LedgerView: Access-Control Views on Hyperledger Fabric

Pingcheng Ruan
National University of Singapore
Singapore
e0079643@u.nus.edu

Yaron Kanza
AT&T Chief Data Office
New Jersey, USA
kanza@research.att.com

Beng Chin Ooi
National University of Singapore
Singapore
ooibc@comp.nus.edu.sg

Divesh Srivastava
AT&T Chief Data Office
New Jersey, USA
divesh@research.att.com

ABSTRACT

We present LedgerView—a system that adds access control views to permissioned blockchains. The approach is motivated by an AT&T application of tracking refurbished devices. A blockchain is a decentralized tamper-resistant ledger managed by a group of peers. It is used in many applications for storing and sharing *sensitive* information, e.g., monetary transactions, health records, personal documents, etc. But in blockchain, all the peers see all the stored transactions, while in some applications, access to sensitive information should be limited, that is, concealed from peers and users who do not have proper access permissions. In database management systems, sets of records that are visible to some users and concealed from others are defined by views, but existing blockchain systems lack such access-control capabilities. Thus, in this paper, we introduce access-control views for Hyperledger Fabric. We present two types of views—*revocable* and *irrevocable*, according to whether access to sensitive information can or cannot be revoked. We explain how to implement the two types of view by using cryptographic hash functions and encryption keys, and we show how to support Role-Based Access Control (RBAC). Experiments with supply chain transactions illustrate the incurred costs of the views in LedgerView, including latency, transaction rate and storage overhead.

CCS CONCEPTS

• **Information systems** → **Database views**; • **Security and privacy** → *Access control*.

KEYWORDS

Blockchain, access control, views, RBAC, privacy, supply chain

ACM Reference Format:
Pingcheng Ruan, Yaron Kanza, Beng Chin Ooi, and Divesh Srivastava. 2022. LedgerView: Access-Control Views on Hyperledger Fabric. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526046

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526046

1 INTRODUCTION

Nowadays, blockchain has many industrial applications. It was initially introduced as a tamper-proof decentralized ledger for prevention of *double spending* in cryptocurrencies like Bitcoin [39]. However, in recent years many new blockchain applications were developed, in a variety of areas, including supply chain management [21, 58], healthcare [16], identity management [22], finance [56], transportation [31], management of personal data [62], tracking IoT devices [8] and managing drones [18]. Blockchain has become an important tool for managing shared data in applications with no agreed-upon trusted entity that could store and manage all the data, or when data records should result from a consensus between organizations that do not fully trust each other [26].

Blockchain was initially designed as a transparent ledger, e.g., in cryptocurrencies all the users see all the transactions. Transparency is needed for preventing double spending—a case where a user pays with the same coins in two different transactions. However, nowadays blockchain has many uses, not just managing cryptocurrencies. In many applications, the information stored on the blockchain is sensitive and should be concealed from some of the users, including some or all of the peers who manage the blockchain. An example of that is a blockchain application for storing health records [27]. Such records contain sensitive private information and should not be revealed to unauthorized users. Access to the information should be granted to authorized users, but the set of authorized users could change over time, e.g., new healthcare workers may need access to health records that were stored before they were hired.

At AT&T, a blockchain-based proof of concept application has been built for tracking refurbished mobile devices. When refurbishing devices, parts that were taken from disposed devices may be used. To offer certified refurbished devices with repair services and warranty, the history of used parts should be recorded. Since parts are made by many manufacturers, used in devices of different companies and are refurbished in different laboratories, there is no single entity that tracks all the devices and parts. However, laboratories need to know the entire history of every part they use, manufacturers want to know where their parts are used if they need to provide warranty, and stores may need to know if the refurbished devices they are selling contain used parts. Thus, a ledger of parts and devices has been built on top of Hyperledger Fabric.

The tracking of refurbished devices should not breach business confidentiality, e.g., a manufacturer should only track parts it produced, a repair lab should not have access to information on parts used by other labs, etc. Existing features of Hyperledger Fabric

like channels and private data collections do not provide sufficient access control for this application because a priori it is unknown who will be allowed to track a part.

In database management systems, an access control module manages access permissions and prevents unauthorized users from accessing restricted data [24]. A similar system is needed for protecting sensitive information on a blockchain. The following example illustrates access-control views in a supply chain management system, similar to the refurbished-device tracking application.

Example 1.1. Fig. 1 presents a data supply chain with 2 manufacturers, 3 warehouses, 2 delivery services and several shops. The manufacturers, shops and warehouses store information about items they produced or handled. Every delivery transaction is visible to all the entities who handled the item. Since there is no single entity that is trusted to manage all the data, the information is kept on a blockchain. However, without any access control, all the information is visible to every user. Such a solution does not provide business confidentiality. Each manufacturer, warehouse, shop and delivery service should only be able to see the information pertaining to items they processed, delivered or received.

Access-control views are used in database management systems for specifying access permissions, but blockchains lack them. For instance, consider in Example 1.1 views V_{M1} and V_{M2} of all the delivery records of items produced by Manufacturer 1 and Manufacturer 2, respectively. Granting Manufacturer 1 (Manufacturer 2) access only to V_{M1} (V_{M2}) would satisfy the business confidentiality requirements. However, there are no access-control views in blockchains. Implementing such views on blockchain requires coping with transparency, immutability and lack of central control.

Access to a view is granted or revoked by users who are authorized to do so. In a centralized access-control system, a central system is responsible for enforcing the access-control policy. Guaranteeing that the access-control policy is enforced correctly in a decentralized environment is more challenging. In many applications, the peers who manage the blockchain cannot be fully trusted. Decisions regarding data are based on a consensus mechanism under the assumption that the majority of the peers are honest (trustworthy). But even a single dishonest peer can leak sensitive information. So, it is undesirable to allow all the peers to access sensitive information when not all of them can be trusted.

Another challenge when managing access control on blockchain is to revoke access permissions. The blockchain is immutable; so past records cannot be changed. Designing access control in which access can be revoked should cope with that. Immutability of the blockchain also makes updates of views a challenging task. If views are stored on the ledger, it is impossible to update the view records when the underlying data change. Thus, views on a blockchain should be designed for an append-only ledger.

In this paper, we present the LedgerView system for supporting access-control views. Our main contributions are as follows.

- Introducing access-control views for blockchain, where sensitive information can be hidden by either encryption or hashing, based on the user preferences.
- Implementing the access-control views and role-based access control (RBAC) on Hyperledger Fabric.

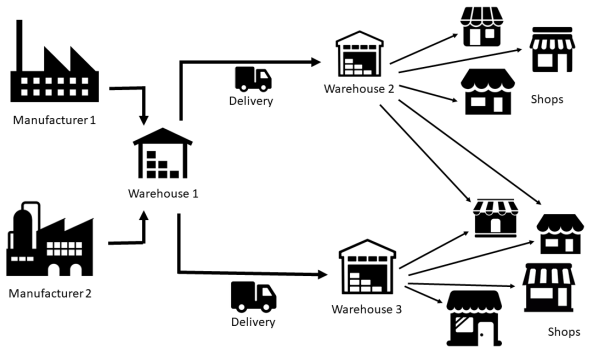

Figure 1: An illustration of a supply chain with 2 manufacturers, 3 warehouses, 2 delivery services and several shops.

- Showing that the methods support verifiable completeness and soundness—a user can verify that each view contains exactly the set of transactions it should consist.
- Presenting a modular system architecture to manage views, query them and verify their integrity.
- We implemented a workload generator for supply chain applications, for benchmarking the LedgerView system.
- Extensive experiments over a supply chain workload demonstrate the effectiveness of our system.

2 RELATED WORK

In this section, we survey related work.

Access control. Access control is a core component in many database management systems [11, 49]. Access control views have been studied extensively in the literature for different types of access control methods, e.g., role-based access control [48], and different types of data, including for relational database systems [14], XML [2, 30, 60], cloud databases [4], and so on. The methods we present in this paper are very different from those in the literature because we need to cope with the unique features of blockchain, including decentralized management, immutability and transparency.

Provenance on Blockchains. A blockchain is a secured ledger that establishes a decentralized consensus on the order of transactions. It provides an immutable storage of transaction history in a provenance-friendly manner—users can inspect the ledger to track item lineage. In a variety of areas, the provenance capabilities of blockchains are exploited, to support different business applications, including supply chains [33, 38], cloud data management [53], and Internet-of-Things [29, 54]. Typically, the above applications are limited to offline analysis. However, it is possible to expose the on-chain lineage to smart contracts, and by that allow developers to express business rules in which the logic of online transactions can be affected by lineage of related data items [45].

Cross blockchain transactions. The popularity of Decentralized Finance has motivated studies of *cross-chain swaps*. A cross-chain swap is an *atomic* transaction that consists of sub-transactions in two or more different ledgers. For example, a purchase of Bitcoins with Ether coins is a transaction composed of cryptocurrency transactions on both the Bitcoin and the Ethereum ledgers. The goal of

the swap protocol is to guarantee atomicity—the end result should be equivalent to a case where either all the sub-transactions are committed or none of them is. The protocol operates under the assumption that participants may arbitrarily deviate from the protocol for their best interest. This problem was formulated in [40], where the authors present a heuristic solution for two participants. Herlihy [28] generalizes the technique to multiple participants. Zakhary et al. [61] propose a further generalization by relaxing assumptions regarding network synchronicity and sequential execution. Notably, AHL [17] uses a 2-phase commit (2PC) protocol to coordinate cross-chain transactions, on top of byzantine protocols, and Caper [5] adds data confidentiality to swaps.

