Technical Report
Fides: Managing Data on Untrusted Infrastructure

Sujaya Maiyya Danny Hyun Bum Cho Divyakant Agrawal Amr El Abbadi
UC Santa Barbara
{sujaya_maiyya, hyunbumcho, divyagrawal, elabbadi}@ucsb.edu

ABSTRACT

Significant amounts of data are currently being stored and managed on third-party servers. It is impractical for many small scale enterprises to own their private datacenters, hence renting third-party servers is a viable solution for such businesses. But the increasing number of malicious attacks, both internal and external, as well as buggy software on third-party servers is causing clients to lose their trust in these external infrastructures. While small enterprises cannot avoid using external infrastructures, they need the right set of protocols to manage their data on untrusted infrastructures. In this paper, we propose ***TFCommit***, a novel atomic commitment protocol that executes transactions on data stored across multiple untrusted servers. To our knowledge, TFCommit is the first atomic commitment protocol to execute transactions in an untrusted environment without using expensive Byzantine replication. Using TFCommit, we propose an *auditable* data management system, ***Fides***, residing completely on untrustworthy infrastructure. As an auditable system, Fides guarantees the detection of potentially malicious failures occurring on untrusted servers using tamper-resistant logs with the support of cryptographic techniques. The experimental evaluation demonstrates the scalability and the relatively low overhead of our approach that allows executing transactions on untrusted infrastructure.

1. INTRODUCTION

A fundamental problem in distributed data management is to ensure the atomic and correct execution of transactions. Any transaction that updates data stored across multiple servers needs to be executed atomically, i.e., either all the operations of the transaction are executed or none of them are executed. This problem has been solved using commitment protocols, such as Two Phase Commit (2PC) [17]. Traditionally, the infrastructure, and hence the servers storing the data, were considered trustworthy. A standard assumption was that if a server failed, it would simply crash; and unless a server failed, it executed the designated protocol correctly.

The recent advent of cloud computing and the rise of blockchain systems are dramatically changing the trust assumptions about the underlying infrastructure. In a cloud environment, clients store their data on third-party servers, located on one or more data centers, and they execute transactions on the data. The servers hosted in the data centers are vulnerable to external attacks or software bugs that can potentially expose a client's critical data to a malign agent (e.g., credit details exposed in Equifax data breach [3], breaches to Amazon S3 buckets [1]). Further, a server may intentionally decide not to follow the protocol execution, either to improve its performance or for any other self-interest (e.g., the next big cyber threat is speculated to be intentional data manipulation[2]).

The increasing popularity of blockchain is also exposing the challenges of storing data on non-trustworthy infrastructures. Applications such as supply chain management [23] execute transactions on data repositories maintained by multiple administrative domains that mutually distrust each other. Open permissionless blockchains such as Bitcoin [32] use computationally expensive mining, whereas closed permissioned blockchains such as Hyperledger Fabric [7] use byzantine consensus protocols to tolerate maliciously failing servers. Blockchains resort to expensive protocols that tolerate malicious failures because for many applications, both the underlying infrastructure and the participating entities are untrusted.

The challenge of malicious untrustworthy infrastructure has been extensively studied by the cryptographic and security communities (e.g., Pinocchio [35] that verifies outsourced computing) as well as in the distributed systems community, originally introduced by Lamport in the famous Byzantine Agreement Protocol [24]. One main motivation for the protocol was to ensure continuous service availability in Replicated State Machines even in the presence of malicious failures.

In most existing databases, the prevalent approach to tolerate malicious failures is by replicating either the whole database or the transaction manager [15, 16, 41, 47, 4]. Practical Byzantine Fault Tolerance (PBFT) [9] by Castro and Liskov has become the predominant replication protocol used in designing data management systems residing on untrusted or byzantine infrastructure. These systems provide fault-tolerance in that the system makes progress in spite of byzantine failures; the replication masks these failures and ensures that non-faulty processes always observe correct and reliable information. *Fault tolerance is guaranteed only if at most one third of the replicas are faulty [8].*

In a relatively open and heterogeneous environment knowing the number of faulty servers – let alone placing a bound on them – is unrealistic. In such settings, an alternate approach to tolerate malicious failures is *fault-detection* which can be achieved using *auditability.* Fault detection imposes no bound on the number of faulty servers – any server can fail maliciously but the failures are always detected as they are not masked from the correct servers; detection requires only one server to be correct at any given time. To guarantee fault detection through audits, tamper-proof logs have been proposed and widely used in systems such as PeerReview [18] and CATS [44].

Motivated by the need to develop a fault-detection based data management system, we make two major propositions in this paper. First, we develop a data management system, ***Fides***[1], consisting of untrusted servers that may suffer arbitrary failures in all the layers of a typical database, i.e., the transaction execution layer, the distributed atomic commitment layer, and the datastore layer. Second, we propose a novel atomic commit protocol – ***TrustFree Commitment*** (TFCommit) – an integral component of Fides that commits distributed transactions across untrusted servers while providing auditable guarantees. To our knowledge, TFCommit is the first to solve the distributed atomic commitment problem in an untrusted infrastructure without using expensive byzantine replication protocols. Although we present Fides with TFCommit as an integral component, TFCommit can be disintegrated from Fides and used in any other design of a trust-free data management.

With detection being the focus rather than tolerance of malicious failures, Fides precisely identifies the point in the execution history at which a fault occurred, as well as the servers that acted malicious. These guarantees provide two fold benefits: i) A malicious fault by a database server is eventually detected and undeniably linked to the malicious server, and ii) A benign server can always defend itself against falsified accusations. By providing auditabiity, Fides incentivises a server not to act maliciously. Furthermore, by designing a stand-alone commit protocol, TFCommit, that leverages cryptography, we take the first step towards developing a full-fledged data management system that fully resides in untrusted infrastructures. We believe it is critical to start with a strong and solid atomic commitment building block that can be expanded to include fault tolerance and other components of a transaction management hierarchy.

Section 2 provides the necessary background used in developing a trust-free data management system. Section 3 discusses the architecture, system, and failure models of Fides. Section 4 describes the auditable transaction model in Fides and also introduces TFCommit. Section 5 provides a few failure examples and their detection. Experimental evaluation of TFCommit is presented in Section 6, followed by related work in Section 7. Section 8 concludes the paper.

2. CRYPTOGRAPHIC PRELIMINARIES

Developing a data management system built on untrusted infrastructure relies heavily on many cryptographic tools. In this section, we provide the necessary cryptographic techniques used throughout the paper.

2.1 Digital Signatures

A digital signature, similar to an actual signature, authenticates messages. A public-key signature [37] consists of a public key, p_k, which is known to all participants, and a secret key, s_k, known only to the message author. The author, $\mathcal{A}$, signs message m using her secret key s_k. Given the message m and the signature, any receiver can verify whether the author $\mathcal{A}$ sent the message m by decrypting the signature using $\mathcal{A}$'s public key p_k. Public-key signature schemes are used to prevent forgery as it is computationally infeasible for author $\mathcal{B}$ to sign a message with author $\mathcal{A}$'s signature.

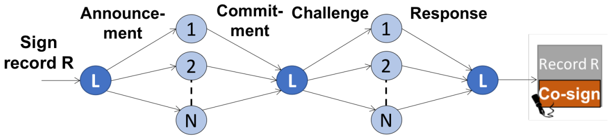


Figure 1: Collective Signing.

2.2 Collective Signing

Multisignature (multisig) is a form of digital signature that allows more than one user to sign a single record. Multisigs, such as Schnorr Multisignature [38], provide additional authenticity and security compared with single user's signature. Collective Signing (CoSi) [40], an optimization of Schnorr Multisigs, allows a *leader* to produce a record which then can be publicly validated and signed by a group of *witnesses*. CoSi requires two rounds of communication to produce a *collective signature* (co-sign) with the size and verification cost of a single signature. Figure 1 represents the phases of CoSi where L is the leader and $1, 2, .., N$ are the witnesses. The phases of CoSi are:

Announcement: The leader *announces* the beginning of a new round to all the witnesses and sends the record R to be collectively signed.

Commitment: Each witness, in response, picks a random secret, which is used to compute the Schnorr commitment, x_{sch}. The witness then sends the commitment to the leader.

Challenge: The leader aggregates all the commits, $X = \sum x_{sch}$ and computes a Schnorr challenge, $ch = hash(X|R)$. The leader then broadcasts the challenge to all the witnesses.

Response: Each witness validates the record before computing a Schnorr-response, r_{sch}, using the challenge and its secret key. The leader collects and aggregates all the responses to finally produce a Schnorr multisignature.

The collective signature provides a **proof** that the record is produced by the leader and that all the witnesses signed it only after a successful validation. Anyone with the public keys of all the involved servers can verify the co-sign and the verification cost is the same as verifying a single signature. An invalid record will not produce enough responses to prove the authenticity of the record. We refer to the original work [40] for a detailed discussion of the protocol.

