* for the transaction. The auditing process is triggered at every 500 transactions.

Note that while auditing can identify invalid transactions, this process often lags behind the transactions. So invalid transactions can still occur until they are rejected. In practice, the consortium of participating organizations should agree on certain business rules to penalize the violations of the range proof and disjunctive proof. This logic, however, is out of scope for this paper, so it is not implemented in our sample application.

VI. Performance Evaluation

In this section, we evaluate the performance of the FabZK system. We aim to address the following aspects: (1) the efficiency of FabZK's cryptographic algorithms, compared to other alternatives, and (2) the overhead introduced to an application when using the privacy and audit functionalities provided by FabZK.

We implement FabZK on top of the Hyperledger Fabric version 1.3.0. The chaincode APIs are written in Go and the client APIs written in NodeJS. FabZK uses the elliptic curve `secp256k1` of the *btcec* library to compute commitments. Our range proofs are based on the protocol from Bulletproofs [31]. The disjunctive proofs use a modified version of Chaum-Pedersen proofs [32] with two non-interactive Σ-protocols [33]. Since these are proven cryptographic primitives, we focus only on the evaluation of their performance, while referring to the references above if the readers are interested in knowing their effectiveness in terms of privacy protection. All evaluations are run on Ubuntu 16.04 VMs provisioned on IBM Cloud.

Table II: Time (in ms) in running cryptographic algorithms by libsnark and FabZK for various numbers of organizations.

# of orgs	Data encryption		Proof generation		Proof verification	
	libsnark	FabZK	libsnark	FabZK	libsnark	FabZK
1	185.6	0.2	193.3	150.1	5.1	2.0
4	186.4	0.6	195.5	158.8	5.7	2.6
8	188.4	0.8	196.4	169.0	6.6	3.9
12	195.2	1.4	195.6	224.9	5.7	4.3
16	194.9	1.8	199.1	313.1	7.2	7.7
20	195.5	2.0	196.4	448.7	9.8	9.2

A. Algorithm Performance

We built a micro-benchmark to evaluate the performance of our privacy-preserving and auditing algorithms and compare them with a state-of-the-art approach, zk-SNARKs. For FabZK, we measure the run time for data encryption, i.e., computing $\langle \text{Com}, \text{Token} \rangle$ tuples, for generating NIZK proofs (i.e., $\langle \text{RP}, \text{DZKP}, \text{Token}', \text{Token}'' \rangle$ quartets), and for verifying the five proofs. These are the three key functions implemented in FabZK's chaincode APIs and are the major contributors to FabZK's overhead. As comparison, we also evaluate zk-SNARKs using libsnark [36], a library that implements a zero-knowledge verification scheme and has been used by other blockchain systems such as Zerocash [12]. Libsnark follows a similar design pattern of data encryption (through key generation), proof generation, and proof verification. Libsnark is implemented in C++.

We run the micro-benchmark on a VM with eight 2.10GHz cores. We vary the number of organizations from 1 to 20. For each setting, we collect data for 100 runs. In each run, the system processes 128 bytes of data (including transfer amount, private key, asset, etc.) for every organization.

As shown in Table II, FabZK outperforms *libsnark* in both data encryption and proof verification. Its proof generation has increasingly more overhead, as the number of organizations increases. *libsnark* has almost constant proof generation time ($\sim$196ms), because it only needs to generate one set of NIZK proofs for each transaction. In contrast, in FabZK's public ledger, each row consists of encrypted data for all organizations. Hence, as the number of organizations increases, it takes longer for FabZK to generate the proofs. From Table II, one can also see that the multithreaded implementation of proof generation is effective. The latency increase is moderate until the system supports more than 8 organizations, as the node used in our experiment only has 8 cores. In our future work, we will further improve FabZK's performance by exploring cross-node job scheduling schemes.