Blockchain channels. Blockchain channels allow limiting the information blockchain peers can access, by providing a private communication link between peers [55]. Transactions on a channel can only be viewed by peers that are members of that channel. Channels, however, are different from views in the following aspects. First, a transaction can be included in several views but only in one channel. In Example 1.1, there can be a view per each manufacturer, warehouse and delivery service. The same transaction could be in the views of a manufacturer, a warehouse and a delivery service. Second, there is no flexible way to grant and revoke access permissions when using channels. In channels, adding or removing members is similar to adding peers to a permissioned blockchain, and hence, it is limited. Third, channels lack access-permission rules that are based on attributes of the users or of the data records.

Private data collections. In Hyperledger Fabric, users can define a *private data* collection [10], to support privacy-demanding applications [35, 42]. In this setting, contract states are managed privately by the peers and only signatures of the data are stored on the blockchain. Thus, this setting cannot support irrevocable access permissions. In addition, the peers have access to contract states. This may cause a privacy breach in applications like tracking refurbished devices where peers should not access information on devices or parts they have not manufactured, processed or sold.

Access-control ledger. The use of blockchain for storing access control permissions was studied in several papers [16, 34, 36, 37]. The goal is to store on the blockchain access permissions or access requests for sensitive data or restricted systems. The sensitive data set is not stored on the blockchain itself. The blockchain is only a ledger of access permissions. These papers, however, do not provide blockchain views or any other flexible access control method where sets of records are defined using queries (the views) and sets of users with access permissions are specified by roles or sets of identifiers.

3 BLOCKCHAIN APPLICATION

This section presents the framework and the blockchain application.

Blockchain. Blockchain is a ledger managed by peers in a decentralized fashion. A blockchain ledger consists of a sequence $B_0, \ldots, B_m$ of blocks of transactions. Block B_0 is the *genesis block*. Any block B_i other than the genesis block contains the cryptographic hash of the previous block, that is, the hash of block B_{i-1}. We denote by $t_{i,1}, \ldots, t_{i,n_i}$ the transactions stored in block B_i. In cryptocurrencies like Bitcoin, transactions represent transfer of coins between *addresses* (aliases of users). In other blockchain applications, transactions can include financial information [56], health records [16], user identity information [22], IoT data [8], personal data [62], and records on parts of cellular devices.

A blockchain is managed by a group of peers. The peers create blocks which batch transactions and add them to the chain based on a consensus mechanism. There are two types of blockchain. In a *permissionless* blockchain, the set of peers may change arbitrarily—new peers can join the set or leave it at any time. In a *permissioned* blockchain, the set of peers is fixed or is determined by pre-defined rules. Consensus algorithms for permissionless blockchains include Proof of Work [39], Proof of Stake [32, 47], Proof of Location [19], Proof of Space [43], etc. Many consensus algorithms were developed or adopted for permissioned blockchains, including PBFT [13], Tendermint [6], and the recent Whatever-Voting [51] . For a comparison of consensus protocols, see [20, 26, 41, 50].

We denote by P the set of peers that manage the blockchain. We denote by U the *users*, a set that includes the blockchain peers, which create blocks, and people or applications that access the blockchain to read the content of blocks but without creating blocks. Each user $u \in U$ has a pair of public and private keys, denoted PubK_u and PrivK_u, in correspondence. This pair of keys can be based on RSA [44] or any other public-key cryptosystem.

Smart contract. In Bitcoin, transactions represent transfer of cryptocurrency between addresses. Only a limited set of operations can be performed in transactions, e.g., hashing or verification of cryptographic signatures, but many blockchain applications require a richer set of tools. Thus, blockchains like Ethereum and Hyperledger Fabric support *smart contracts* (*chaincode*)—code executed by the blockchain peers for allowing users to define arbitrary functions that are computed in a decentralized manner. The transactions are generalized into invocations of a smart contract that modifies internal states of the contract.

While smart contracts increase the utility of blockchains, they often consume computational resources and add complexity to the system. In blockchains that support smart contracts, a state can no longer be represented by transactions only. State values are maintained explicitly, synchronized among the mutually-distrusting peers and are part of the consensus. Typically, it is done using a Merkle tree. A Merkle tree is a rooted labeled binary tree whose leaves store values and the label of each inner node is the hash of the concatenation of the labels (or values) of its children.

For state management, the blockchain protocols specify a deterministic tree-building procedure with contract states as the leaves of the Merkle tree [59]. The root hash of the Merkle tree serves as the state digest, and it is included in each block header, as a snapshot of the states. Consensus regarding a block in the chain is a consensus on the digest in the block header, and hence, a consensus on the digest of states. A Merkle path from the root hash to a state stored in a leaf provides an integrity proof for a contract state. Note that the size of values stored in the leaves of the Merkle tree does not affect the size of the digest stored on the ledger.

In many systems, the data stored on the ledger and the states of smart contracts are maintained in a local database, where the Merkle tree is used for validating the integrity of data and of answers to queries posed to the local database. By using the local database,

a query over the ledger is computed efficiently. In Hyperledger Fabric, LevelDB or CouchDB are used for storing the state database and answering queries that are posed to the blockchain.

Sensitive information. A blockchain may include sensitive information such as health records, personal documents, financial data and location data. Each transaction t_{ij} has two parts, a non-secret part denoted $t_{ij}[N]$ and a secret part, denoted $t_{ij}[S]$. The non-secret part is visible to all the users and can be used in the consensus protocol, to determine which transactions to include in the blockchain. The access to the secret part should be limited and granted to users according to the access permissions.

Example 3.1. Suppose that in the supply chain of Example 1.1 every transfer of items is recorded on the blockchain, e.g., transfers from a manufacturer to a warehouse or from a warehouse to a shop. Each transaction includes shipment data. Details like shipment number, date and the involved entities are included in the non-secret part. The type of items, amount of delivered items and price are confidential and included in the secret part.

All the transactions on the blockchain are visible to all the peers, and thus, the secret part is concealed by either encrypting it or storing on the blockchain just the cryptographic hash of the secret. We elaborate on that in the following sections. The encryption of a secret part can be by a symmetric encryption key like AES [15] or TDEA [9], or using a public key cryptosystem like RSA [44]. As a secure cryptographic hash, SHA-256 can be used [25]. We assume that each transaction has a unique identifier, and we denote by tid_{ij} the identifier of transaction t_{ij}. Accordingly, a transaction is a 3-tuple $(tid_{ij}, t_{ij}[N], t_{ij}[S])$ of identifier, non-secret part and secret part.

Granting and revoking access. Each transaction is added to the blockchain by some user $u \in U$. We assume that u knows the secret part or the key to decrypt it, for all the transactions u adds to the blockchain. *Granting access* to user u' for transaction t_{ij} is providing the means for u' to see the secret part of t_{ij}. *Revoking access* means that the permission to see the secret part is repealed so that a request to access the secret information would be denied.

To grant access to sets of records, we use *access-control views*. A view specifies a set of records and access permissions specify the users who are allowed to read these records. There are two types of access-control permissions over blockchain, *revocable* and *irrevocable*. A revocable access permission is similar to standard access control in database management systems. An irrevocable permission cannot be revoked and it is unique to blockchains. We present views with revocable and irrevocable permissions.

Verifiable soundness and completeness. Given a database D and a query Q over D, a view V can be defined as $Q(D)$, that is, the result of applying Q to D. If V is a maintained view, the following two properties are required. *(1) Soundness*: V is sound if $V \subseteq Q(D)$, and *(2) completeness*: V is complete if $V \supseteq Q(D)$.

In a decentralized environment, it is not always possible to guarantee soundness and completeness. Thus, we relax the requirement to the ability to verify soundness and completeness of the view at any given time T. *Verifiable soundness* and *verifiable completeness* at T are the ability of users with access permissions to V to verify the soundness and completeness of V according to all the transactions that were added to the underlying dataset until time T.

View definition. Typically, views are defined using a query. In a *maintained view*, the query answer is stored and updated when there are changes in the underlying data. In an *unmaintained view*, the view query is executed when the view is invoked. For access control, we can store on the blockchain a maintained view and grant users access to it, or store on the blockchain only the information regarding who has access to data and provide the data when requested if access is allowed according to the blockchain.

A *view definition* is a predicate P_V over the non-secret part of transactions. A predicate P_V defines a view $V = \{t_{ij} \mid P_V(t_{ij}[N])\}$ with the set of transactions t_{ij} for which $P_V(t_{ij}[N])$ is true.

Example 3.2. Consider the supply chain in Example 1.1. Suppose that each shipment transaction has `from` and `to` attributes, to specify the entities in the shipment. A predicate $P(\texttt{to})$ = "Warehouse 1" specifies the set of all transactions that are delivered to Warehouse 1.

The view defined in Example 3.2 is local and does not connect transactions to one another, e.g., it cannot relate to the history of item deliveries. To address this, we add recursion to the view definition, in a datalog fashion [3]. Since datalog programs with recursion are a well known concept, we avoid providing formal definitions here, and only explain the main concepts.