2.3 Merkle Hash Tree

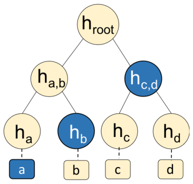


Figure 2: Merkle Hash Tree example.

A merkle hash tree (MHT) [29] is a binary tree with each leaf node labeled with the hash of a data item and each internal node labeled with the hash of the concatenated labels of its children. Figure 2 shows an example of a MHT. The

[1] *Fides* is the Roman Goddess of trust and good faith.

hash functions, h, used in MHTs are *one way hash functions* i.e., for a given input x, $h(x) = y$, such that, given y and h, it is computationally infeasible to obtain x. The hash function h must also be collision-free, i.e., it is highly unlikely to have two distinct inputs x and z that satisfies $h(x) = h(z)$. Any such hash function can be used to construct a MHT.

Data Authentication Using MHTs: MHTs are used to authenticate a set of data values [29] by requiring the prover, say Alice, to publicly share the root of the MHT, h_{root}, whose leave form the data set. To authenticate a single data value, all that a verifier, say Bob, needs from Alice is a *Verification Object* ($\mathcal{VO}$) consisting of all the sibling nodes along the path from the data value to the root. The highlighted nodes in Figure 2 form the verification object for data item a, $\mathcal{VO}(a)$, which is of size $\log_2 n$. To authenticate data item a, Alice generates the $\mathcal{VO}(a)$, and provides the value of a and $\mathcal{VO}(a)$ to Bob. Given the value of a, Bob computes $h(a)$ and uses h_b from $\mathcal{VO}(a)$ to compute $h_{a,b} = h(h(a)|h(b))$ i.e., the hash of $h(a)$ concatenated with $h(b)$. Finally, using $h_{a,b}$ and $h_{c,d}$ sent in the $\mathcal{VO}(a)$, Bob computes the root, $h_{a,b,c,d} = (h_{a,b}|h_{c,d})$. Bob then compares the computed root, $h_{a,b,c,d}$, with the root publicly shared by Alice h_{root}. Assuming the use of a collision free hash function ($h(a_1) \neq h(a_2)$ where $a_1 \neq a_2$), it would be computationally infeasible for Alice to tamper with a's value such that the h_{root} published by Alice matches the root computed by Bob using the verification object.

3. FIDES ARCHITECTURE

Fides is a data management system built on untrusted infrastructure. This section lays the premise for Fides by presenting the system model, the failure model, and the audit mechanism of Fides.

3.1 System Model

Fides is a distributed database of multiple servers; the data is partitioned into multiple shards and distributed on these servers (perhaps provisioned by different providers). Shards consist of a set of data items, each with a unique identifier. The system assumes neither the servers nor the clients to be trustworthy and can behave arbitrarily. Servers and clients are uniquely identifiable using their public keys and are aware of all the other servers in the system. All message exchanges (client-server or server-server) are digitally signed by the sender and verified by the receiver.

The clients interact with the data via transactions consisting of read and write operations. The data can be either single-versioned or multi-versioned with each committed transaction generating a new version. Every data item has an associated read timestamp r_{ts} and a write timestamp w_{ts}, indicating the timestamp of the last transaction that read and wrote the item, respectively. When a transaction commits, it updates the timestamps of the accessed data items.

We choose a simplified design for a database server to minimize the potential for failure. As indicated in Figure 3, each database server is composed of four components: an *execution layer* to perform transactional reads and writes; a *commitment layer* to atomically (i.e., all servers either commit or abort a transaction) terminate transactions; a *datastore* where the data shards are stored; and a *tamper-proof log*.

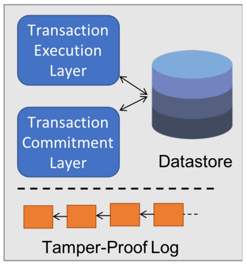


Figure 3: Components of a database server.

As individual servers are not trusted, we replace the local transaction logs used in traditional protocols such as Aries [30] with a *globally replicated tamper-proof log* (this approach is inspired by blockchain). The log – a linked-list of transaction *blocks* linked using cryptographic hash pointers – guarantees immutability. Global replication of the log guarantees that even if a subset (but not all) of the servers collude to tamper the log, the transaction history is persistent.

3.2 Failure model

In Fides, a server that fails maliciously can behave arbitrarily i.e., send arbitrary messages, drop messages, or corrupt the data it stores. Fides assumes that each server and client is computationally bounded and is incapable of violating any cryptographic primitives such as forging digital signatures or breaking one-way hash functions – the operations that typically require brute force techniques.

Let n be the total number of servers and f the maximum number of faulty servers. Fides tolerates up to $n-1$ faulty servers, i.e., $n > f$. To detect failures, Fides requires at least one server to be correct and failure-free (free of malicious, crash, or network partition failures) at a given time. This implies that the correct set of servers are not static and can vary over time. This failure model is motivated by Dolev and Strong's [11] protocol where the unforgeability of digital signatures allows tolerating up to n-1 failures rather than at most $\frac{1}{3}n$ malicious failures without digital signatures.

An individual server, comprising of four components as shown in Figure 3, can fail at one or more of the components. A fault in the *execution layer* can return incorrect values; in the *commit layer* can violate transaction atomicity; in the *datastore* can corrupt the stored data values; and in the *log* can omit or reorder the transaction history. We discuss these faults in depth in Section 4. These failures can be intentional (to gain application level benefits) or unintentional (due to software bugs or external attacks); Fides does not distinguish between the two.

A malicious client in Fides can send arbitrary messages or semantically incorrect transactions to a database server but later blame the server for updating the database inconsistently. To circumvent this, the servers store all digitally signed, unforgeable messages exchanged with the client. This message log serves as a proof against a falsified blame or when a client's transaction sends the database to a semantically inconsistent state.

3.3 Auditing Fides

Auditability has played a key role in building dependable distributed systems [42, 43, 18]. Fides provides auditablility: the application layer or an external auditor can audit

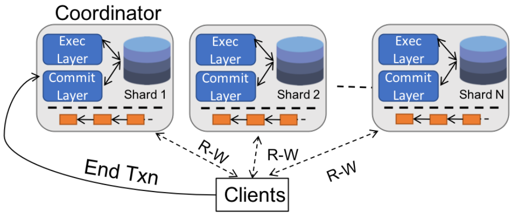


Figure 4: Client interactions in Fides

individual servers with an intent to either detect failures or verify correct behavior.

Fides guarantees that any failure, as discussed in Section 3.2, will be detected in an offline audit. Fides focuses on failure detection rather than prevention; detection includes identifying (i) the **precise point** in transaction history where an anomaly occurred, and (ii) the exact misbehaving server(s) that is irrefutably linked to a failure.

The auditor is considered to be a powerful external entity and during each audit:

(i) The auditor gathers the tamper-proof logs from all the servers before the auditing process.

(ii) Given that at least one server is correct, from the set of logs collected from all servers, the auditor identifies the *correct* and *complete* log (how is explained in detail in Section 4.4). The auditor uses this log to audit the servers.

Optimizations such as checkpointing [22] can be used to minimize the log storage space at each server; these optimizations are orthogonal and hence not discussed further. If the audit uncovers any malicious activity, a practical solution can be to penalize the misbehaving server in legal, monetary, or other forms specific to the application. This discourages a server from acting maliciously.

4. FIDES

In this section we present Fides: an *auditable* data management system built on untrusted infrastructure. The basic idea is to integrate crypotographic techniques such as digital signatures (public and private key encryption), collective signing, and Merkle Hash Trees (MHT) with the basic transaction execution in database systems. This integration results in *verifiable* transaction executions in an environment where the database servers cannot be trusted.

4.1 Overview

Figure 4 illustrates the overall design of Fides. The clients read and write relevant data by directly interacting with the appropriate database partition server (this can be accomplished by linking the client application with a run-time library that provides a lookup and directory service for the database partitions). The architecture intentionally avoids the layer of front-end database servers (e.g., Transaction Managers) to coordinate the execution of transaction reads and writes as these front-end servers may themselves be vulnerable and exhibit malicious behavior by relaying incorrect reads/writes. Hence, in Fides all data-accesses are managed directly between the client and the relevant database server.

Since data-accesses are handled with minimal synchronization among concurrent activities, the burden of ensuring the correct execution of transactions occurs when a transaction is *terminated*. We use a simplified setup where one *designated* server acts as the transaction *coordinator* responsible for terminating all transactions. The coordinator is also an untrusted database server that has additional responsibilities only during the termination phase.