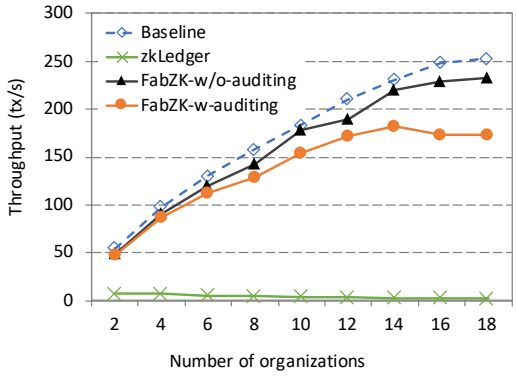

Figure 5: Throughput of asset exchange transactions for the prototype using native Fabric APIs (baseline), zkLedger, and FabZK's APIs with and without auditing (higher is better).

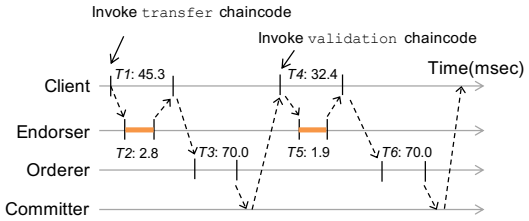

Figure 6: Timeline of an asset transfer transaction of the sample application with 8 organizations. The transaction involves two chaincode invocations: transfer (*T1*) and validation (*T4*). The duration of `ZkPutState` (*T2*) and `ZkVerify` (*T5*) are highlighted on the endorser's axis. The orderer spends about 70ms (*T3* and *T6*) in creating the block committed to the public ledger.

B. Application Performance

Next, we evaluate the overhead introduced by FabZK's privacy and audit functionalities. We use the sample application described in Section V-C for this evaluation.

Testbed: We deploy the sample application in a Hyperledger Fabric network, where each organization owns one peer node acting as its endorser and committer, and one certificate authority (CA) node. We setup a Kafka-based ordering service with 3 ZooKeeper nodes, 4 Kafka brokers, and one Fabric orderer. The orderer node creates blocks using the default configuration: 2 second batch timeout and $\leq$10 transactions per block. We group 5 octa-core VMs into a docker swarm cluster and then provision all Fabric components as containers in the cluster. Peer nodes and CA nodes of all organizations are evenly distributed to 4 VMs and the ordering service nodes are on the other VM.

Throughput Evaluation: We compare the throughput of the sample application running on three systems: FabZK, zkLedger,[2] and the native Fabric (i.e., the baseline). In this experiment, all organizations generate transactions concurrently, and each organization submits 500 transactions sequentially. A round of auditing is triggered when the ledger accumulates 500 new transactions. The results are shown in Figure 5.

We observe that FabZK's throughput scales similarly to the baseline. Without turning on audit, FabZK introduces only 3% to 10% throughput degradation, compared to the baseline. With audit turned on for every 500 transactions, FabZK's throughput overhead becomes 3% to 32%, compared to the baseline. Apparently, the additional overhead of computing and verifying the range and disjunctive proofs is quite significant. In practice, however, this can be mitigated by carefully selecting the audit frequency, especially during peak hours of operation.

Compared to zkLedger, the efficiency of FabZK is obvious. Its throughput with (without) auditing is 5 (5) to 189 (235) times that of zkLedger. This is expected, because transactions in zkLedger are validated and committed sequentially, while FabZK benefits from the parallelized execution described earlier.

Latency Evaluation: Figure 6 illustrates the timing of each step during an asset exchange transaction, without auditing being triggered. From the application's perspective, it takes about 45.3ms and 32.4ms to run a *transfer* chaincode method and a *validation* method, respectively. The run time of `ZkPutState` (2.8ms) includes 0.8ms of computing $\langle \text{Com}, \text{Token} \rangle$ tuples and 2ms of serializing the tuples to byte stream and writing it to the peer's transient data store. The run time of `ZkVerify` (1.9ms) includes 0.5ms of verifying *Proof of Balance* and *Proof of Correctness*, and 1.4ms of serializing and writing them to the peer's data store. In addition, the orderer node often waits to batch-process several transactions in a single block. Compared to the end-to-end transaction latency in Fabric, the absolute overhead of the FabZK APIs is relatively small: `ZkPutState` and `ZkVerify` contribute to less than 10% of the overall latency, while more than 90% of latency is caused by node-to-node communications, serialization/deserialization, block validation, I/O to the ledger, etc.