A datalog *rule* contains a head and body and it is defined in a recursive way. Given a predicate $P(t)$ over transactions, a rule $Q(t) \leftarrow P(t)$ specifies that $Q(t)$ is true for all the transactions that satisfy predicate $P(t)$. In this case, $Q(t)$ is the head of the rule and $P(t)$ is the body of the rule.

A rule $Q(t) \leftarrow P_1(t), P_2(t), \ldots, P_m(t)$ specifies that Q is satisfied by transactions that satisfy the predicates $P_1, P_2, \ldots, P_m$. For a set of rules $Q(t) \leftarrow P_1(t), Q(t) \leftarrow P_2(t), \ldots, Q(t) \leftarrow P_k(t)$, the transactions satisfying $Q(t)$ are the union $\{t \mid P_1(t) \vee P_2(t) \vee \ldots P_k(t)\}$ comprising the set of transactions satisfying at least one of the rules. Datalog rules can be recursive. For example, $\texttt{Delivery}(t, \texttt{from}, \texttt{to})$ contains transactions t delivered between the two locations specified by `to` and `from`. The following set of rules

$$
\begin{aligned}
P_1(t, \texttt{from}, \texttt{to}) &\leftarrow \texttt{Delivery}(t, \texttt{from}, \text{“Warehouse 1”}) \\
P_1(t, X, Z) &\leftarrow \texttt{Delivery}(t, X, Y), P_1(t_1, Y, Z) \\
P(t) &\leftarrow P_1(t, X, Y)
\end{aligned}
$$

defines a predicate $P(t)$ of all the transactions that are part of a delivery to "Warehouse 1". The predicates can relate to any attribute of stored transactions or of blocks that contain them, e.g., transaction time, transaction identifier, block creation time, block identifier, etc. Similar queries help tracing parts for refurbished devices.

4 ACCESS-CONTROL VIEWS

In this section we show how to create views with revocable and irrevocable access permissions. For refurbished devices, information on a repurposed part or related to a warranty is irrevocable. Information on devices that have not been dismantled may be revocable.

In Hyperledger Fabric, encryption is used in channels and hash is used in private data collections. We explore both encryption-based and hash-based views, to provide equivalent revocable and irrevocable views for channels and private data collections. In all

the methods, we assume that the blockchain peers cannot be fully trusted, hence, the secret part of transactions is hidden from the peers. In this section, we elaborate on that.

4.1 Encryption-based Irrevocable Permissions

We present now encryption-based irrevocable access-control views, namely EI. In EI, the secret part of each transaction is stored encrypted and the decryption key is provided only to users who are granted access to the secret information.

Storing transactions. Initially, before adding a transaction t_{ij} to the blockchain, the user u who adds t_{ij} generates a new key K_{ij}, and conceals the secret part of t_{ij} by encrypting it with the key K_{ij}. Each transaction is encrypted with a unique symmetric encryption key. Transaction t_{ij} is stored as a 3-tuple $(tid_{ij}, t_{ij}[N], enc(t_{ij}[S], K_{ij}))$ of identifier, non-secret part and encrypted secret part.

Initially, only user u has the key to see the transaction. To grant access to the secret part of transactions, the keys should be disseminated to authorised users. This is done via access-control views.

Creating a view. An access-control view is a list of keys of the transactions it comprises. Suppose that user u wants to grant access to transactions $t_1, \ldots, t_n$. Let $K_1, \ldots, K_n$ be the keys of the transactions $t_1, \ldots, t_n$, in correspondence. To create an access-control view V, user u produces a new symmetric key K_V. The view V is stored on the blockchain as a list $enc([tid_1, K_1, \ldots, tid_n, K_n], K_V)$, that is, the encryption with the view key K_V of the keys $K_1, \ldots, K_n$ and the identifier of the corresponding transactions. The list is stored as a transaction t_V. Note that at this point only u knows K_V so no secret information is revealed yet. However, providing the key K_V to a user u' will allow u' to decrypt the content of t_V, acquire the keys of $t_1, \ldots, t_n$, and access the secret parts of these transactions.

Granting access to a view. To grant users $u'_1, \ldots, u'_m$ with access to a view V, the user u that created V can send the key to these users via a secured communication channel. Another option is to create and add to the blockchain a transaction that contains the encryption of K_V with the public keys of $u'_1, \ldots, u'_m$, that is, $enc(K_V, \mathsf{PubK}_{u'_1}), \ldots, enc(K_V, \mathsf{PubK}_{u'_m})$. Each user u'_i will be able to decrypt $enc(K_V, \mathsf{PubK}_{u'_i})$ using the private key $\mathsf{PrivK}_{u'_i}$. Any user who is not among $u'_1, \ldots, u'_m$ will not know the private key for decrypting the content of this dissemination transaction.

The blockchain is an append-only ledger; so all the transactions, including the view and the access to the view, are tamper-resistant and cannot be deleted. Hence, access permissions are irrevocable.

4.2 Encryption-based Revocable Permissions

We now describe revocable permissions for an encryption-based storage of transactions as in Section 4.1, namely ER.

Creating a view. To make the access permission revocable, the secret information should be disseminated in a way that allows denying access when permissions change. The secret information should not be revealed by an immutable blockchain transaction. Instead, the transaction keys should be provided to users with an access permission upon request, as long as the access permission was granted and has not been revoked.

Consider a revocable view V created by user u, for transactions $t_1, \ldots, t_n$. Let $u'_1, \ldots, u'_m$ be the users that should have access to the transactions in view V. The user u has access to all the keys of the transactions in V, but these keys are not irreversibly provided to the users $u'_1, \ldots, u'_m$. Instead, to implement revocable access permissions, the view has two parts, V_{ids} and V_{access}. The first part is the identifiers of the transactions in the view, i.e., $V_{ids} = \{tid_1, \ldots, tid_n\}$ is the list of identifiers of the transactions $t_1, \ldots, t_n$ contained in V. The V_{access} part is used for managing access to the view. User u creates a unique key K_V for the view. That key is disseminated to the users encrypted, as a list $V_{access} = \{enc(K_V, \mathsf{PubK}_{u'_1}), \ldots, enc(K_V, \mathsf{PubK}_{u'_m})\}$. Key K_V is stored encrypted and only a user u'_i who knows the private key $\mathsf{PrivK}_{u'_i}$ can read it. Note that K_V is not a key of any transaction; so knowing it still does not guarantee access to secret information.

The user u who created the view V is considered the *view owner* and any access to the information on the view is through this user.

Granting and revoking access. When a user u'_j wants to access transactions $t_{i_1}, \ldots, t_{i_k}$ of view V, it requests the keys of transactions $t_{i_1}, \ldots, t_{i_k}$ from u. These keys are provided encrypted as $enc(K_{i_1}, K_V), \ldots, enc(K_{i_m}, K_V)$. If the user u'_j has the key K_V, it can decrypt the sent keys and access the transactions. We assume here that K_V is a symmetric encryption key. Note that this does not reveal transaction keys that were not requested.

To grant additional users access to the view, u can share the key K_V with these users, encrypted using their public key. To revoke access from some users, u needs to replace the key K_V with a new key, say K'_V, and disseminate the new key K'_V to the users that still have permission to access V. This is done as described above, i.e., by encrypting the new key K'_V with the public keys of the authorized users and including that in the transaction of V_{access} stored on the blockchain. After the change, u should only use K'_V to encrypt requested transaction keys. When access is revoked, users may still have access to information they downloaded and stored locally, but they cannot access and download additional information.

4.3 Hash-based Irrevocable Access Permissions

The hash-based methods are very similar to the encryption-based methods, however, for a complete exposition, we present them. It is important to understand them when comparing our methods with private data collections of Hyperledger Fabric.

In hash-based views, only the hash of the the secret part of transactions is stored on the blockchain. For irrevocable access control (namely, HI), the secret part of transactions should be revealed only to authorized users, so it is stored in an encrypted view. The decryption key is provided only to users with access permission.

Storing transactions with a secret part. Initially, before adding a transaction t_{ij} to the blockchain, the user u who adds the transaction conceals the secret part of t_{ij} by selecting a random *salt* s, computing the hash of $h(t_{ij}[S] \parallel s)$, where $t_{ij}[S] \parallel s$ is the concatenation of the secret part of t_{ij} with the salt s, and storing this hashed value on the blockchain instead of storing $t_{ij}[S]$ itself. The salt prevents seeing that two secret values are the same in different

transactions, to prevent a Dictionary Attack [12]. Each transaction t_{ij} is stored as a 4-tuple $(tid_{ij}, t_{ij}[N], s_{ij}, h(t_{ij}[S] \parallel s_{ij}))$ of the identifier, non-secret part, salt and hash of salt and secret part.

Initially, only user u has the secret values of transactions. To grant access to the secret part of transactions, these values are disseminated to authorised users, via the access-control views.

Creating a view. Suppose that user u wants to grant access to transactions $t_1, \ldots, t_n$. Before the view is created, only the hash values $h(t_1[S] \parallel s_1), \ldots, h(t_n[S] \parallel s_n)$ of the secret parts of the transactions are stored on the blockchain. To create a view V and grant access to the secret parts of transactions $t_1, \ldots, t_n$, user u creates a new key K_V, and stores on the blockchain the encrypted content $enc((tid_1, t_1[S]), K_V), \ldots, enc((tid_n, t_n[S]), K_V)$.