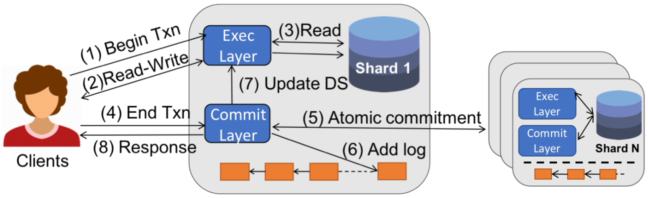


Figure 5: Transaction life-cycle in Fides

When a client application decides to terminate its transaction, it sends the termination request to the designated coordinator; all other database servers act as cohorts during the termination phase. For ease of exposition, we first present a termination protocol executed globally involving ***all*** database servers, irrespective of the shards accessed in that transaction. The global execution implies transactions are terminated *sequentially*. Later we relax this requirement and allow different coordinators for concurrent transactions.

The following is an overview of the client-server interaction: a typical life-cycle of a transaction as depicted in Figure 5.

1. Begin transaction: A client starts accessing the data by first sending a *Begin Transaction* request to all the database servers storing items read or written by the transaction.

2. Read-write request: The client then sends requests to each server indicating the data items to be read and written.

3. Read-write response: The transaction execution layer responds to a read request by fetching the data from the datastore and relaying it to the client. The write requests are buffered.

4. End Transaction: After completing data access, the client sends *End Transaction* to the coordinator which coordinates the commitment to ensure transaction correctness (i.e., serializability) and transaction atomicity (i.e., all-or-nothing property).

5. Atomic commitment: The coordinator and the cohorts collectively execute the atomic commit protocol – TFCommit– and decide either to commit or abort the transaction. The commitment produces a block (i.e., an entry in the log) containing the transaction details. If the decision is commit, then the next two steps are performed.

6. Add log: All servers append, to their local copy of the log, the same block in a consistent order, thus creating a globally replicated log.

7. Update datastore: The datastore is updated based on the buffered writes, if any, along with updating the timestamps r_{ts} and w_{ts} of the data items accessed in the transaction.

8. Response: The coordinator responds to the client informing whether the transaction was committed or aborted.

The log, stored as a linked-list of blocks, encompasses the transaction details essential for auditing. It is vital to understand the structure of each block before delving deeper into the transaction execution details. Every block stores the information shown in Table 1. Although a block can store multiple transactions, for ease of explanation, *we assume that only one transaction is stored per block.*

key	***description***
TxnId	commit timestamp of txn
R set	list of $\langle id : value, r_{ts}, w_{ts} \rangle$
W set	list of $\langle id : new\_val, old\_val, r_{ts}, w_{ts} \rangle$
$\sum roots$	MHT roots of shards
decision	commit or abort
h	hash of previous block
co-sign	a collective signature of participants

Table 1: Details stored in each block

As indicated in Table 1, each transaction is identified by its commit timestamp, assigned by the client that executed this transaction. Any timestamp that supports total ordering can be used by the client – e.g., a Lamport clock with $\langle client\_id : client\_time \rangle$ – as long as all clients use the same timestamp generating mechanism.

A block contains the transaction read and write sets consisting of three vital pieces of information: 1) the data-item identifiers that are read/written, 2) the values of items read and the new values written; the *old_val* in the write set is populated only for blind writes, and 3) the latest read r_{ts} and write w_{ts} timestamps of those data items at the time of access (read or write).

The blocks also contain: the Merkle Hash Tree roots of the shards involved in the transaction (explained more in Section 4.2); the commit or abort transaction decision; the hash of the previous block forming a chain of blocks linked by their hashes; and finally, a collective signature of all the servers (how and why are explained in Section 4.3).

The following subsections elaborate on the functionalities of a database server in a transaction life cycle. For each functionality, we first explain the correct behavior followed by the techniques to detect malicious faults.

4.2 Transaction Execution

This section describes the correct mechanism for executing transactions (reads and writes) and discusses techniques to detect deviations from the expected behavior.

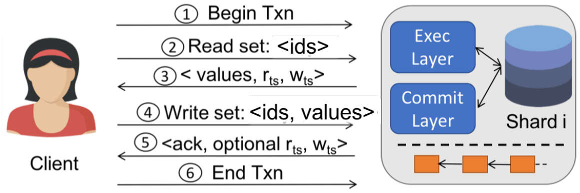

① Begin Txn
② Read set: <ids>
③ < values, r_{ts}, w_{ts}>
④ Write set: <ids, values>
⑤ <ack, optional r_{ts}, w_{ts}>
⑥ End Txn
Client
Exec Layer
Commit Layer
Shard i

Figure 6: Transaction execution in Fides

4.2.1 *Correct Behavior*

Figure 6 depicts the client-server interactions during transaction execution. With regard to transaction execution, a correct database server is responsible for the following actions: (i) return the values and timestamps of data-items specified in the read requests, and (ii) buffer the values of data-items updated in the transaction and if the transaction successfully commits, update the datastore based on the buffered writes. We explain how a correct server achieves these actions.

Reads and Writes: A client sends a *begin transaction* message to all the database servers storing the items read or written by the transaction. The client then sends a *Read* request consisting of the data-item ids to the respective servers. For example, if a transaction reads data item x from server $S1$ and item y from server $S2$, the client sends *Read(x)* to $S1$ and *Read(y)* to $S2$. The servers respond with the data values ***along with*** the associated read r_{ts} and write w_{ts} timestamps.

The client then sends the *Write* message with the data-item ids and their updated values to the respective servers. For example, if a transaction writes data item x in server $S1$ with value 5 and item y in server $S2$ with value 10, the client sends *Write(x,5)* to $S1$ and *Write(y,10)* to $S2$. The servers buffer these updates and respond with an acknowledgement. To support blind writes, the acknowledgement includes the old values and associated timestamps of the data-items that are being written but were not read before.

After completing the data accesses, the client sends the *end transaction* request – sent only to the designated coordinator – consisting of the read and the write set: a list of data item ids, the corresponding timestamps r_{ts} and w_{ts} returned by the servers, and the values read and the new values written. The coordinator then executes TFCommit among all the servers to terminate (commit or abort) the transaction (explained in detail in Section 4.3). If all the involved servers decide to commit the transaction, each involved server constructs a Merkle Hash Tree (MHT) (Section 2.3) of its data shard with all the data items – with updated values – as the leaves of the tree and with the root node $root_{mht}$. The read and write sets and MHT roots become part of the block in the log once the transaction is committed.

Updating the datastore: If the transaction commits, the servers involved in the transaction update the data values in their datastores based on the buffered writes. The servers also update the read and write timestamps of the data items accessed in the transaction to the transaction's commit timestamp.

The data can be single-versioned or multi-versioned. For multi-versioned data, when a transaction commits, a correct server additionally creates a new version of the data items accessed in the transaction *while maintaining the older versions*. Although an application using Fides can choose between single-versioned or multi-versioned data, multi-versioned data can provide ***recoverability***. If a failure occurs, the data can be reset to the last sanitized version and the application can resume execution from there.

4.2.2 *Detecting Malicious Behavior*

With regard to transaction execution, a server may misbehave by: (i) returning inconsistent values of data-items specified in the read requests; and (ii) buffering incorrect values of data-items updated in the transaction or updating the datastore incorrectly.

(i) Incorrect Reads: All faults in Fides are detected by an auditor during an audit. As mentioned in Section 3.3, during an audit, the auditor collects the log from all servers and constructs the correct and complete log.

To detect an incorrect read value returned by a malicious server, the auditor must know the expected value of the data-item. The read and write sets in each log entry contains the information on the updated value of a written item and the read value of a read item. Note that in our simplifying assumption (which will be relaxed later), each block contains *only one* transaction and the transactions are committed sequentially with the log reflecting this sequential order. By traversing the log, at each entry, the auditor knows the most recent values of a given data item. We leverage this to identify incorrectly returned values.

Lemma 1: The auditor detects an incorrect value returned for a data item by a malicious server.

Proof: Consider a transaction T_i that committed at timestamp ts_i and stored in the log at block b_i. Assume transaction T_i read an item x and updated it. Let b_j be the first block after b_i to access the same data item x – where $j > i$, indicating that transaction T_j in b_j committed ***after*** the transaction T_i in b_i. The read value of x in b_j must reflect the value written in b_i; if the values differ, an anamoly is detected. □

(ii) Incorrect Writes: The effect of incorrectly buffering a write or incorrectly updating the datastore is the same: the datastore ends up in an inconsistent state. The definition of incorrect datastore depends on the type of data: for single versioned data, the latest state of data (data values and timestamps) in the datastore is incorrect; for multi-versioned data, one or more versions of the data are incorrect. We discuss techniques to detect incorrect datastore for both types of data.

To detect an inconsistent datastore, we use the data authentication technique proposed by Merkle [29] discussed in Section 2.3. To use this technique, the auditor requires the read and written values in each transaction and the resultant Merkle Hash Tree (MHT) root – all pieces of information stored within each block.

Multi-versioned data: For multi-versioned data, the audit policy can involve auditing a single version chosen arbitrarily or exhaustively auditing all versions starting from either the first version (block 0) or the latest version. We explain auditing a single version, which can easily be extended to exhaustively auditing all versions.