Next, we evaluate the audit latency, i.e., computing and verifying the range proofs and disjunctive proofs for a transfer row on the public ledger. In particular, we study the effect of the number of CPU cores in a peer node on the performance of FabZK's chaincode APIs: `ZkAudit` and `ZkVerify`. Figure 7 plots the latency of `ZkAudit` and `ZkVerify` for a 4-organization network, as the number of CPU cores increases from two to eight. For `ZkAudit`, using peer nodes with 4 and 8 CPU cores improves its performance by 50% and 90%, respectively, compared to using 2 cores. This performance gain is due to the parallelized computation of range proofs and disjunctive proofs (Section V-B). How-

[2]We implement a prototype of zkLedger on top of the Fabric architecture, too. Our prototype uses the BulletProofs instead of Borromean ring signatures to generate/validate range proofs for zkLedger. This change can only improve the throughput.

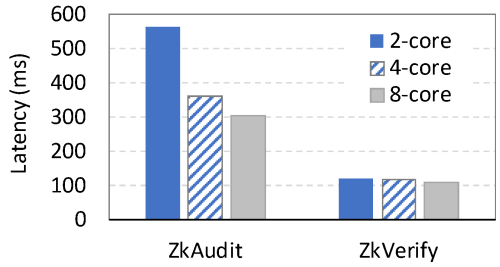

Figure 7: Latency of running `ZkAudit` and `ZkVerify` on VMs with different number of CPU cores.

ever, the improvement diminishes from 4 cores to 8 cores, since the chaincode only needs to spawn 4 threads for the 4 organizations. We also observe that parallelized processing has minimal impact on the performance of `ZkVerify`, as the computation is less intensive for this verification.

VII. Related Work

Confidential Transactions: Mechanisms for supporting confidential transactions have been studied extensively previously. To conceal balances and transaction values, publicly-verifiable cryptographic commitment schemes have been used to allow pseudonymous transfer of assets [14], [15], [19], [20], [37], [38].

To prevent double spending, work has also been done to show proof of assets using range proofs. For example, the Borromean ring signature [39] is widely used for range proofs in [14], [15], [19], [20], [40]. However, such range proof's overhead is significant: a transaction with two outputs and 32 bits of precision requires 5 KiB of range proof [31]. To reduce the size of range proofs, zero-knowledge Succinct Non-interactive ARgument of Knowledge proofs (zk-SNARKs) are used [12], [41], [42], [43], [44], [45], [46], [47], [48]. Even though these approaches can provide short-sized range proofs, they require an expensive trusted setup. Recently, an inner-product range proof has been proposed in Bulletproofs [31] with short proof size, and linear proving and verification time.

FabZK uses Pedersen commitments [14] and the inner-product range proofs [31] to achieve efficient confidential transactions.

Anonymized Transactions: Previous work has anonymized the transaction participants to conceal the transaction graph using identity mixes, oblivious RAM, or tabular structured ledgers. Examples of using the identity-mix approach in current cryptocurrency and blockchain systems are [49], [50], [51], [52], [53], [54], [55], [56]. Although mixes can protect participating users, it provides partial anonymity. Mix-type approaches are vulnerable to adversary tools, such as Coinjoin Sudoku [57], that can identify users within a transaction by correlating transaction outputs and inputs [58].

Solidus [19] obscures the transaction graph via publicly verifiable oblivious RAM machines (PVORM). A PVORM provides users a map from logical memory addresses to remote physical addresses. It provides the confidentiality of transaction graph and transaction details by obscuring memory access patterns. However, Solidus only works on bank-intermediated networks, which means it can only hide the information of bank's users but still exposes the transaction graph among banks (or organizations).