Granting access to the view. The key K_V is distributed to all the users with access permissions to view V, encrypted using the public keys of these users. If user u' receives from u the key K_V, then u' can decrypt $t_1[S], \ldots, t_n[S]$ and verify that their hash values concatenated with the salt are equal to the hash values $h(t_1[S] \parallel s_1), \ldots, h(t_n[S] \parallel s_n)$ stored on the blockchain. Note that access to the data is irrevocable because if u' has the key K_V then that key reveals the secret part of each transaction in the view.

4.4 Hash-based Revocable Permissions

In hash-based view with revocable permissions (HR), the secret part of transactions is concealed—only the hash of the concatenation of the secret with salt is stored on the ledger. The view is similar to ER view, as in Section 4.2, with adaptation to hash-based storage.

Creating a view. To make the access permission revocable, secret information is not irreversibly disseminated to users, and secret information is not revealed by a transaction stored on the blockchain. Secret information is only provided upon request and only to authorized users, i.e., to users with an access permission.

Consider a revocable view V created by user u, for transactions $t_1, \ldots, t_n$. Let $u'_1, \ldots, u'_m$ be the users that should have access to the view V. User u must know the secret part of all the transactions in V to grant access to that information. But these secrets are only provided to the users $u'_1, \ldots, u'_m$ upon request, in a revocable way.

To support revocable access permissions, the view comprises the two parts V_{ids} and V_{access}. The list $V_{ids} = \{tid_1, \ldots, tid_n\}$ is the list of identifiers of the transactions $t_1, \ldots, t_n$ that V consists of. The list V_{access} is created with a key K_V that is selected by u and disseminated to the authorized users encrypted using their public keys, $V_{access} = \{enc(K_V, \mathsf{PubK}_{u'_1}), \ldots, enc(K_V, \mathsf{PubK}_{u'_m})\}$. Key K_V is stored encrypted and only a user u'_i who knows the private key $\mathsf{PrivK}_{u'_i}$ can get K_V. The user u who created V is considered the *view owner* and any access to secret information is via u.

Granting and revoking access. When a user u'_j wants to access transactions $t_{i_1}, \ldots, t_{i_k}$ of view V, it requests the secret values of these transactions from u. These values are provided encrypted as $enc(t_{i_1}[S], K_V), \ldots, enc(t_{i_k}[S], K_V)$. If the user u'_j has the key K_V, it can decrypt the provided information and see the secret part of the requested transactions. We assume here that K_V is a symmetric encryption key. Note that this does not reveal secret parts of transactions that were not included in the request. The user u'_j can verify that the decrypted values $t_{i_1}[S], \ldots, t_{i_k}[S]$ are indeed the secret values stored on the blockchain by computing $h(t_{i_j}[S] \parallel s_{i_j})$ and verifying that the computed value is equal to the value stored on the blockchain for transaction t_{i_j}.

To grant access to the view, u can securely share the key K_V with the new users who should gain access to V, where K_V is encrypted using their public key. To revoke access from users, u needs to replace the key K_V with a new key, say K'_V, and disseminate the new key K'_V to the users that still have a permission to access V. The new key K'_V is encrypted using the public keys of the authorized users, and the encrypted keys are included in V_{access}. After the change, u should only use K'_V to encrypt secret values it serves.

4.5 Encryption versus Hashing

In the encryption-based methods, all the information is stored on the blockchain, where the secret data is encrypted, while in the hash-based methods, only the hash of the secret is stored on the blockchain. An advantage of encryption is that users maintain only transaction keys, while the actual data is stored on the blockchain. When storing just the hash values on the blockchain, the secret information must be stored by the users, and the hash values on the blockchain are only used for verification. A disadvantage of encryption is the cost of creating a new key per each transaction.

The storage space of the hash values is fixed while for the encrypted data it depends on the attribute size. Hence, if the average secret size per transaction is larger than the hash size, the encryption-based methods will use more space than the hash-based methods, and vice versa. For example, hash is preferred when records contain large objects like images or videos. Encryption is suitable for storing small records like personal details of people.

Both revocable and irrevocable views can be implemented using encryption or hashing. The decision whether to use a revocable or an irrevocable view depends on the application. These two types of views may serve different purposes. For example, health records are typically stored in a revocable way; so that access could be revoked from healthcare workers who are no longer active, e.g., when they retire. Access to legal information, like deeds, patents, licences, and warranty should typically be irrevocable, to make the information available in the future to all the involved parties.

4.6 Role-Based Access Control

In role-based access control (RBAC), roles are assigned to users and access permissions are given to roles [23, 48]. A user u can access an entity e only if u has a role with access permissions to e. Administrative roles to allow access to non-business-related information are examples of roles in the refurbished devices application.

Let U be a set of users, R be a set of roles and D be a dataset. Let $\mathcal{V} \subseteq 2^D$ be the set of views over D—each view is a subset of items of D. Assignment of roles to users is a subset $A_r \subseteq U \times R$. Access permission is a set of pairs $A_p \subseteq R \times \mathcal{V}$. A pair $(u, r) \in A_r$ means that user u has role r, e.g., user u is a nurse with access to medical records. The assignment of roles is many-to-many. An access permission $(r, V) \in A_p$ means that users with role r are authorized to access view V. The access-permission relation is many-to-many. For a role assignment A_r and access permissions A_p, the set $D_u = \{V \mid \exists r.(u, r) \in A_r \wedge (r, V) \in A_p\}$ consists of the

views user u is allowed to access. In a blockchain setting, access permissions relate to views that are sets of transactions.

RBAC can be implemented for all four types of access-control views presented. To support this, first, the association A_p of roles to views is stored on the blockchain as a transparent list. This list can be stored as a transaction or as a state of a smart contract. Second, the association A_r between roles and public keys of users is stored on the blockchain in the same way. Any user can apply join to A_p and A_r, to find all the public keys of users with access permissions, for any given view. These public keys are used for granting access to the views. Let $\mathcal{K}_{A_r \bowtie A_p}(V)$ denote the set of all public keys of users with access to V according to A_r and A_p.

In irrevocable views, EI and HI, access to a view V is granted by providing the key K_V that is used for decryption of the information (keys in EI and values in HI). In the irrevocable version of RBAC, the key K_V is distributed to all the users with access permissions by encrypting it with their keys and disseminating the encrypted information. Formally, this set is $\{enc(K_v, K_i) | K_i \in \mathcal{K}_{A_r \bowtie A_p}(V)\}$. The users with access permissions have the corresponding private keys and they can decrypt K_V and access the view. Other users are not privy to the key K_V and cannot read the content of the encrypted view V. The implementation of the key distribution for the revocable view methods ER and HR is similar.

When a role grants access to many views, it is inefficient to distribute each key K_V independently. Thus, a pair PrivK_r and PubK_r of public and private keys is created for each role r, as if the role is a user. This can be done by any user. The private key PrivK_r is securely shared with all the users with role r, e.g., by encrypting PrivK_r with the public keys of these users and sending the encrypted PrivK_r to them. Then, in all the four methods, the access to the view is granted to the role whose public key is PubK_r in the same way that it is granted to a user, as described in Section 4. The methods are indifferent to whether the public key belongs to a single user or to a group of users defined by a role. When the set of users changes for role r, a new key is created and disseminated.

4.7 Soundness and Completeness

We now analyze the proposed approach. A malicious user who generates or updates a view can (1) add to a view a transaction that should not be included in the view, (2) add to the view a transaction that should be included in the view but in a corrupted or tampered way, or (3) not add to the view a transaction that should be included in the view. We explain now how verifiable soundness and completeness (defined in Section 3) are maintained.

Case 1: Suppose that a transaction t has a non-secret part $t[N]$ that does not satisfy the view definition. Including t in view V will violate the soundness of the view. However, because $t[N]$ is the non-secret part of the transaction, any user with access to the view can detect such transactions and verify that there are no such transactions in the view. Hence, verifiable soundness is maintained.

Case 2: If a corrupted transaction is added to the view, this could violate the soundness of the view. A case where the non-secret part of a transaction is corrupted can be detected by any user with access to the view, by comparing the non-secret part of the transaction to the non-secret part of the transaction stored on the ledger. If the secret information is corrupted, the hash of the transaction would not match the one on the ledger, in the hash-based methods. The keys would not match, in the case of corrupted keys in encryption-based methods. Hence, verifiable soundness is maintained.

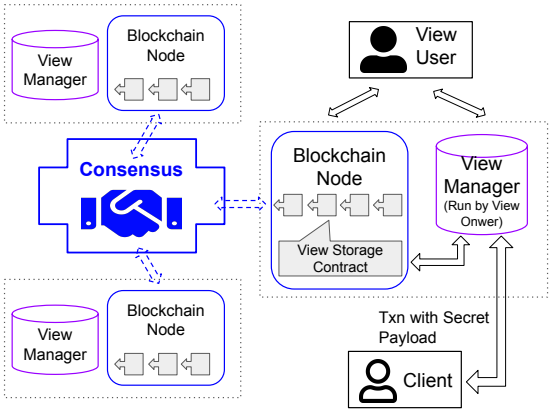

Figure 2: System architecture. Each blockchain node is associated with a view manager, under the same administrative domain of the view owner (dashed box). Transactions with secret data are added to the blockchain and accessed through a view manager. The view contract provides the integrity guarantee. The number of blockchain nodes may vary.