Let T_i be a transaction committed at timestamp ts that read and wrote data item x stored in server S_k. Assume the auditor audits server S_k at version ts. Once the auditor notifies the server about the audit, the server constructs the Merkle Hash Tree with the data at version ts as the leaves; S_k then shares the *Verification Object* $\mathcal{VO}$– consisting of all the sibling nodes along the path from the data x to the root – with the auditor.

The log entry corresponding to transaction T_i stores the value read for item x and the new value written. The auditor uses (i) the $\mathcal{VO}$ sent by S_k, and (ii) the hash of x's value stored in the write set of the log, to compute the expected MHT root for the data in S_k (discussed in Section 2.3). The auditor then compares the computed root with the one stored in the log. A mismatch indicates that the data at version ts is incorrect.

Single-versioned data: For single versioned data, the correctness is only with respect to the latest state of the data. Hence, rather than using an arbitrary block to obtain the MHT root of server S_k, the auditor uses the latest block in the log that accessed the data in S_k to obtain the latest MHT root. The other steps are similar to multi-versioned data: the auditor fetches the $\mathcal{VO}$ based on the latest state of S_k and recomputes the MHT root to compare the root stored in the log.

Lemma 2: The auditor detects an inconsistent datastore. For multi-versioned data, the auditor detects the precise version at which the datastore became inconsistent.

Proof: Detection is guaranteed since Merkle Hash Trees (MHT) use collision-free hash functions (i.e., $h(x) \neq h(y)$ where $x \neq y$), and a malicious server cannot update a data value such that the MHT root stored in the block matches the root computed by the auditor using the verification object sent by the server. For multi-versioned datastores, the auditor identifies the precise version at which data corruption occurred by systematically authenticating all blocks in the log until a version with mismatching MHT roots is detected. □

4.3 Transaction Commitment

This section describes how transactions are terminated in Fides and presents a novel distributed atomic commitment protocol – ***TrustFree Commit*** (TFCommit) – that handles malicious failures. This section also discusses techniques to detect failures if a server deviates from the expected behavior. With regard to transaction commitment, a correct database server is responsible for the following actions: (i) Ensure transaction isolation (i.e., strict serializability); (ii) Ensure atomicity – either all servers commit the transaction or no servers commit the transaction; and (iii) Ensure *verifiable* atomicity.

4.3.1 Correct Behavior

Transaction Isolation: Transaction isolation determines how the impact of one transaction is perceived by the other transactions. In Fides, even though multiple transactions can execute concurrently, Fides provides serializable executions in which concurrent transactions seem to execute in sequence. To do so, servers in Fides abort a transaction if it cannot be serialized with already committed transactions in the log. The read r_{ts} and write w_{ts} timestamps associated with each data item is used to detect non-serializable transactions. The latest timestamps can be obtained from either the datastore or the transaction log. Similar to timestamp based optimistic concurrency control mechanism, at commit time, a server checks if the data accessed in the terminating transaction has been updated since they were read. If yes, the server chooses to abort the transaction.

Atomicity and Verifiablity: Consider a traditional atomic commit protocol that provides atomicity: Two Phase Commit (2PC) [17]. 2PC guarantees atomicity provided servers are benign and trustworthy. It is a centralized protocol where one server acts as a coordinator and the others act as cohorts. To terminate a transaction, the coordinator collects commit or abort votes from all cohorts, and decides to commit the transaction *only if* all the cohorts choose to commit, and otherwise decides to abort. The decision is then asynchronously sent to the client and the cohorts. 2PC is sufficient to ensure atomicity if servers are trustworthy; but in untrusted environments, 2PC is inadequate as a cohort or the coordinator may maliciously lie about the decision. We need to develop an atomic commitment protocol that can overcome such malicious behaviour.

To make 2PC trust-free, we combine 2PC with a multi-signature scheme, Collective Signing or CoSi (Section 2.2): a two-round protocol where a set of processes collectively sign a given record using their private keys and random secrets. CoSi guarantees that a record (or in our case block) produced by a leader (or coordinator) is validated and signed

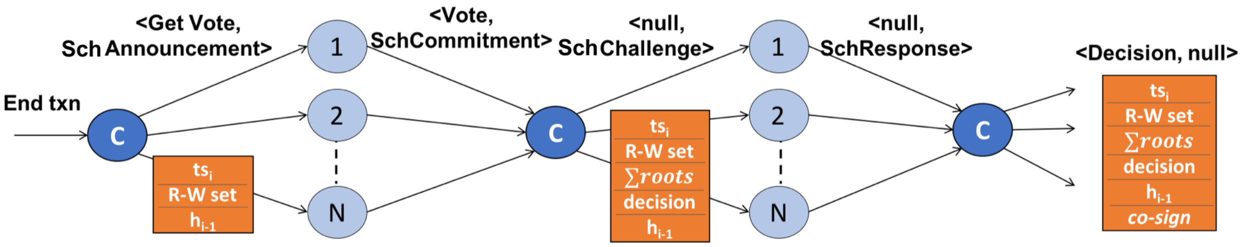


Figure 7: Different phases and block generation progress made in each phase of TFCommit

by all the witnesses (or cohorts) and that *if any of the involved processes lied in any of the phases, the resulting signature will be incorrect.* A signature is bound to a single record; any process with the public keys of all the processes can verify whether the signature is valid and *corresponds* to that record.

We propose a novel approach of integrating 2PC with CoSi to achieve the atomicity properties of 2PC *and* the verifiable properties of CoSi. The basic idea is that the coordinator, similar to 2PC, collects commit or abort votes from the cohorts, forms a decision, and encapsulates the transaction details including the decision in a block. The coordinator then sends the block to be verified and collectively signed by the cohorts. An incorrect block (either with inaccurate transaction details or wrong decision) produced by a malicious coordinator will not be accepted by correct servers, thus resulting in an invalid signature that can be easily verified by an auditor.

A successful round of TFCommit produces a block to be appended to the log *in a consistent order* by all servers. For ease of exposition, this section presents TFCommit with two main assumptions: (i) the transactions are committed sequentially to avoid forks in the log; and (ii) all servers participate in transaction termination – even the servers that did not partake in transaction execution – to have identical block order in their logs. In Section 4.6 we relax these assumptions and discuss various techniques to scale TFCommit.

Recall from Table 1 all the details stored in each block. Once a block is cosigned and logged by all servers, it is immutable; hence, all the details must be filled in during different phases of TFCommit. However, to ensure atomicity and verifiability of TFCommit, we only need the transaction id, its decision, and the co-sign. Other details such as the *Read* and *Write* sets, Merkle Tree roots, and hashes are necessary to detect other failures including isolation violation and data corruption.

The protocol:
A client, $\mathcal{A}$, upon finishing transaction execution, sends a signed $\mu = \langle end\_transaction(T_{id}, ts_i, R\ set\text{-}Wset)\rangle_{\sigma\mathcal{A}}$ request to the coordinator, where T_{id} is a unique transaction id and ts_i is a client-assigned commit timestamp of the transaction. The request also includes $R\ set\text{-}Wset$: the read and write sets consisting of data item ids, values read and new values written, r_{ts}, and w_{ts}. The servers ignore any *end transaction* request with a timestamp lower than the latest committed timestamp.

TFCommit is a 3-round protocol involving 5 phases of communication as shown in Figure 7. Since TFCommit merges 2PC with CoSi, we indicate each phase by a mapping of <2PC phase, CoSi phase>. Figure 7 shows the phases as well as the progress made in constructing the block at each phase. The phases of TFCommit are:

1) <GetVote, SchAnnouncement>: Upon receiving the $\mu = \langle end\_\ transaction(T_i, ts_i, R\ set\text{-}Wset)\rangle_{\sigma\mathcal{A}}$ request from the client, to commit transaction T_i, the coordinator $\mathcal{C}$ prepares a partially filled block, $b_i = [ts_i, Rset\ \text{-}\ Wset, h_{i-1}]$, containing the commit timestamp, read and write sets, and hash of the previous block. $\mathcal{C}$ then encapsulates the signed client request μ and sends the $\langle get\_vote(b_i, \mu)\rangle_{\sigma C}$ message to all the cohorts.

2) <Vote, SchCommitment>: Every cohort $\mathcal{H}$ verifies both the get_vote message and the encapsulated client request, and computes the Schnorr-commitment (x_{sch}) for CoSi. Then, *only the cohorts that are part of the transaction*, perform the following actions. A cohort involved in the transaction locally decides whether to commit or abort the transaction. If the cohort locally decides to commit, then it constructs a Merkle Hash Tree (MHT) (Section 2.3) of its shard with all the data items as leaves of the MHT and with the root node $root_{mht}$. The MHT reflects all the updates in T_i assuming that T_i be committed; since MHT computation is done in memory, the datastore is unaffected if T_i eventually aborts. (The MHT root is required for datastore authentication, as explained in Section 4.2.2.) The involved cohorts then send $\langle vote(decision, root_{mht}, x_{sch})\rangle_{\sigma\mathcal{H}}$ whereas the cohorts not part of the transaction send $\langle vote(x_{sch})\rangle_{\sigma\mathcal{H}}$ to the coordinator. As the coordinator is also involved in co-signing, it produces the appropriate vote message.