Most related to this work, zkLedger [20] uses a tabular structured ledger to conceal the transaction graph. In each transaction, zkLedger computes the commitments for all the organizations. By adding extra indistinguishable commitments in each transaction, zkLedger hides the identities of senders and receivers, thus concealing the transaction graph. However, zkLedger requires auditors and every participant to actively validate each transaction before this transaction is accepted to the ledger, which inevitably increases the latency and reduces the throughput.

In FabZK design, We adopt the tabular structured ledger from zkLedger, but develop additional proofs and validation mechanisms to boost audit performance.

VIII. Conclusion

Data privacy and confidentiality are critical for peer-to-peer transactions in blockchain systems. Although Hyperledger Fabric prohibits unidentified peers from accessing channel resources, transaction data are exposed to all channel participants. To overcome this limitation, we present FabZK, an extension to Fabric that supports auditable privacy-preserving smart contracts via well-constructed and verifiable cryptographic primitives, including Pedersen commitments and non-interactive zero-knowledge proofs. FabZK provides a set of APIs for both client code and chaincode to achieve on-demand, automated validation. We have implemented FabZK on Fabric v1.3.0, evaluated its performance against other, state-of-the-art approaches (i.e., zk-SNARKs, zkLedger). Our micro-benchmarking results show that the cryptographic primitives used by FabZK outperform those by zk-SNARKs in generating and verifying non-interactive zero-knowledge proofs. We have also demonstrated a sample application using FabZK APIs. Evaluations on its performance show that FabZK enables auditable privacy-preserving transactions at the cost of 3% to 32% throughput degradation and less than 10% latency increase, compared to the native Fabric system. FabZK achieves throughput up to 180 times higher than zkLedger.

Acknowledgments

We would like to thank Alysson Bessani, our shepherd, and the anonymous reviewers for their insightful feedback and valuable comments.

APPENDIX

Range Proof: An inner-product range proof in BulletProofs [31] takes a user specified u_{RP} and r_{RP} as input, and generates a proof $\text{RP} = rp$, including 1) a Pedersen commitment $\text{Com}_{\text{RP}} = rp.\text{Com} = com(u_{\text{RP}}, r_{\text{RP}})$, 2) two Pedersen vector commitments $rp.\vec{A}$, $rp.\vec{S}$ with a binding value $rp.\mu$, 3) an inner-product of two linear vector polynomials denoted by $rp.\hat{t}$ with a binding value $rp.\tau$ and an inner-product proof $rp.\text{IPP}$, 4) two Pedersen commitments to the two coefficients of $rp.\hat{t}$ denoted by $rp.T_1$, $rp.T_2$, and 5) three challenges $rp.C_x$, $rp.C_y$, $rp.C_z$. To prevent modular wraparound, i.e., $com(u, r) = com(u+p, r)$, p is the prime order of the cyclic group $\mathbb{G}$, we specifically prove that $\sum_{i=1}^{m} u_i \in [0, 2^t)$ for some small integer t. In our implementation, we set $t = 64$.

Disjunctive Zero-knowledge Proof: A non-interactive variant of the Chaum-Pedersen zero-knowledge proofs for the transaction tx_m is represented as:

$$\begin{aligned}\text{DZKP} = \text{ZK}_1(g_1^{x_1}, y_1^{x_1} \wedge g_1^{w_1}, y_1^{w_1}, chall_1, resp_1) \\ \wedge \text{ZK}_2(g_2^{x_2}, y_2^{x_2} \wedge g_2^{w_2}, y_2^{w_2}, chall_2, resp_2),\end{aligned} \tag{7}$$

where $\text{ZK}(g^x, y^x \wedge g^w, y^w, chall, resp)$ represents a non-interactive Σ-protocol [33] to prove the knowledge of the secret key sk (i.e., $x_1 = \text{sk}$) or the knowledge of random numbers (i.e., $x_2 = r - r_{\text{RP}}$), $g_1^{x_1}$ and $g_2^{x_2}$ are two generalized Schnorr proofs [59], w_1 and w_2 are two random numbers, $chall_1 = \text{Hash}(\text{Token}')$, $chall_2 = \text{Hash}(\text{Token}'')$, $resp_1 = w_1 + x_1 chall_1$, $resp_2 = w_2 + x_2 chall_2$, $g_1 = (\prod_{i=0}^{m} \text{Com}_i)/\text{Com}_{\text{RP}}$, $y_1 = (\prod_{i=0}^{m} \text{Token}_i)/\text{Token}'$, $g_2 = \text{pk}$, $y_2 = \text{Token}/\text{Token}''$.