Case 3: Not adding to a view all the transactions it should include will violate completeness. There are two ways to test completeness of the view at given time T. First, by iterating over all the transactions in the ledger, up to time T, and verifying that all those that satisfy the view definition are included in the view. Second, by comparing the view to a complete list of the view transactions maintained by a smart contract. Such a list is continuously updated for efficient verification. We elaborate on that in the next section. By that, verifiable completeness at T is maintained.

Proposition 4.1. *Given a view V and a user u with access permission to V and to the ledger, then for any time T, u can test and verify the soundness and completeness of V at T.*

5 IMPLEMENTATION

In this section we describe the system and its architecture. The code is available on GitHub [1].

5.1 Hyperledger Fabric

We implemented the system using Hyperledger Fabric 2.2—a permissioned blockchain system [7]. Fabric is customized for enterprise-grade applications, and has a couple of unique features. The smart contracts were implemented using a special life cycle, known as *chaincode*. In the life cycle of a transaction, the first step is an *endorsement* phase, where clients collect invocation effects of chaincodes from a subset of peers (namely *endorsers*). These invocations are cryptographically signed by the endorsers. The signed endorsements are included in transactions and distributed to *orderers*. The orderers reach a consensus on the order of transactions, batch them into blocks and link the blocks to form the chain. Each peer independently pulls blocks from orderers and applies the effect of

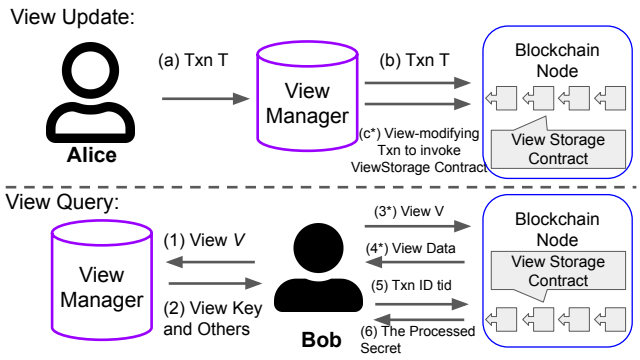

Figure 3: View-management workflow, for both encryption-based and hash-based view managers. Alice is a blockchain user who invokes a transaction T, which is included in view V. Bob is a view user accessing view V. Steps marked with an asterisk are only executed for irrevocable views, where view data are uploaded to a view contract on the chain.

transactions and chaincodes to the local database, after validation by verification of the signed endorsements.

Although we implemented access-control view on Fabric, our implementation of views is generic, because we do not rely on any feature that is unique to Fabric and does not exist in other blockchain systems. Our system uses smart contracts in a general way, without relying on the chaincode lifecycle. We only rely on tamper-evidence guarantees of smart contracts for the integrity of view data, but these are provided by any blockchain that supports smart contracts. Any guarantee of state integrity, e.g., by a Merkle tree, can be used for securing the integrity of blockchain views.

5.2 View as a Smart Contract State

The size of a view can be arbitrarily large. In some cases, access control views can be as large as the database itself. In such cases, it is impractical to store the entire view in a single blockchain transaction. The same may also apply to storing the list of users who have access to a particular view. To solve that, views and user lists are implemented as smart-contract states. The entire state is stored in the leaves of a Merkle tree, for each one of the local databases of the peers, and the hash at the root is stored on the ledger, as explained in Section 3. The consensus among the peers on the view state makes it possible to verify the integrity of the views while only storing a digest on the ledger.

By representing the view as a state of the smart contract, every change in the view, like the addition of transactions to the view or the removal of transactions from the view, is represented by a new state that is propagated to all the peers. The new state is accepted as part of the consensus when the new root of the Merkle tree that reflects the change is included in the header of a block in the chain.

5.3 System Architecture

To implement decentralized view management, a view manager component is associated with each blockchain node. Users access views via the view manager for (1) reading from the view, and (2) adding transactions to the blockchain. The view manager is implemented using smart contracts. An overview of the architecture and the interactions between components are depicted in Fig. 2 and Fig. 3. Next, we elaborate on these components and interactions.

User Applications. There are three types of user applications.
(1) *Clients*, like Alice in Fig. 3, are users who invoke transactions with secret data. Client transactions have a hidden secret part. The access-control views are defined over client transactions.
(2) *View Owner.* View owners run a process that implements a `ViewManager` interface. The view manager intercepts and handles client requests. It translates requests into transactions, determines inclusion of transactions in views and regulates access to transactions using the methods presented in Section 4. A view owner can be any user with access to all the information of the view. Hence, a view can have many view owners.
(3) *View Reader.* View readers, like Bob in Fig. 3, issue access requests. The requests are handed over to view owners, to be handled based on the view name and the public key of the requesting user. View readers do not always trust view owners; so provided data can be validated against the blockchain—by checking that returned secrets match the hash values on the blockchain, or that provided keys can be used for decryption of the secret data on the blockchain.

View Manager. The different users interact with the `ViewManager`. Transactions are instantiated by contract invocations, as common in contract-based blockchains. To handle transactions with a secret part, the smart contracts of the view manager implement a common interface `WithSecret`. The `WithSecret` interface partitions transactions into a non-secret part and a secret part, and it contains methods for processing and accessing the secret part of transactions. An `Invoke` smart contract implements the invocation logic. For a given transaction, the invocation associates the concealed secret data in the transaction with the transaction identifier in the local database of the node. Then, the contract may execute additional business logic, such as the supply-chain management procedures.

There are two main components in `ViewManager`: `CryptoHelper` which is a library of basic cryptographic primitives, and `ViewBuffer` for managing the information on each view. `ViewManager` and the interactions between the components are illustrated in Fig. 2.

The `ViewBuffer` component of `ViewManager` stores the view information. This includes (1) `ViewKeys`, which is a map structure that associates the view name to the view key, (2) the key K_v that encrypts the list of keys in the encryption-based methods of Section 4.1, (3) `ViewData`, which is a map structure that associates the view name to the relevant data. The implementation of `ViewData` changes based on the view method—encryption or hashed-based.

View Management. View management has the following steps.
(1) View owners create views by calling the method `CreateView` of the view manager. Each created view has a unique name and a view definition. The view definition is an implementation of the predicate that defines the view. It is a function that determines for each transaction whether it satisfies the view predicate. Transactions that satisfy the view predicate are included in the view. If the view owner opts for an irrevocable view, the view manager calls `Init` for instantiating the view when executing the `ViewStorage` contract (to be explained later), which manages the view storage.

(2) Transactions with a secret part are inserted into the blockchain through `InvokeWithSecret`. The view manager processes the secret data provided by the client, based on its subclass implementation of `ProcessSecret`, to cope with different view types. The processed secret and the other arguments adhere to the `WithSecret` interface. The view owner retrieves the transaction identifier based on the formatted parameters, and determines all the view-definition predicates that are satisfied by the parameters. For each view whose predicate is satisfied, the view manager calls `InsertIntoView`, to include the transaction identifier in the view. For example, clients, like Alice in Fig. 3, contact the view owner and invoke a transaction, e.g., for insertion into the blockchain or for adding to a view. The request is processed by the view owner who invokes `InvokeWithSecret` in its `ViewManager` process. The view manager of a view owner processes the request and securely delivers the data to a `ViewStorage` contract that handles the data storage.

(3) A view user, like Bob in Fig. 3, may pose a query to a view by invoking `QueryView` via the view manager. Despite implementation differences, the public key of the user is provided with the query for authentication. Query results are encrypted with that key.

View Storage Contract. A view-storage contract is called for storing data of irrevocable views. The type of view (revocable versus irrevocable) is set when `CreateView` is first called. The `ViewStorage` contract maintains a map structure `ViewInfo` that associates the view name with the view data. The view data is also a map structure whose internal structure for the key-value pairs depends on the implementation of `ViewManager`, to support the different view types. `ViewStorage` supports two methods. An `Init` method initializes for a given view name an empty view data, i.e., an empty map. A `Merge` method adds data to the view by incorporating in `ViewInfo` missing key-value pairs from `ViewData`. By maintaining the irrevocable view data in a contract, we use security features of the blockchain to protect the view integrity. Thus, query results that are returned to view users are protected from tampering.

5.3.1 Encryption-based View Manager. The implementation of the view manger depends on the view type. We elaborate now on encryption-based view manager.

Overview. `EncryptionBasedManager` is a class that implements the `ViewManager` interface and the methods of sections 4.1 and 4.2. The format of the view and the `ViewBuffer` are designed to support the encryption-based methods. The view data is a mapping from the transaction identifier tid_i to the transaction key K_i, in the notation of Section 4.1. The procedures `InsertIntoView`, `ProcessSecret`, and `QueryView` are adapted to this format.

Secret Processing. In `ProcessSecret`, a new symmetric key is created for each new transaction. The procedure encrypts the secret part of the transaction and returns the encrypted text and key.

View Update. When `InsertIntoView` is executed, the transaction identifier and the generated transaction key are stored in the `ViewBuffer`. In the irrevocable case, the view key (which is generated if needed) is used for encrypting the identifier and the key of the transaction. The encrypted result is provided to the `Merge` procedure of the `ViewStorage` contract. For views with irrevocable access permissions, the view data are automatically updated on the chain to obtain the blockchain-provided immutability.