3) <null, SchChallenge>: In this phase, the coordinator $\mathcal{C}$ collects all the cohort responses and checks if any cohort (or itself) involved in the transaction decided to abort. If none, it chooses commit, otherwise abort. It then aggregates all the MHT roots of the involved cohorts ($roots = \sum root_{mht}$), and fills the roots field in the block b_i along with the decision field. If any involved cohorts chose abort, the respective roots will be missing in the block. Finally, the coordinator aggregates the Schnorr-commitments $X_{sch} = \sum x_{sch}$ from all the servers and computes the Schnorr-challenge by concatenating and hashing X_{sch} with b_i *i.e.*, $ch = h(X_{sch}||b_i)$. The coordinator then sends $\langle challenge(ch, X_{sch}, b_i)\rangle_{\sigma C}$ to all cohorts.

4) <null, SchResponse>: In this phase, every cohort, $\mathcal{H}$, checks if the decision within the block b_i is abort, and if so, b_i should have some missing roots; if the decision is commit, b_i should have all the roots from the involved servers. Every involved cohort that sent the MHT root in the *vote* phase verifies if its corresponding root in the block is the same as the one it sent. Cohorts also verify whether a potentially malicious coordinator computed the challenge, *ch*, correctly by hashing the concatenated X_{sch} and b_i, both of which were sent in the challenge message. A cohort then computes the Schnorr-response r_i using its secret key and the challenge *ch*, and sends $\langle response(r_i)\rangle_{\sigma\mathcal{H}}$ to the coordinator.

5) <Decision, null>: The coordinator collects all the Schnorr-responses and aggregates them, $R_{sch} = \sum r_{sch}$, to form the collective signature represented by $\langle ch, R_{sch}\rangle$. Intuitively, the challenge *ch* is computed using the block; and the Schnorr-response R_{sch} requires the private keys of the servers, thus the signature binds the block with the public keys of the servers. The coordinator then updates the *co-sign* field in the block and sends the finalized block to the client and the cohorts. If the decision is commit, all servers append block b_i to their log and update their respective datastores.

The client, with the public keys of all the servers, verifies the co-sign before accepting the decision – even an aborted transaction must be signed by all the servers. If the verification fails, the client detects an anomaly and triggers an audit, which may halt the progress in the system.

TFCommit, similar to 2PC, can be blocking if either the coordinator or any cohort fails (crash or malicious). TFCommit can be made non-blocking by adding another phase that makes the chosen value available, as in the case of Three Phase Commit [39]; we leave this extension for future work.

4.3.2 Detecting Malicious Behavior

A correct execution of TFCommit ensures serializable transaction isolation, atomicity, and verifiable commitment. However, a malicious server can (i) violate the isolation guarantees by committing non-serializable transactions; (ii) a malicious coordinator can break atomicity by convincing some servers to commit a transactions and others to abort; or (iii) a server can send wrong cryptographic values during co-signing to violate verifiability.

Lemma 3: The auditor detects serializablity violation.

Proof: Transaction execution is based on executing read and write operations in the timestamp order. The transactions are ordered based on the timestamps, which are monotonically increasing. If a transaction has done a conflicting access inconsistent with the timestamp order, it leads to one of the following conflicts: 1) RW-conflict: a transaction with a smaller timestamp read a data-item with a larger timestamp; 2) WW-conflict: a transaction with a smaller timestamp wrote a data-item that was already updated with a larger timestamp; 3) WR-conflict: a transaction with a smaller timestamp wrote a data-item after it was read by a transaction with a larger timestamp. For each transaction audited, the auditor verifies if any of the above violations exist, and if so, the auditor detects the server responsible for the violation to be misbehaving. This is equivalent to verifying that no cycle exists in the Serialization Graph of the transactions being audited. □

Lemma 4: The auditor or a correct server detects incorrect cryptographic values for CoSi sent by a malicious server – which hampers verifiablity of TFCommit.

Proof: If any server sends an incorrect cryptographic value used for co-signing, this results in an invalid signature, and the original work CoSi [40] guarantees identifying the precise server that computed the crytographic values incorrectly. Since TFCommit incorporates CoSi, it inherits this guarantee from CoSi. Intuitively, in the *schResponse* phase, the coordinator can identify if the signature is invalid, in which case, it can check partial signatures produced by excluding one server at time and detect the precise server without which the signature is valid. The coordinator is incentivised to perform this rigorous check because if the signature is invalid, the auditor suspects the coordinator for producing an incorrect block. We refer to the original work [40] that discusses the proof in depth. □

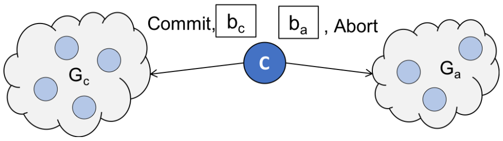


Figure 8: Atomicity violation of TFCommit

Lemma 5: The auditor or a correct server detect atomicity violation of TFCommit.

Proof: Recall that the coordinator $\mathcal{C}$ collects votes in phase two of TFCommit, forms the decision, and sends the partial block containing the decision in the *challenge* message. Consider Figure 8 where a malicious coordinator sends block b_c with commit decision to group G_c and block b_a with abort decision to group G_a. More precisely, the coordinator sends $\langle challenge(ch, X_{sch}, b_c)\rangle_{\sigma C}$ to G_c (X_{sch} is the aggregated Schnorr-commits) and $\langle challenge(ch, X_{sch}, b_a)\rangle_{\sigma C}$ to G_a. Since the decision is part of the block, the two blocks b_c and b_a have to be different if the coordinator violates atomicity. But with respect to the challenge *ch*, there are two possibilities, both producing invalid signatures:

- *Case 1*: Coordinator sends the same challenge *ch* computed using block b_c (or b_a) to both groups.

Any correct server in the group G_a will recompute the challenge using the block it received, b_a, and immediately recognize that the challenge sent by the coordinator does not correspond to the block b_a. (Alternatively, if the coordinator used b_a to compute the challenge *ch*, then servers in G_c will detect the anomaly.) Even if the servers in one group, say G_a, collude with the coordinator and do not expose the anomaly, the challenge *ch* corresponds only to block b_c. The auditor, while auditing a server in group G_a, detects that the co-sign in block b_a is invalid as it does not correspond to that block.

- *Case 2*: Coordinator sends the challenge *ch* computed using block b_c to group G_c and the challenge ch' computed using block b_a to group G_a.

In the final step of TFCommit, the servers in group G_c will use *ch* to compute the Schnorr-response, whereas the servers in group G_a will use ch' to compute the Schnorr-response. Given that the final collective signature can be tied only to a single block, the co-sign does not correspond to either b_c or b_a, hence producing a wrong signature. □

The coordinator or a cohort **can never force** all servers to commit if at least one server decides to abort a transaction. For committed transaction, the transaction block must contain MHT roots from all the involved servers; for aborted transactions, the block should have at least one MHT root missing. Assume a server S_b chooses abort and hence, does not send its MHT root. If the coordinator produces a fake root for server S_b, the server will detect it in the *schResponse* phase. And in case server S_b colludes with the coordinator by either not exposing the fake root or by producing a fake root itself, the datastore verification (discussed in Section 4.2.2), which uses MHT roots, will fail for server S_b. An involved server (coordinator or cohort) can only force an abort on all servers by choosing to abort the transaction, which is tolerable as the decision will be consistent across all servers and will not violate the atomicity of TFCommit.

4.4 Transaction Logging

The transaction log in Fides is a tamper-proof, globally replicated log. When a transaction commits after a successful round of TFCommit, all servers append the newly produced block to their logs.

Detecting Malicious Behavior: One or more faulty servers can collude (but not all at once) to (i) tamper an arbitrary block, (ii) reorder the blocks, or (iii) omit the tail of the log (last few blocks). The auditor collects logs from all the servers and uses the collective signature stored in each block to detect an incorrect log.

Lemma 6: Given a set of logs collected from all servers, the auditor detects all incorrect logs – logs with arbitrary blocks that are modified or logs with reordered blocks.

Proof: The collective signature in each block prevents a malicious server from manipulating that block once it is appended to the log. The signature is tied specifically to one block and if the contents of the block are manipulated, the signature verification will fail. One or more malicious servers cannot tamper with an arbitrary block successfully without the cooperation of *all* the servers. And since the hash of the previous block is part of a log entry, unless *all* the servers collude, the blocks cannot be successfully re-ordered. □

Lemma 7: Given a set of logs collected from all servers, the auditor detects all incomplete logs – logs with missing tail entries.

