On the Security and Performance of Blockchain Sharding

Runchao Han*†, Jiangshan Yu*⋆, Haoyu Lin‡§, Shiping Chen†, Paulo Esteves-Veríssimo¶
*Monash University, {runchao.han, jiangshan.yu}@monash.edu
†CSIRO-Data61, shiping.chen@data61.csiro.au
‡FluiDex, chris.haoyul@gmail.com
§ZenGo X
¶KAUST, paulo.verissimo@kaust.edu.sa

***Abstract*— In this paper, we perform a comprehensive evaluation on blockchain sharding protocols. We deconstruct the blockchain sharding protocol into four foundational layers with orthogonal functionalities, securing some properties. We evaluate each layer of seven state-of-the-art blockchain sharding protocols, and identify a considerable number of new attacks, questionable design trade-offs and some open challenges. The layered evaluation allows us to unveil security and performance problems arising from a fundamental design choice, namely the coherence of system settings across layers. In particular, most sharded blockchains use different trust and synchrony assumptions across layers, without corresponding architectural guarantees. Unless a hybrid architecture were used, assuming differentiated system settings across layers can introduce subtle but severe failure syndromes or reduce the system's performance.**

1. Introduction

Bitcoin [1] introduces the concept of permissionless blockchain, where participants (aka nodes) jointly maintain a public ledger of transactions. Nodes can join and leave the system at any time, and continuously execute consensus protocols to agree on incoming transactions and append them to the ledger. Consensus protocols for permissionless blockchains usually suffer from poor scalability. For example, Proof-of-Work (PoW)-based consensus in Bitcoin processes less than ten transactions per second. Solutions for blockchain scalability and performance have been seeked primarily by improving the mechanisms used. Dramatic improvements have been achieved along this early avenue since Bitcoin (e.g. in Proof-of-Stake-based consensus [2], [3] and Layer-2 protocols [4]).

More recently, *sharding*, used earlier in other kinds of permissioned distributed transactional systems [5]–[10], has been showing promise for scaling permissionless blockchains. In recent years, several academic proposals [11]–[15] and industry projects [16], [17] on permissionless sharded blockchains have been proposed. Sharding serves as an orthogonal approach that can work with any kind of existing blockchains, and complement further previous "mechanism-oriented" advances in performance and scalability. In a nutshell, sharding aims at dividing nodes into different consensus groups (aka shards) that process transactions concurrently, so that it advances with

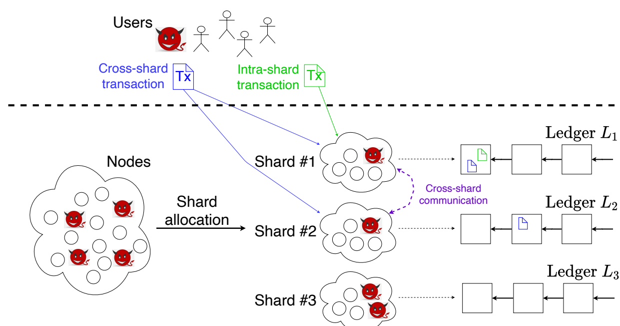


Figure 1: An example sharded blockchain with three shards. Byzantine nodes and users are labelled in red. Intra-shard and cross-shard transactions are labelled in green and blue, respectively.

less blocking or serialising situations, without compromising consistency or scalability. As shown in Figure 1, in a sharded blockchain, nodes (that are either correct or Byzantine) are allocated into different shards. Nodes in each shard maintain their ledger that contains transactions, which are submitted from users of the sharded blockchain. A transaction may involve a single shard (called intra-shard transaction) or multiple shards (called cross-shard transaction). To process cross-shard transactions, nodes in different shards may communicate with each other.

While this "divide-and-conquer" principle is intuitive, designing blockchain sharding protocols is challenging, and it still remains unknown whether existing blockchain sharding protocols are secure or practical. Similar to sharding other distributed systems, sharding blockchains introduces consistency issues across shards. Existing analysis either merely summarises designs [18], or focuses on specific components of blockchain sharding such as identity management [19] and cross-shard transactions [20], [21]. A deep security analysis of sharded blockchains, namely in the context of model choices and security-performance trade-offs, is still missing. The task is made difficult as the existing blockchain sharding papers are diverse in terms of definitions, design objectives and application scenarios.

1.1. Our contributions

In this paper, we close this gap by performing a comprehensive evaluation of sharded blockchains. Our evaluation identifies a considerable number of new attacks,

. ⋆ Jiangshan Yu is the corresponding author.

design trade-offs and open challenges. Most notably, we identify an important design choice that is overlooked by existing sharded blockchains, namely the coherence of system settings across layers.

Comprehensive evaluation of sharded blockchains. To evaluate sharded blockchains, we deconstruct the blockchain sharding problem into four foundational layers with orthogonal functionalities. The four layers, summarised in §3, include *data layer* that defines how the ledger is formatted and divided into different shards; *membership layer* that defines how nodes are allocated to different shards; *intra-shard layer* that defines how each shard processes local transactions; and *cross-shard layer* that defines how shards process cross-shard transactions. The functionality of data layer is captured by a set of verification rules; the functionality of membership layer is captured by *shard allocation* [19]; the functionality of intra-shard layer is captured by *leader election* [22] and *consensus*; and the functionality of cross-shard layer is captured by *Concurrency Control* [23] and *Atomic Commit* [24]. For each protocol layer, we suggest and analyse the required security properties and performance metrics for sharded blockchains w.r.t. their design objectives and possible attacks. Of independent interest, this can serve as an evaluation framework assisting the future development for sharded blockchains.