Querying a View. There are differences between revocable and irrevocable view management, when answering user queries. For revocable access permissions, a symmetric key is generated per each query. This key is used for encrypting the view data, which is a mapping from transaction identifiers to transaction keys. For irrevocable access permissions, the view key has already been created in `InsertIntoView`; so it is directly retrieved from `ViewBuffer`. In both cases, the user-provided public key is applied, in order to encrypt the returned data and protect the view keys.

Validation. Users validate query results against the content of the blockchain, to verify that the results were not tampered with. After receiving a reply from a view owner, when calling `QueryView`, the user validates the secret data, for each transaction in the view. First, view keys are revealed by decryption, using the private key of the user. The query answer contains the encrypted view data, hence, the user can decrypt the view data with the provided view keys. For views with irrevocable permissions, users retrieve the encrypted view data from the `ViewStorage` contract, and then execute decryption. With the decrypted identifiers and keys, of the requested transactions, the user can pull from any blockchain node the relevant transactions and decrypt their secret part.

5.3.2 Hash-based View Manager. The hash-based view manager `HashBasedManager` has the same architecture as the encryption-based manager. The storage, the buffer and methods like view update, querying and validation are modified accordingly.

5.4 Transaction List per View

To test completeness for views, it is efficient to maintain for each view its list of transactions. But transactions cannot be added to views through smart contracts because that would require revealing the view key to the blockchain peers, which would expose information to unauthorized users. Thus, insertion of transactions to the main database is through a smart contract that updates lists of transaction identifiers per view, but does not update the views themselves, e.g., if transaction t_i is added to the blockchain and should be included in views V_1 and V_2, then tid_i will be added to the transaction lists of V_1 and V_2, based on the non-secret part $t_i[N]$. We refer to this smart contract as `TxListContract`.

The smart contract maintains a list of views, their predicates and their list of transaction identifiers. For each added transaction, its identifier is added to all the lists for which the view predicate is satisfied. To cope with the low update rate of blockchains, the list updates are conducted in batches. Initially, updates are collected and transaction identifiers are associated with the insertion time stamp. Every time interval, say 30 seconds, all the accumulated updates are written to the ledger. Completeness can be tested for the time of the latest update.

6 EXPERIMENTAL EVALUATION

Goals. The goals of the experiments are as follows. (1) Examine the effect of our methods on the performance, in terms of transaction rate, latency and storage space. (2) Compare our access-control views with a baseline in which different views are stored on independent ledgers and cross-blockchain transactions keep the consistency of the views and the main ledger. (3) Explore the overhead

of managing a list of transaction identifers per view—the contract `TxListContract` (`TLC`, for short) described in Section 5.4.

Experimental setup. To test the methods, we deployed an instance of a Fabric 2.2 network in the Google Cloud Platform, GCP. The network consists of 2 peer processes and 3 orderer processes. Each process runs on distinct computing nodes of `e2-standard-4` machine types. To test a decentralized setting, we deployed nodes in different geographical regions — two peers at the `europe-north1-a` and `northamerica-northeast1-a` regions, and three orderers at the `asia-southeast1-a` region. We opt to use Raft as the consensus protocol of orderers. LevelDB was used for maintaining the local storage in each node. Every experiment was repeated three times and we report the average result of the different executions.

6.1 Baseline

As a baseline, we consider the case where each view is stored on a separate blockchain, called *view blockchain*, and each blockchain is accessible only to users with access permissions for the corresponding view. To keep the views consistent with the main blockchain, which stores all the transactions, we use cross-chain transactions. If a transaction t is included in n different views, a cross-blockchain insertion transaction will guarantee consistency among all the view blockchains—t should either be included in all the n corresponding blockchains or in none of them. Hence, the operation is atomic.

We implemented the cross-blockchain transactions protocol of AHL [17] where each cross-blockchain transaction follows the two-phase commit (2PC) protocol. The main blockchain operates like a database transaction coordinator of 2PC. For each database update, it employs a smart contract to determine the updated views and the view blockchains that store them. It then issues to all of them a `Prepare` request. Each view blockchain operates as a 2PC shard where the 2PC protocol logic is implemented as a smart contract. After gathering sufficient positive responses from the view blockchains, the main blockchain issues a `Commit` request to all the view blockchains, and the update is included in their ledger.

If a cross-blockchain transaction involves n blockchains, it is translated into $2n$ blockchain transactions, following the 2PC protocol. The first n transactions simulate the `Prepare` requests on each one of n blockchains. The following n transactions simulate the corresponding `Commit` requests of the 2PC protocol. Note that n `Prepare` requests or n `Commit` requests can occur concurrently.

If the majority of the peers in all the blockchains are following the protocol, the 2PC procedure provides verifiable soundness and completeness. Each view blockchain will include only transactions of the view and will include all the transactions (otherwise, the commit phase of 2PC will not be carried out). Without 2PC, where view blockchains are independent from the main chain, neither soundness nor completeness will be guaranteed.

6.2 Supply Chain Workload

We benchmarked the access-control views on a variety of supply chain workloads similar to the supply chain illustrated in Fig. 1. To that end, we implemented a workload generator that produces workloads of supply chain transactions according to given parameters. When a workload is created, first the topology of the supply-chain graph is defined (e.g., see the graph in Fig. 1). The user specifies the nodes and the edges of the graph of the supply chain. Each node represents a real-world entity that is part of the supply chain and has a need to access information. Each edge from node n_i to node n_j represents a delivery link from node n_i to node n_j, for delivering received items from n_i to n_j. An item cannot be forwarded by node n_i to more than one following node.

Some nodes are *dispatching nodes*, e.g., manufacturers. These nodes can create items and send them on to following nodes in the chain. All the other nodes can only forward items that they received. Some nodes are *terminal nodes*. They receive items and do not transfer them on to a following node, e.g., shops in Fig. 1. The user can select how many items will be dispatched and the behavior of intermediate nodes. Each transfer is recorded on the blockchain. In the creation of the workload, when an item is forwarded, all the nodes that handled it can see the transfer transaction. That is, nodes can continue tracking an item they delivered. Nodes can also see all the historical transfers of the items they received. However, a node should not have access to any other delivery transactions.

For the generated records of the supply-chain, transactions are stored on a blockchain and access restrictions are implemented as access-control views. For each node n, a view V_n contains all the transactions that n has access to. When an item is transferred, the transaction is added to the main ledger and all the relevant views are updated. For example, suppose that item i was created in node n_0, then it was transferred from n_0 to node n_1, then from n_1 to n_2, and the most recent transaction is a transfer of i from node n_2 to node n_3. Then, the transaction (i, n_2, n_3) is added to the blockchain. The access to it is granted to nodes n_0, n_1, n_2 and n_3. So, the access-control views of all these nodes, i.e., V_{n_0}, V_{n_1}, V_{n_2} and V_{n_3}, are updated by adding this transaction to them. In addition, the view of node n_3 is updated by adding the historical transfers of item i to it, that is, granting access to n_3 for these transactions. This means that the transactions (i, n_0, n_1) and (i, n_1, n_2) are added to view V_{n_3}, to reflect granting access permissions to n_3 for item i.

The workload generator was used for benchmarking the access control views on supply chains with varying topology, size and volume of transactions. Note that the size of the supply chain, i.e., the number of nodes in it, determines the number of views. We experimented with two workloads. Workload `WL1` with 7 nodes—one dispatching, 3 intermediate and 3 terminal nodes; and workload `WL2` with 14 nodes—2 dispatching, 5 intermediate and 7 terminal nodes. Accordingly, there are 7 views in `WL1` and 14 views in `WL2`.

6.3 Results

For different workloads, we measured throughput, latency, storage overhead and the number of on-chain transaction invocations. For the baseline we also measured cross-chain invocations.

Transaction Rate. We measured the transaction rate, i.e., the number of committed application requests per second (TPS), as a function of the number of clients, over workload `WL1`. Each client groups 25 requests into a batch and transaction batches are invoked in a sequential manner. Fig. 4 presents the result throughput as a function of the number of client processes. Our view methods achieve higher scalability than the baseline. Revocable views and irrevocable with `TxListContract` have the highest throughput. When the number of clients exceeds 48, the throughput becomes stable, around 800 requests per second. Similar throughput has been measured for Fabric in other papers [46, 52, 57].