Proof: A subset of servers cannot successfully modify arbitrary blocks in the log (proof in Lemma 6) but they can omit the tail of the log. During an audit, the auditor gathers the logs from all the servers. At least one correct server exists with the complete log – which can easily be verified for correctness by validating the collective signature and hash pointer in each block. The auditor uses this complete and verified log to detect that one or more servers store an incomplete log. □

4.5 Correctness of Fides

Definition 1: *Verifiable ACID properties*

In transaction processing, ACID refers to the four key components of a transaction:

i) A̲tomicity: A transaction is an atomic unit in that either all operations are executed or none.

ii) C̲onsistency: Data is in a consistent state before and after a transaction executes.

iii) I̲solaiton: When transactions are executed concurrently, isolation ensures that the transactions seem to have executed sequentially.

iv) D̲urability: If a transaction commits, its updates are persistent even in the presence of failures.

We define *v-ACID* as the ACID properties that can be verified. *v-ACID* indicates that a database system provides verifiable evidence that the ACID guarantees are upheld. This definition is useful when individual database servers are untrusted and may violate ACID – in which case the system must allow verifying and detecting the violations.

Theorem 1: *Fides provides Verifiable ACID guarantees.*

Proof: Fides guarantees that an external auditor can verify if the database servers provide ACID guarantees or not.

The first step in the verification is for the auditor to obtain a *correct* and *complete* log. Given the assumption that at least one server is correct at a given time, Lemmas 6 and 7 prove that during an audit, the auditor always identifies the correct and complete log.

Lemma 5 proves that *A̲tomicity* violation is verifiable; Lemma 2 proves that the auditor verifies if the effect of a transaction resulted in an inconsistent database when a server buffers inconsistent writes, i.e., verifiable *C̲onsistency*; Lemma 3 proves that the *I̲solation* guarantee which ensures serializable transaction execution is verifiable; and finally, Lemmas 1 and 2 verify if the effects of committed transactions are *D̲urable*. Hence, an auditor verifies whether the servers in Fides uphold ACID properties.

Note that multiple ACID violations can exist in the transaction execution. Since the log is sequential, the auditor identifies the first occurrence of any of these violations and the blocks after that need not be audited since everything following that violation can be incorrect and hence irrelevant to a correct execution. □

4.6 Scaling TFCommit protocol

The TFCommit protocol discussed in Section 4.3 makes simplifying assumptions that each block contains a single transaction and a globally designated coordinator terminates all transactions which requires participation fromm all servers. This makes TFCommit expensive as any server not involved in a transaction must also participate in its termination. In this section we provide an intuitive overview of how to scale TFCommit.

To scale TFCommit, two aspects can be enhanced: (i) Allow multiple transactions to commit simultaneously by storing multiple transactions in a block, and (ii) Reduce the number of servers participating in transaction termination to only the servers involved in that transaction.

Extending each block to contain multiple transactions is straight-forward. The coordinator collects and inserts a set of *non-conflicting* client generated transactions and orders them within a single block at the start of TFCommit. Once the protocol begins, the coordinator or any other server cannot re-order the transactions within the block (the argument is similar to Lemma 4). This technique allows each execution of TFCommit to commit multiple transactions. In our evaluations in Section 6, we store multiple transactions in each block.

To reduce the number of servers participating in transaction termination, servers are divided into small dynamic *groups*. The servers accessed by a transaction forms one group, in which one server acts as the coordinator to terminate that transaction (instead of one globally designated coordinator). Each group executes TFCommit internally and upon a successful execution, the coordinators of each group publish the block to all other groups. The problem with such a solution is in deciding the order of blocks *across* groups such that all the servers maintain a consistently ordered transaction log.

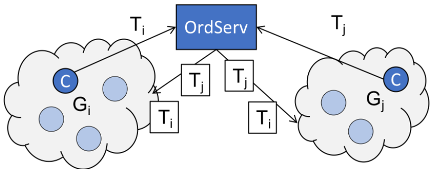


Figure 9: Scaling TFCommit.

There are multiple ways to solve the ordering problem. Figure 9 depicts a scalable solution that abstracts the ordering of blocks as a service (OrdServ). The figure shows two groups of servers G_i and G_j, each accessed by transactions T_i and T_j respectively. The OrdServ component is responsible for atomically broadcasting a single stream of blocks, each generated by TFCommit executed in different groups of servers. OrdServ can use a byzantine consensus protocol such as PBFT [9] among the coordinators to consistently order blocks; or it can be an off-the-shelf application such as Apache Kafka, used to provide ordering service in a recent work, Veritas [5]. OrdServ is also responsible for chaining the blocks i.e., the coordinators of the groups do not fill in the hash of previous block, rather it is filled by the OrdServ. There are two possible scenarios regarding the groups:

- $G_i \cap G_j = \varnothing$: If any two groups of servers have no overlapping server, there is no dependency between the two blocks of transactions T_i and T_j, and OrdServ can order them in any way and broadcast a consistent order.
- $G_i \cap G_j \neq \varnothing$: If any two groups have a non-empty intersection, then transactions T_i and T_j may have a dependency order (e.g., T_j wrote a data item after T_i read it); the OrdServ should ensure that the transaction log reflects this dependency between the published blocks.

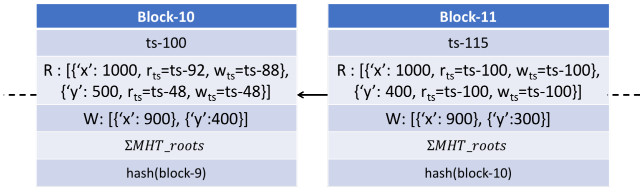
Although there is flexibility in choosing OrdServ, it is important to choose a solution that maintains local transaction order (within a group) across the globally replicated log. Solutions such a ParBlock [6] track the transaction dependency order and maintains that order while publishing blocks. We plan to integrate ParBlock with TFCommit as future work.

5. FAILURE EXAMPLES

In this section we discuss various malicious failures and safety violation scenarios and explain how the failures are detected. The failure model of Fides permits a server to misbehave but captures enough details in the transaction log for an auditor to detect the malicious failures as well as the failing servers.

Scenario 1: Incorrect Reads

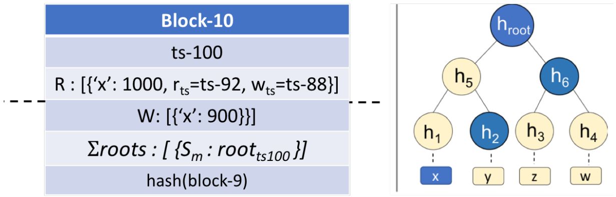
A malicious server can respond with incorrect values for the data items read in the read requests. We use Lemma 1 to detect this.



Block-10
ts-100
R : [{'x': 1000, r_{ts}=ts-92, w_{ts}=ts-88}, {'y': 500, r_{ts}=ts-48, w_{ts}=ts-48}]
W: [{'x': 900}, {'y':400}]
ΣMHT_roots
hash(block-9)

Block-11
ts-115
R : [{'x': 1000, r_{ts}=ts-100, w_{ts}=ts-100}, {'y': 400, r_{ts}=ts-100, w_{ts}=ts-100}]
W: [{'x': 900}, {'y':300}]
ΣMHT_roots
hash(block-10)

Figure 10: Isolation guarantee violation example.



Block-10
ts-100
R : [{'x': 1000, r_{ts}=ts-92, w_{ts}=ts-88}]
W: [{'x': 900}}]
Σroots : [{S_m : $root_{ts100}$ }]
hash(block-9)

Figure 11: Data corruption example

Figure 10 gives an example of incorrect reads. Assume that the severs store bank details and there are two transactions T_1 and T_2 deducting $100 from two accounts, x and y. Block-10 contains T_1 and Block-11 contains T_2. T_1 reads two data items: one with id x, value 1000, r_{ts} = *ts-92*, and w_{ts} = *ts-88*, and the second with id y, value 500, r_{ts} = *ts-48*, and w_{ts} = *ts-48*. T_1 updates x to $900 and y to $400, and upon commitment, it also updates their r_{ts} and w_{ts} to *ts-100*. Any transaction executing after this must reflect the latest data. But T_2, committing at timestamp *ts-115*, has incorrect value of $1000 for x (but up-to-date timestamps). This indicates that the server storing data items x is misbehaving by sending incorrect read values.

Scenario 2: Incorrect Block Creation

While executing TFCommit to terminate a transaction T_i, a malicious coordinator can add an incorrect Merkle Hash Tree (MHT) root of a benign server S_b in the block; this can cause audit failure of S_b (as Lemma 2 uses MHT roots to detect datastore corruption). But such an attempt will be detected by the benign server, as proved in Lemma 5.