To verify whether such a DZKP in Equation (7) is valid, the verifier first checks whether $g^{resp} = (g^x)^{chall} g^w$ and then $y^{resp} = (y^x)^{chall} y^w$ for the two non-interactive Σ-protocols ZK_1 and ZK_2 [59].

Note that, Token' for the spending organization and Token'' for other organizations must be $\text{pk}^{r_{\text{RP}}}$. This is because $(s/\text{Com}_{\text{RP}})^{\text{sk}_{\text{spend}}} = t/\text{Token}'$ holds for the spending organization while $(\text{Com}/\text{Com}_{\text{RP}})^{\text{sk}_{\text{other}}} = t/\text{Token}''$ holds for other organizations. Moreover, Token' and Token'' guarantee that ${g_1}^{\text{sk}} = y_1$ and ${g_2}^{r-r_{\text{RP}}} = y_2$.

Proof: For the spending organization,

$$\begin{aligned}{g_1}^{\text{sk}} &= (\prod_{i=0}^{m} \text{Com}_i)^{\text{sk}}/(\text{Com}_{\text{RP}})^{\text{sk}} \\ &= (g^{\sum_{i=0}^{m} u_i} h^{\sum_{i=0}^{m} r_i})^{\text{sk}}/(g^{\sum_{i=0}^{m} u_i} h^{r_{\text{RP}}})^{\text{sk}} \\ &= (h^{\sum_{i=0}^{m} r_i})^{\text{sk}}/(h^{r_{\text{RP}}})^{\text{sk}} = t/\text{pk}^{r_{\text{RP}}} = y_1, \\ {g_2}^{r-r_{\text{RP}}} &= \text{pk}^{r-r_{\text{RP}}} = (\prod_{i=0}^{m} \text{Com}_i)^{\text{sk}}/(\text{Com}_{\text{RP}})^{\text{sk}} \\ &= (s/\text{Com}_{\text{RP}})^{\text{sk}} = \text{Token}/\text{Token}'' = y_2.\end{aligned}$$

As for other organizations,

$$\begin{aligned}{g_1}^{\text{sk}} &= (s/\text{Com}_{\text{RP}})^{\text{sk}} = t/\text{Token}' = y_1, \\ {g_2}^{r-r_{\text{RP}}} &= \text{pk}^{r-r_{\text{RP}}} = \text{Token}/\text{Token}'' = y_2.\end{aligned}$$

■

A DZKP allows the prover to create a real proof using real values (e.g., sk) and a fake proof using fake values (e.g., an arbitrary number), and the verifier to validate DZKP by itself without distinguishing between real proof and fake proof. When a prover knows the secret key of the spending organization sk_{spend} but not others' sk_{other}, sk in Equation (5) is an arbitrary random number but not sk_{other}. To conceal the transaction graph, the sk in Equation (6) is an arbitrary random number other than sk_{spend}.

Proof: Suppose sk is the spending organization's secret key sk_{spend}. Substitute sk with sk_{spend} in Equation (6), we have:

$$\begin{aligned}\text{Token}'' &= \text{Token} \cdot (s/\text{Com}_{\text{RP}})^{\text{sk}_{\text{spend}}} = \text{Token} \cdot {g_1}^{\text{sk}_{\text{spend}}} \\ &= \text{Token} \cdot y_1 = \text{Token} \cdot t/\text{Token}'.\end{aligned} \tag{8}$$

Equation (8) shows a linear relationship among Token, Token', Token'', and t. A linear relationship reveals the identity of the spending organization through trivial computation by an observer. Therefore, $\text{sk} \neq \text{sk}_{\text{spend}}$ in Equation (6). ■

REFERENCES

[1] "British Airways data breach: Russian hackers sell 245,000 credit card details," https://www.theweek.co.uk/96327/british-airways-data-breach-how-to-check-if-you-re-affected.