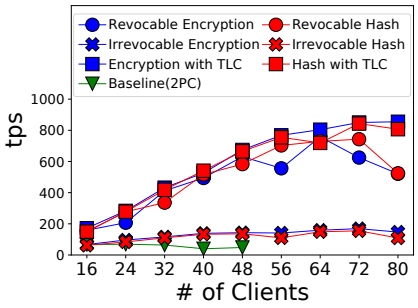

Figure 4: Throughput

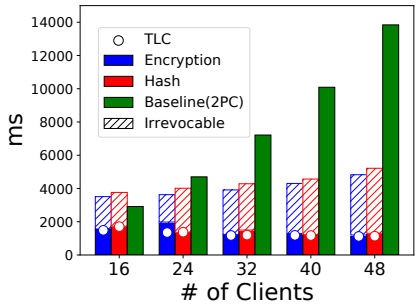

Figure 5: Latency

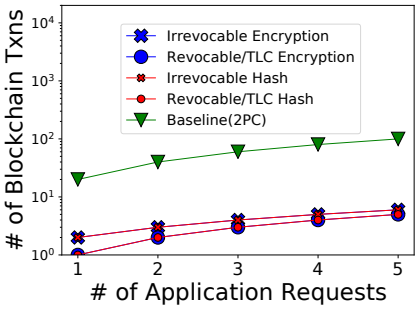

Figure 6: Transactions per request

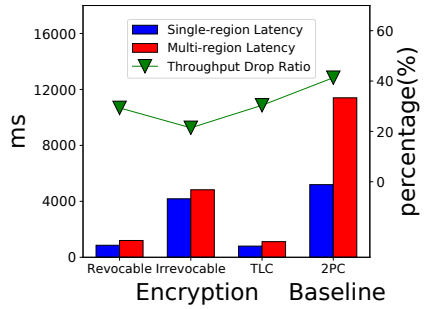

Figure 7: Single vs. multiple regions

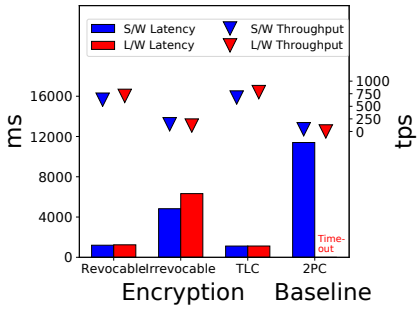

Figure 8: Different workloads

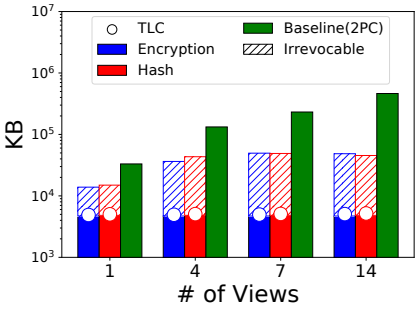

Figure 9: Storage overhead

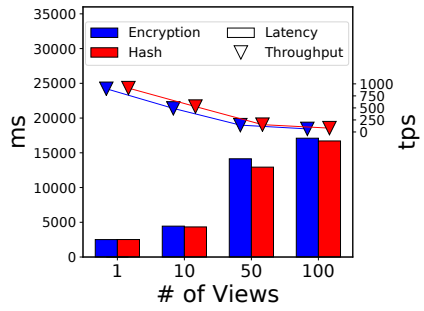

Figure 10: Each tx is in all the views

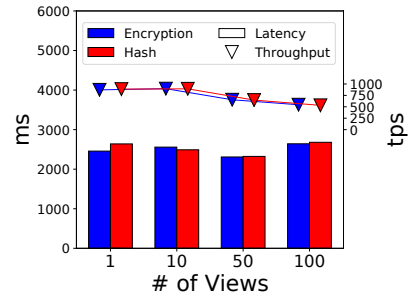

Figure 11: Each tx is in a single view

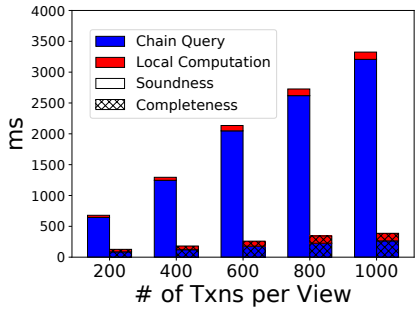

Figure 12: Verification performance

For views with irrevocable access, the system handled around 150 requests per second. The lower throughput of irrevocable views is because in addition to the procedures performed by the revocable methods, they also call view-modifying transactions, which require extra computations. (For comparison, in the AT&T app for tracking refurbished devices the requirement was to support 100 transactions per second.) The baseline requires many more transactions per request, i.e., a request is translated into $2n$ transactions, for updating n views. Hence, it does not scale well, and it has a much lower throughput of less than 70 requests per second, with a peak at 24 clients. Beyond 48 clients, the baseline becomes unresponsive—the system cannot handle the large number of cross-chain transactions.

Latency. Fig. 5 depicts the per-request latency for our view methods and the baseline, over workload `WL1`. The irrevocable views have a higher latency than the revocable views due to the additional view-modifying on-chain transactions. Note that the time needed for off-chain computations, like encryption and hash computations, is negligible. Employing `TxListContract` (TLC) reduces the number of on-chain transactions and decreases latency, making the latency of irrevocable views close to that of revocable views. The baseline has high latency, and as the number of clients increases many more cross-chain transactions are needed and the latency soars.

Processing Cost. The performance is affected by the total number of on-chain transactions, hence, we measured the number of on-chain transactions as a function of the number of Application Requests (see Fig. 6). Without the extra view-modifying contract, in revocable views and when using `TxListContract`, the number

of application requests is equal to the number of on-chain transactions. In irrevocable views, we need an extra on-chain transaction to invoke the view storage contract; so the number of on-chain transactions is doubled—r requests will result in $2r$ on-chain transactions. The cross-chain baseline, for comparison, requires $2 \cdot |V| \cdot n$, where $|V|$ is the average number of views per transaction, and n is the number of requests. In Fig. 6, $|V| = 10$.

Spatial Distribution. We investigated the effect of spatial distribution on the performance by comparing deployment on a single GCP region to deployment on distant GCP regions. See results in Fig. 7. The effect on latency is small for our methods but it is significant for the baseline. However, the throughput of our methods dropped by 20-30% and dropped by more than 40% for the baseline.

Different Workloads. We compared the performance of the methods over workloads `WL1` and `WL2`. The results are presented in Fig. 8, where `S/W` refers to the smaller workload (`WL1`) and `L/W` to the larger one (`WL2`). Since the views are maintained as a smart-contract instance and most of the operations are off-chain, the increase in the size of the workload has a very small effect—the effect is negligible in comparison to on-chain computations. For the baseline, however, the increase in workload leads to too many cross-chain transactions, and it reached a timeout without delivering results.

Storage Overhead. We examined the storage overhead of the blockchain per the number of views, after committing 40 supply-chain requests. Fig. 9 presents the results. Without the view-storage contract, the revocable methods need the least space and are not affected by the number of views. When using `TxListContract`, the list of transaction identifiers and view predicates are stored in the smart contract. This reduces storage in the contract state, and the overall storage. For irrevocable views without `TxListContract`, increasing the number of views requires storing more contract states, which leads to more storage overhead. Yet, the baseline approach is the most wasteful in storage space because it needs to duplicate transactions—a transaction in n views is duplicated n times. Therefore, the total space needed for the baseline approach is tenfold greater than the storage space used by our view methods.

Scalability. We examined the effect on latency and throughput of an increase in the number of views and the number of transactions per view. When each transaction is in all the views (Fig. 10), increasing the number of views from 1 to 100 makes the latency rise from around 2500 ms to about 17000 ms, and it drops the throughput from around 800 TPS to 80 TPS. When each transaction is in a single view (Fig. 11), increasing the number of views has only a small affect on the performance—the latency is about 2500 ms in all scenarios and the throughput values are between 600 and 900 TPS. The results are similar for both the hash-based and encryption-based methods.

The difference between the cases in Fig. 10 and Fig. 11 is because when updating many views per transaction, the transaction needs to include more information in its payload. This increases the size of transactions and reduces the number of transactions per block, affecting the throughput and the latency.

Verification Delay. Fig. 12 presents the cost of verifying soundness and completeness of a view. There is a linear increase in verification time per the number of transactions, for both soundness and completeness. Most of the delay is due to access to the ledger, while local computations only slightly increase the delay. Soundness verification is much more costly than verifying completeness, because the soundness test requires access to the ledger per each transaction, to validate it, while completeness uses the list presented in Section 5.4. Note that the cost of soundness verification can be reduced when assuming that users with access to the data can be trusted; however, in our system we do not make such an assumption.

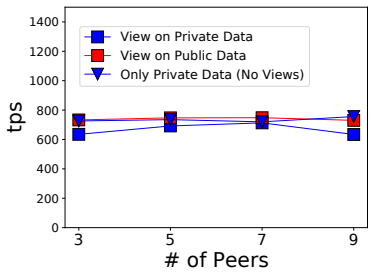

Figure 13: Comparison with Private Data Collections

Comparison with Private Data. We compared the efficiency of our views with the private data collections of Fabric (described in Section 2). The results, presented in Fig. 13, compare (1) a private data collection, (2) a revocable view on top of private data collection, by including our soundness and completeness tests, and (3) our revocable hash-based view. There is a slight performance decrease when comparing views to private data collections, and utilizing built-in private data collections in our methods does not improve performance. Recall that our views provide additional capabilities that private data collections do not provide, e.g., irrevocable views, effective way to grant and revoke access to revocable views, etc.

7 CONCLUSIONS

In this paper, we presented the LedgerView system that adds access-control views to Hyperledger Fabric. We demonstrated the use of access-control views for management of supply chains and in an AT&T application of tracking parts of refurbished cellular devices. We introduced four methods of views on blockchain. Two methods are based on data encryption (comparable with channels) and two methods protect sensitive data by storing on the blockchain only the hash values of sensitive data (comparable with private data collections). We showed how to implement views with revocable and irrevocable access permissions. Revocable access permissions allow revoking access to data from users, e.g., from retired employees. Irrevocable access should be used for irrevocable information, e.g., warranty, contracts, ownership documents, deeds, etc.