In the *vote* phase of TFCommit, explained in Section 4.3, server S_b sends the MHT root corresponding to transaction T_i to the coordinator. If the coordinator stores an incorrect MHT root or a correct root but corresponding to an older transaction T_{i-1}, S_b can detect this in the *schResponse* phase of TFCommit. and not cooperate to produce a valid co-sign.

Scenario 3: Data corruption

A server may corrupt the data stored in the datastore, essentially not reflecting the expected changes requested by the clients. We assume a multi-versioned datastore in this example and use Verification Objects $\mathcal{VO}$ and MHT roots to detect datastore corruption, as proved in Lemma 2. Consider a transaction T_i committed at timestamp *ts-100* and updated a data item x stored in S_m. Figure 11 indicates the data stored in server S_m that is being audited at version *ts-100*. The auditor fetches the corresponding block (block 10) from the log and extracts x's value written by T_i and the MHT root corresponding to S_m. This MHT root should reflect x's updated value.

Assume S_m was malicious and did not update x to 900. In the next step of verification, auditor asks S_m for the $\mathcal{VO}$ of data item x at timestamp *ts-100*. S_m responds with $\{h_2, h_6, h_{root}\}$ (hash values of the sibling nodes of data x in the path from leaf to root). Auditor hashes x's value stored in the block ($H(900)$) and uses h_2 sent in $\mathcal{VO}$ to compute h'_5 and further, hash h'_5 and h_6 (from $\mathcal{VO}$) to compute the expected root, h'_{root}. This computed root should match the root the root stored in the block i.e., $h'_{root} = root_{S_m - ts100}$. But since S_m did not update the value of x to 900, the root computed by the auditor will not not match the root stored in the block (assuming collision-free hash functions).

Thus data corruption at S_m, precisely at version *ts-100* is detected.

6. EVALUATION

In this section, we discuss the experimental evaluation of TFCommit. Our goal is to measure the overhead incurred in executing an atomic commit protocol on untrusted infrastructure. The focus of Fides and TFCommit is *fault detection* in a non-replicated system, hence solutions based on replication that typically use PBFT [9] are orthogonal to TFCommit.

In evaluating TFCommit, we measure the performance using two aspects: *commit latency* - time taken to terminate a transaction once the client sends end transaction request, and *throughput* - the number of transactions committed per second; TFCommit was implemented in Python. We deployed multiple database servers on a single Amazon AWS datacenter (US-West-2 region) where each server was an EC2 m5.xlarge vm consisting of 4 vCPUs, 16 GiB RAM and upto 10 Gbps network bandwidth. Unless otherwise specified in the experiment, each database server stores a single shard (or partition) of data consisting of 10000 data items.

To evaluate the protocol, we used Transactional-YCSB-like benchmark [10] consisting of transactions with read-write operations. Each transaction consisted of 5 operations on different data items thus generating a multi-record workload. The data items were picked at random from a pool of all the data partitions combined, resulting in distributed transactions. Although we presented TFCommit and Fides with the simplifying assumption of one transaction per block, in the experiments, we typically stored 100 non-conflicting transactions in each block. Every experimental run consisted of 1000 client requests and each data point plotted in this section is an average of 3 runs.

6.1 TFCommit vs. 2PC

As a first step, we compare the trust-free protocol TFCommit with its trusted counterpart Two Phase Commit [17]. TFCommit is essentially 2PC combined with the cryptographic primitives (Co-Signing and Merkle Hash Trees) which results in an additional phase due to the trust-free nature. Thus, comparing TFCommit with 2PC highlights the overhead incurred by TFCommit to operate in an untrusted setting. Both 2PC and TFCommit are implemented such that transactions are terminated and blocks are produced *sequentially* so that the log does not have forks.

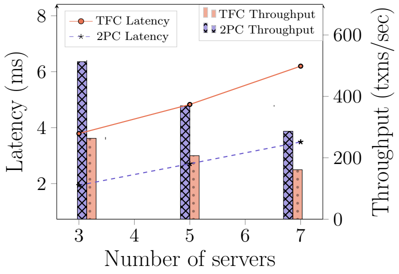


Figure 12: 2PC vs. TFCommit (TFC).

Figure 12 contrasts the performance of 2PC vs. TFCommit. We increase the number of servers and measure commit latency and throughput. In this experiment, each block stores *a single* transaction so that we can measure the overhead induced by TFCommit *per transaction*. Given that each block contains a single transaction and that blocks are generated sequentially, the servers are essentially committing one transaction after another.

As indicated in the figure, the average latency to commit a single transaction in an untrusted setting is approximately 1.8x more than a trusted environment. The throughput for 2PC is approximately 2.1x higher than TFCommit. TFCommit performs additional computations compared with 2PC: Merkle Hash Tree (MHT) updates to compute new roots after each transaction, collective signature on each block, and an additional phase. In spite of the additional computing and achieving trust-free atomic commitment, TFCommit is only 1.8x slower than 2PC. Having shown the overhead of TFCommit as compared to 2PC, the following experiments measure the performance of TFCommit by varying different parameters.

6.2 Number of transactions per block

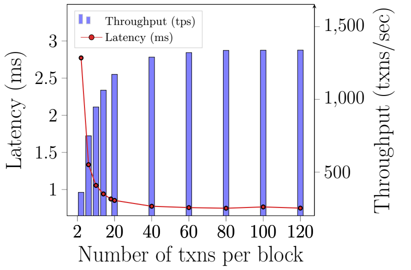


Figure 13: Varying number of transaction per block

In this experiment, we fix the number of servers to 5 and increase the load on the system by increasing the number of transactions stored within each block. Each database server consisted of 10000 data items. Figure 13 indicates the average latency to commit a single transaction and the throughput while increasing number of transactions stored within each block from 2 to 120. The latency to commit a single transaction reduces by 2.6x and the throughput increases by 2.5x when 80 or more transactions are batched in a single block. This experiment highlights that even though the blocks are produced sequentially, the performance of TFCommit can be significantly enhanced by processing multiple transactions in one block.

6.3 Number of shards

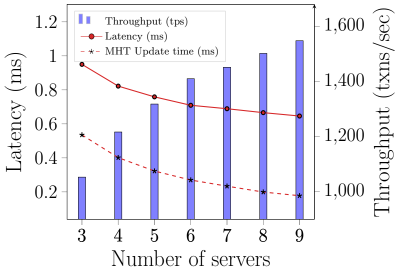


Figure 14: Varying number of servers.

In this experiment, we measure the scalability of TFCommit by increasing the number of database servers (each storing a shard of 10000 data items) from 3 to 9, while keeping the number of transaction per block constant (100 per block). Figure 14 depicts the experimental results. The throughput of TFCommit increases by 47% and the commit latency reduces by 33% when the number of servers are increased from 3 to 9. Figure 14 also shows the most expensive operation in committing transactions i.e., Merkle Hash Tree (MHT) updates. Recall from Section 4.3 that in TFCommit, termination of each transaction requires computing the updated MHT root. Given that each block has 100 transactions, which in turn consists of 5 operations each, there are 500 operations in each block. With only 3 servers, all the operations access the three shards whereas with 9 servers, the 500 operations are spread across nine shards. Thus, the load per server reduces when there are more servers, resulting in the reduction of MHT update latencies. This experiment highlights that TFCommit is scalable and performs well with increasing number of database servers.

6.4 Number of data items

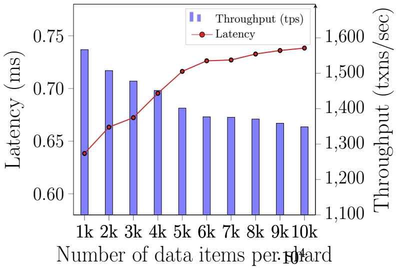


Figure 15: Varying number of data items per shard

In the final set of experiments, we measure the performance of TFCommit by varying the number of data items stored in each database server, while keeping a constant of 100 transactions per block and using 5 database servers. The number of items stored in each server increased from 1000 to 10000 to measure the commit latency and throughput of TFCommit, as shown in Figure 15. The commit latency increases by 15% and the throughput reduces by 14% with the increase in number of data items per shard. The performance fluctuation is due to the Merkle Hash Tree updates that varies with the number of data items. Updating a single leaf node in a binary hash tree with 1000 leaf nodes (data items) updates 10 nodes (from leaf to the root) and a tree with 10000 leaf nodes updates roughly 14 nodes. Thus, the performance of TFCommit decreases with increasing number of data items stored within each server.