[2] "Equifax Data Breach Impacts 143 Million Americans," https://www.forbes.com/sites/leemathews/2017/09/07/equifax-data-breach-impacts-143-million-americans/.

[3] "Facebook Security Breach Exposes Accounts of 50 Million Users," https://www.nytimes.com/2018/09/28/technology/facebook-hack-data-breach.html.

[4] S. Gressin, "The Marriott data breach," https://www.consumer.ftc.gov/blog/2018/12/ marriott-data-breach, 2018.

[5] "Uber Settles Data Breach Investigation for $148 Million," https://www.nytimes.com/2018/09/26/technology/uber-data-breach.html.

[6] "Verizon partner data breach exposes millions of customer records," https://www.theverge.com/2017/7/12/15962520/verizon-nice-systems-data-breach-exposes-millions-customer-records.

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf."

[8] "Ripple," https://ripple.com/.

[9] "Digital Asset," https://www.digitalasset.com/.

[10] "Stella," https://www.stellar.org/.

[11] "Zcash," https://z.cash/.

[12] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *IEEE Symposium on Security and Privacy*, 2014, pp. 459–474.

[13] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2017.

[14] G. Maxwell, *Confidential Transactions*, https://people.xiph.org/~greg/confidential_values.txt.

[15] A. Poelstra, A. Back, M. Friedenbach, G. Maxwell, and P. Wuille, "Confidential assets," in *4th Workshop on Bitcoin and Blockchain Research*, April 2017.

[16] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *EuroSys*, 2018, pp. 30:1–30:15.

[17] "JPMC Quorum," https://www.jpmorgan.com/global/Quorum.

[18] "Private and confidential transactions with Hyperledger Fabric," https://developer.ibm.com/tutorials/cl-blockchain-private-confidential-transactions-hyperledger-fabric-zero-knowledge-proof/.

[19] E. Cecchetti, F. Zhang, Y. Ji, A. Kosba, A. Juels, and E. Shi, "Solidus: Confidential distributed ledger transactions via pvorm," in *ACM CCS*, 2017, pp. 701–717.

[20] N. Narula, W. Vasquez, and M. Virza, "zkledger: Privacy-preserving auditing for distributed ledgers," in *Symposium on Networked Systems Design and Implementation*, 2018, pp. 65–80.

[21] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology — CRYPTO '91*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 129–140.

[22] CPA Canada and American Institute of CPAs, "Blockchain technology and its potential impact on the audit and assurance profession," 2017.

[23] "Tendermint," https://tendermint.com.

[24] "Chain," https://chain.com/.

[25] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference*, 2014, pp. 305–320.

[26] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, 2011, pp. 245–256.

[27] "Go SDK for Hyperledger Fabric," https://github.com/hyperledger/fabric-sdk-go.

[28] "Java SDK for Hyperledger Fabric," https://github.com/hyperledger/fabric-sdk-java.

[29] "Node.js SDK for Hyperledger Fabric," https://github.com/hyperledger/fabric-sdk-node.

[30] M. Blum, A. De Santis, S. Micali, and G. Persiano, "Noninteractive zero-knowledge," *SIAM J. Comput.*, vol. 20, no. 6, pp. 1084–1118, Dec. 1991.

[31] B. Bnz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *IEEE Symposium on Security and Privacy*, May 2018, pp. 315–334.

[32] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Proceedings of Advances in Cryptology*, 1992, pp. 89–105.