To validate the effectiveness of LedgerView, we compared it to a baseline of storing different views on independent blockchains and using cross-chain transactions for guaranteeing consistency. Experiments with supply-chain workloads show that LedgerView has much higher throughput than the baseline and the cost of the baseline is much higher than the cost of LedgerView, in terms of latency, storage overhead, and communication between nodes.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation, Singapore under its Emerging Areas Research Projects (EARP) Funding Initiative. We would like to thank Thomas Jenkins from AT&T for his insights on tracking refurbished devices.

REFERENCES

[1] Ledgerview. https://github.com/sbip-sg/BlockchainView.

[2] Serge Abiteboul. On views and XML. *ACM SIGMOD Record*, 28(4):30–38, 1999.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[4] Abdulrahman Almutairi, Muhammad Sarfraz, Saleh Basalamah, Walid Aref, and Arif Ghafoor. A distributed access control architecture for cloud computing. *IEEE software*, 29(2):36–44, 2011.

[5] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Caper: a cross-application permissioned blockchain. *PVLDB*, 12(11):1385–1398, 2019.

[6] Andy Amoordon and Henrique Rocha. Presenting Tendermint: Idiosyncrasies, weaknesses, and good practices. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 44–49. IEEE, 2019.

[7] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[8] Ahmed Banafa. IoT and blockchain convergence: benefits and challenges. *IEEE Internet of Things*, 2017.

[9] Elaine Barker and Nicky Mouha. Recommendation for the triple data encryption algorithm (tdea) block cipher. Technical report, National Institute of Standards and Technology, 2017.

[10] Fabrice Benhamouda, Shai Halevi, and Tzipora Halevi. Supporting private data on hyperledger fabric with secure multiparty computation. *IBM Journal of Research and Development*, 63(2/3):3–1, 2019.

[11] Elisa Bertino, Gabriel Ghinita, and Ashish Kamra. *Access control for databases: Concepts and systems*. Now Publishers Inc, 2011.

[12] L Bošnjak, J Sreš, and Bosnjak Brumen. Brute-force and dictionary attack on hashed real-world passwords. In *2018 41st international convention on information and communication technology, electronics and microelectronics (mipro)*, pages 1161–1166. IEEE, 2018.

[13] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[14] Donald D Chamberlin, Jim N Gray, and Irving L Traiger. Views, authorization, and locking in a relational data base system. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 425–430, 1975.

[15] Joan Daemen and Vincent Rijmen. Reijndael: The advanced encryption standard. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 26(3):137–139, 2001.

[16] Gaby G Dagher, Jordan Mohler, Matea Milojkovic, and Praneeth Babu Marella. Ancile: Privacy-preserving framework for access control and interoperability of electronic health records using blockchain technology. *Sustainable cities and society*, 39:283–297, 2018.

[17] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[18] Tamraparni Dasu, Yaron Kanza, and Divesh Srivastava. Geofences in the sky: herding drones with blockchains and 5G. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 73–76, 2018.

[19] Tamraparni Dasu, Yaron Kanza, and Divesh Srivastava. Unchain your blockchain. In *Proc. Symposium on Foundations and Applications of Blockchain*, volume 1, pages 16–23, 2018.

[20] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1366–1385, 2018.

[21] Davor Dujak and Domagoj Sajter. Blockchain applications in supply chain. In *SMART supply network*, pages 21–46. Springer, 2019.

[22] Paul Dunphy and Fabien AP Petitcolas. A first look at identity management schemes on the blockchain. *IEEE Security & Privacy*, 16(4):20–29, 2018.

[23] David Ferraiolo, D Richard Kuhn, and Ramaswamy Chandramouli. *Role-based access control*. Artech House, 2003.

[24] Michael Gertz and Sushil Jajodia. *Handbook of database security: applications and trends*. Springer Science & Business Media, 2007.

[25] Shay Gueron, Simon Johnson, and Jesse Walker. Sha-512/256. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 354–358. IEEE, 2011.

[26] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. Fault-tolerant distributed transactions on blockchain. *Synthesis Lectures on Data Management*, 16(1), 2021.

[27] John D Halamka, Andrew Lippman, and Ariel Ekblaw. The potential for blockchain to transform electronic health records. *Harvard Business Review*, 3(3):2–5, 2017.

[28] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254, 2018.

[29] Uzair Javaid, Muhammad Naveed Aman, and Biplab Sikdar. Blockpro: Blockchain based data provenance and integrity for secure iot environments. In *Proc. of the 1st Workshop on Blockchain-enabled Networked Sensor Systems*, pages 13–18, 2018.

[30] Yaron Kanza, Alberto O Mendelzon, Renée J Miller, and Zheng Zhang. Authorization-transparent access control for XML under the non-truman model. In *International Conference on Extending Database Technology*, pages 222–239. Springer, 2006.

[31] Yaron Kanza and Eliyahu Safra. Cryptotransport: blockchain-powered ride hailing while preserving privacy, pseudonymity and trust. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 540–543, 2018.

[32] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[33] Henry M Kim and Marek Laskowski. Toward an ontology-driven blockchain design for supply-chain provenance. *Intelligent Systems in Accounting, Finance and Management*, 25(1):18–27, 2018.

[34] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Private data management for decentralized ledgers. *PVLDB*, 14(4):586–599, 2020.

[35] Shuaicheng Ma, Tamraparni Dasu, Yaron Kanza, Divesh Srivastava, and Li Xiong. Fraud buster: Tracking irsf using blockchain while protecting business confidentiality. In *CIDR*, 2021.

[36] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. Blockchain based access control. In *IFIP international conference on distributed applications and interoperable systems*, pages 206–220. Springer, 2017.

[37] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. A blockchain based approach for the definition of auditable access control systems. *Computers & Security*, 84:93–119, 2019.

[38] Sidra Malik, Salil S Kanhere, and Raja Jurdak. Productchain: Scalable blockchain framework to support provenance in supply chains. In *17th International Symposium on Network Computing and Applications (NCA)*, pages 1–10. IEEE, 2018.

[39] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[40] Tier Nolan. Alt chains and atomic transfers. In *Bitcoin Forum*, 2013.

[41] Sunny Pahlajani, Avinash Kshirsagar, and Vinod Pachghare. Survey on private blockchain consensus algorithms. In *1st International Conference on Innovations in Information and Communication Technology (ICIICT)*, pages 1–6. IEEE, 2019.

[42] Zhe Peng, Cheng Xu, Haixin Wang, Jinbin Huang, Jianliang Xu, and Xiaowen Chu. P2b-trace: Privacy-preserving blockchain-based contact tracing to combat pandemics. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2389–2393, 2021.

[43] Ling Ren and Srinivas Devadas. Proof of space from stacked expanders. In *Theory of Cryptography Conference*, pages 262–285. Springer, 2016.

[44] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[45] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *PVLDB*, 12(9):975–988, 2019.

[46] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 543–557, 2020.

[47] Fahad Saleh. Blockchain without waste: Proof-of-stake. *The Review of financial studies*, 34(3):1156–1190, 2021.

[48] Ravi S Sandhu. Role-based access control. In *Advances in computers*, volume 46, pages 237–286. Elsevier, 1998.

[49] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[50] Lakshmi Siva Sankar, M Sindhu, and M Sethumadhavan. Survey of consensus protocols on blockchain applications. In *4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 1–5. IEEE, 2017.

[51] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich, and Divya Agrawal. chainifydb: How to get rid of your blockchain and use your dbms instead. In *CIDR*, 2021.

[52] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, pages 105–122, 2019.

[53] Sachin Shetty, Val Red, Charles Kamhoua, Kevin Kwiat, and Laurent Njilla. Data provenance assurance in the cloud using blockchain. In *Disruptive Technologies in Sensors and Sensor Systems*, volume 10206, page 102060I. International Society for Optics and Photonics, 2017.

[54] Marten Sigwart, Michael Borkowski, Marco Peise, Stefan Schulte, and Stefan Tai. Blockchain-based data provenance for the internet of things. In *Proceedings of the 9th International Conference on the Internet of Things*, pages 1–8, 2019.

[55] Joao Sousa, Alysson Bessani, and Marko Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 51–58. IEEE, 2018.

[56] Alex Tapscott and Don Tapscott. How blockchain is changing finance. *Harvard Business Review*, 1(9):2–5, 2017.

[57] 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. IEEE, 2018.

[58] Daniel Tse, Bowen Zhang, Yuchen Yang, Chenli Cheng, and Haoran Mu. Blockchain application in food supply information security. In *2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 1357–1361. IEEE, 2017.

[59] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[60] Ting Yu, Laks VS Lakshmanan, Divesh Srivastava, and HV Jagadish. Compressed accessibility map: Efficient access control for XML. In *VLDB'02: Proc. of the 28th International Conference on Very Large Databases*, pages 478–489. Elsevier, 2002.

[61] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. Atomic commitment across blockchains. *arXiv preprint arXiv:1905.02847*, 2019.

[62] Guy Zyskind, Oz Nathan, et al. Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE Security and Privacy Workshops*, pages 180–184. IEEE, 2015.