7. RELATED WORK

The literature on databases that tolerate malicious failures is extensive [15, 16, 41, 14, 26, 36]. All of these solutions differ from *Fides* as they: assume a singe non-partitioned database, rely on *replicating* the database to tolerate byzantine failures, and some also require a trusted component for correctness. Garcia-Molina et al.[15] were the earliest to propose a set of database schemes that tolerate malicious faults. The work presents the theoretical foundations on replicating the database on enough servers to handle malicious faults but lacks a practical implementation. Gashi et al. [16] discuss fault-tolerance other than just crash failures and provide a report composed of database failures caused by software bugs. HRDB by Vandiver et al. [41] propose a replication scheme to handle byzantine faults wherein a trusted coordinator delegates transactions to the replicas. The coordinator also orders the transactions and decides when to safely commit a transaction. Byzantium by Garcia et al. [14] provides an efficient replicated middleware between the client and the database to tolerate byzantine faults. It differs from previous solutions by allowing concurrent transactions and by not requiring a trusted component to coordinate the replicas.

The advent of blockchains brought with it a set of technologies that manage data in untrusted environments. In both the open perimissionless and closed permissioned blockchains, due to lack of trust, the underlying protocols must be designed to tolerate any type of malicious behavior. But these protocols and their applications are mostly limited to crypto-currencies and cannot be easily extended for large scale distributed data management. Although permissionless blockchain solutions such as Elastico [27] Omniledger [21], and RapidChain [45] discuss sharding, it is with respect to transactions, i.e., different servers execute different transactions to enhance performance but all of them maintain copies of same data, essentially acting as replicas of a single database. These solutions differ from Fides as they focus of replicated data rather than distributed data.

In the space of transaction commitment, proposals such as [31, 47, 4, 46] tolerate malicious faults. Mohan et al. [31] integrated 2PC with byzantine fault-tolerance to make 2PC non-blocking and to prevent the coordinator from sending conflicting decisions. Zhao et al.[47] propose a commit protocol that tolerates byzantine faults at the coordinator by replicating it on enough servers to run a byzantine agreement protocol to agree on the transaction decision. Chainspace [4] proposes a commit protocol in a blockchain setting wherein each shard is replicated on multiple servers to allow executing byzantine agreement per shard to agree on the transaction decision. All these solutions require replication and execute byzantine agreement on the replicas, and hence differ from TFCommit. TFCommit uses Collective Signing (CoSi) [40], a cryptographic multisignature scheme to tolerate malicious failures during commitment. CoSi has been adapted to make consensus more efficient in blockchains, e.g., ByzCoin [20]. To our knowledge, TFCommit is the first to merge CoSi with atomic commitment.

Fides uses a tamper-proof log to audit the system and detect any failures across database servers; this technique has been studied for decades in distributed systems [42, 43, 44, 18]. In [42] and [43], Yumerefendi et al. highlight the use of accountability – a mechanism to detect and expose misbehaving servers– as a general distributed systems design. They implement CATS [44] an accountable network storage system that uses secure message logs to detect and expose misbehaving nodes. PeerReview [18] generalizes this idea by building a practical accountable system that uses tamper-evident logs to detect and irrefutably identify the faulty nodes. More recent solutions such as BlockchainDB [12], BigchainDB [28], Veritas [5] and [13] use blockchain as a tamper-proof log to store transactions across fully or partially replicated databases. CloudBFT [34], on the other hand, tolerates malicious faults in the cloud by relying on tamper-proof hardware to order the requests in a trusted way.

The datastore authentication technique that uses Merkle Hash Trees (MHT) and Verification Objects was first proposed by Merkle [29]. The technique employed in Fides that enables verifing the datastore per transaction is inspired by the work of Jain et al. [19]. Their solution assumes a single outsourced database, and more importantly, it requires a central trusted site to store the MHT roots of the outsourced data and the transaction history. Fides replaces the trusted entity by a globally replicated log that stores the necessary information for authentication. Many works have looked at query correctness, freshness, and data provenance for static data but only few solutions such as [25] and [33] (apart from [19] discussed above) consider data updates. [25] and [33] discuss alternate data authentication techniques but also assume a single outsourced database.

8. CONCLUSION

Traditional data management systems typically consider crash failures only. With the increasing usage of the cloud, crowdsourcing, and the rise of blockchain, the need to store data on untrusted servers has risen. The typical approach for achieving fault-tolerance, in general, uses replication. However, given the strict bounds on consensus in malicious settings, alternative approaches need to be explored. In this paper, we propose Fides, an auditable data management system designed for infrastructures that are *not* trusted. Instead of using replication for fault-tolerance, Fides uses fault-detection to discourage malicious behavior. An integral component of any distributed data management system is the commit protocol. We propose TFCommit, a novel distributed atomic commitment protocol that executes transactions on untrusted servers. Since every server in Fides is untrusted, Fides replaces traditional transaction logs with a tamper-proof log similar to blockchain. The tamper-proof log stores all the necessary information required to audit the system and detect any failures. We discuss each component of Fides i.e., the different layers of a typical DBMS comprising of a transaction execution layer, a transaction commitment layer, and a datastore. For each layer, both correct execution and failure detection techniques are discussed. To highlight the practicality of TFCommit, we implement and evaluate TFCommit. The experiments emphasize the performance and scalability aspects of TFCommit.

9. REFERENCES

[1] Amazon S3 Bucket Breaches. `https://www.riskiq.com/blog/labs/magecart-amazon-s3-buckets/`. Accessed: 2019-07-10.

[2] Cyber Threat Data Manipulation. `https://www.theguardian.com/technology/2015/sep/10/cyber-threat-data-manipulation-us-intelligence-chief`. Accessed: 2019-07-10.

[3] Equifax Data Breach. `https://investor.equifax.com/news-and-events/news/2017/09-15-2017-224018832`. Accessed: 2017-09-15.

[4] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[5] L. Allen, P. Antonopoulos, A. Arasu, J. Gehrke, J. Hammer, J. Hunter, R. Kaushik, D. Kossmann, J. Lee, R. Ramamurthy, et al. Veritas: Shared verifiable databases and tables in the cloud. CIDR, 2019.

[6] M. J. Amiri, D. Agrawal, and A. E. Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019.

[7] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[8] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[9] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[11] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[12] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb: a shared database on blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, 2019.

[13] E. Gaetani, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone. Blockchain-based database to ensure data integrity in cloud computing environments. *ITA-SEC*, 2017.

[14] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for byzantine fault tolerant database replication. In *European Conference on Computer Systems (EuroSys)*, pages 107–122. ACM, 2011.

[15] H. Garcia Molina, F. Pittelli, and S. Davidson. Applications of byzantine agreement in database systems. *ACM Transactions on Database Systems (TODS)*, 11(1):27–47, 1986.

[16] I. Gashi, P. Popov, V. Stankovic, and L. Strigini. On designing dependable services with diverse off-the-shelf sql servers. In *Architecting Dependable Systems II*, pages 191–214. Springer, 2004.

[17] J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[18] A. Haeberlen, P. Kouznetsov, and P. Druschel. Peerreview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 41(6):175–188, 2007.

[19] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 529–540. IEEE, 2013.

[20] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 279–296, 2016.

[21] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[22] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, (1):23–31, 1987.

[23] K. Korpela, J. Hallikas, and T. Dahlberg. Digital supply chain transformation toward blockchain integration. In *proceedings of the 50th Hawaii international conference on system sciences*, 2017.

[24] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[25] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132. ACM, 2006.

[26] A. F. Luiz, L. C. Lung, and M. Correia. Byzantine fault-tolerant transaction processing for replicated databases. In *2011 IEEE 10th International Symposium on Network Computing and Applications*, pages 83–90. IEEE, 2011.

[27] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

[28] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[29] R. C. Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[30] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[31] C. Mohan, R. Strong, and S. Finkelstein. Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 89–103. ACM, 1983.

[32] S. Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[33] M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *International conference on database systems for advanced applications*, pages 420–436. Springer, 2006.

[34] R. Nogueira, F. Araújo, and R. Barbosa. Cloudbft: elastic byzantine fault tolerance. In *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pages 180–189. IEEE, 2014.

[35] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.

[36] F. Pedone and N. Schiper. Byzantine fault-tolerant deferred update replication. *Journal of the Brazilian Computer Society*, 18(1):3, 2012.

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

[38] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[39] D. Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142. ACM, 1981.

[40] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. Keeping authorities" honest or bust" with decentralized witness cosigning. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 526–545. Ieee, 2016.

[41] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[42] A. R. Yumerefendi and J. S. Chase. Trust but verify: accountability for network services. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 37. ACM, 2004.

[43] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of HotDep*, volume 5, pages 3–3. Citeseer, 2005.

[44] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *ACM Transactions on Storage (TOS)*, 3(3):11, 2007.

[45] M. Zamani, M. Movahedi, and M. Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948. ACM, 2018.

[46] H. Zhang, H. Chai, W. Zhao, P. M. Melliar-Smith, and L. E. Moser. Trustworthy coordination of web services atomic transactions. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1551–1565, 2011.

[47] W. Zhao. A byzantine fault tolerant distributed commit protocol. In *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007)*, pages 37–46. IEEE, 2007.