[33] R. Cramer, I. Damgård, and B. Schoenmakers, "Proofs of partial knowledge and simplified design of witness hiding protocols," in *14th Annual International Cryptology Conference on Advances in Cryptology*, 1994, pp. 174–187.

[34] "Protobuf," https://developers.google.com/protocol-buffers/.

[35] "What is Over-The-Counter - OTC," https://www.investopedia.com/terms/o/otc.asp.

[36] "libsnark: a C++ library for zkSNARK proofs," https://github.com/scipr-lab/libsnark/.

[37] D. Lukianov, "Compact confidential transactions for bitcoin," http://voxelsoft.com/dev/cct.pdf, 2015.

[38] T. Ruffing and G. Malavolta, "Switch commitments: A safety switch for confidential transactions," in *Financial Cryptography Workshops*, ser. Lecture Notes in Computer Science, vol. 10323. Springer, 2017, pp. 170–181.

[39] G. Maxwell and A. Poelstra, "Borromean ring signatures," 2015.

[40] S. Noether and A. Mackenzie, "Ring confidential transactions," *Ledger*, vol. 1, pp. 1–18, 2016.

[41] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: verifying program executions succinctly and in zero knowledge," in *CRYPTO (2)*, ser. Lecture Notes in Computer Science, vol. 8043. Springer, 2013, pp. 90–108.

[42] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *USENIX Security*, San Diego, CA, 2014, pp. 781–796.

[43] N. Bitansky, A. Chiesa, Y. Ishai, O. Paneth, and R. Ostrovsky, "Succinct non-interactive arguments via linear interactive proofs," in *Theory of Cryptography*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 315–333.

[44] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *Advances in Cryptology EUROCRYPT*, 2013, pp. 626–645.

[45] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *Advances in Cryptology - ASIACRYPT 2010*, 2010, pp. 321–340.

[46] H. Lipmaa, "Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments," in *Theory of Cryptography*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 169–189.

[47] ——, "Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes," in *ASIACRYPT (1)*, ser. Lecture Notes in Computer Science, vol. 8269. Springer, 2013, pp. 41–60.

[48] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE Symposium on Security and Privacy*, May 2013, pp. 238–252.

[49] J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten, "Mixcoin: Anonymity for bitcoin with accountable mixes," in *Financial Cryptography and Data Security*. Springer, 2014, pp. 486–504.

[50] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *NDSS*, 2017.

[51] G. Maxwell, "CoinJoin: Bitcoin privacy for the real world," https://bitcointalk.org/index.php?topic=279249.

[52] M. Rosenfeld, "Using mixing transactions to improve anonymity," https://bitcointalk.org/index.php?topic=54266.

[53] T. Ruffing, P. Moreno-Sanchez, and A. Kate, "Coinshuffle: Practical decentralized coin mixing for bitcoin," in *ESORICS (2)*, ser. Lecture Notes in Computer Science, vol. 8713. Springer, 2014, pp. 345–364.

[54] L. Valenta and B. Rowan, "Blindcoin: Blinded, accountable mixes for bitcoin," in *Financial Cryptography Workshops*, ser. Lecture Notes in Computer Science, vol. 8976. Springer, 2015, pp. 112–126.

[55] E. Z. Yang, "Secure multiparty Bitcoin anonymization," http://blog.ezyang.com/2012/07/secure-multiparty-bitcoin-anonymization/.

[56] J. H. Ziegeldorf, F. Grossmann, M. Henze, N. Inden, and K. Wehrle, "Coinparty: Secure multi-party mixing of bitcoins," in *ACM CODASPY*, 2015, pp. 75–86.

[57] "Coinjoin Sudoku," http://www.coinjoinsudoku.com/.

[58] "Blockchains SharedCoin Users Can Be Identified, Says Security Expert," https://www.coindesk.com/blockchains-sharedcoin-users-can-identified-says-security-expert.

[59] J. Camenisch, A. Kiayias, and M. Yung, "On the portability of generalized schnorr proofs," in *Conference on Advances in Cryptology - EUROCRYPT*, vol. 5479, 2009, pp. 425–442.