We select seven state-of-the-art sharded blockchains, deconstruct them into the four layers, and evaluate the layer according to our definitions. The evaluated sharded blockchains, summarised in §4, include five academic proposals (Elastico [11], Omniledger [12], Chainspace [14], RapidChain [13], and Monoxide [15]) and two industrial projects (Zilliqa [16] and Ethereum 2.0 [17]).

Evaluation results. Our evaluation shows that *none of the evaluated proposals is secure nor practical*, due to a considerable number of new attacks and questionable design choices. Most notably, our analysis (§6.1) shows that the leader election protocols in Elastico and Zilliqa cannot remain correct under a single Byzantine node, breaking their liveness and safety, respectively (Table 3a). In addition, we identify three attacks (§7.2) on the Atomic Commit protocols in Omniledger, RapidChain, and Monoxide, breaking their correctness (Table 5).

We summarise our findings in each protocol layer below. For the data layer (§5), we identify five design choices and analyse two design trade-offs regarding how the ledger is partitioned among shards and how transactions are ordered. For the intra-shard layer (§6), we show that existing sharded blockchains provide an inadequate treatment of leader election, which in fact is challenging to design and relies on strong assumptions. For the cross-shard layer (§7), we relate the cross-shard communication problem with the distributed transaction problem in distributed systems research. We apply well-established models in database literature [25]–[27] to define Concurrency Control for sharded blockchains, evaluate Concurrency Control proposals in sharded blockchains, narrow down design space of Concurrency Control for sharded blockchains and reveal the trade-offs behind different design choices. For Atomic Commit, we identify three new attacks, and show that sharded blockchains should employ Non-Blocking Atomic Commit (NB-AC) [28] to resist these attacks. We evaluate existing sharded blockchains in the NB-AC model, and show that only Elastico and Zilliqa satisfy all properties of NB-AC by using a specialised shard as centralised coordinator. This result is consistent with a recent work [20] that proves solving cross-chain/shard communication is *impossible* without a trusted third party. We also analyse the trade-off between performance and *consistency* [29] for Atomic Commit.

Insights on the coherence of system settings. Based on the system-level evaluation, we identify an overlooked design choice that greatly affects the security and performance of sharded blockchains, namely *the coherence of system settings across layers* (§8). All evaluated sharded blockchains assume different system settings for different protocol layers, without corresponding architectural guarantees. Unless a hybrid architecture were used, assuming differentiated system settings across layers can introduce subtle but severe failure syndromes, or reduce the system's performance [30]. To achieve optimal security and performance, different protocol layers should instead work under the same system setting.

1.2. Paper organisation

Section 2 provides the system model for sharded blockchains. Section 3 summarises our protocol stack. Section 4 summarises all evaluated blockchain sharding protocols. Section 5, 6 and 7 provide the analysis of data layer, intra-shard layer, and cross-shard layer, respectively. Section 8 provides the analysis on concerns across layers. Section 9 concludes this paper.

2. System model

In this section, we present the model of sharded blockchains, including system setting, system components and correctness properties.

2.1. System setting

Instead of providing a fixed system setting for the evaluated sharded blockchains, we include their system settings as a part of our evaluation. This is because sharded blockchains specify different system settings for different application scenarios. The system setting of a sharded blockchain concerns three aspects, namely network synchrony, trust model and fault tolerance degree.

Network synchrony concerns the timing guarantees of message deliveries. We consider three types of network models: *synchronous*, *partially synchronous*, and *asynchronous*. In synchronous networks, messages will be delivered within a known finite time bound; in partially synchronous networks [31], messages will be delivered within some unknown finite time bound, or a known finite time bound that becomes valid after some elapsed time (i.e., the global stabilisation time); and in asynchronous networks, messages will be delivered eventually but without a known time bound.

Trust model concerns the dependence of the protocol on particular components, in order to execute correctly. For example, a protocol may assume the existence of a trusted-third-party, i.e. a centralised identity authority to

issue and manage identities for the system, or make use of a smart contract platform.

Fault tolerance degree concerns the resilience of the protocol, in terms of the level of threat it can cope with while remaining correct. Given the permissionless settings, we consider Byzantine faults. The fault tolerance degree is quantified as the least percentage of *voting power* that the Byzantine adversary should control to break the protocol. Different protocols quantify *voting power* using different metrics, such as computing power in PoW-based consensus and deposited cryptocurrency in PoS-based consensus.

2.2. System components

Nodes are participants who jointly maintain the ledger for the system. The system is permissionless: anyone can participate in the system as a node, and nodes can join and leave the system at any time. Each node p_i has a pair of secret key and public key (sk_i, pk_i), and is identified by its public key pk_i in the system. Each node only connects to a small subset of nodes. Nodes are partitioned into different *shards*, and each node only belongs to a single shard. In each shard, nodes execute in epochs. For each epoch, nodes execute consensus to agree on some new transactions; pack agreed transactions into a block; and append the block to the ledger. To process cross-shard transactions, nodes in a shard may communicate with nodes in other shards.

Users create transactions and send them to nodes. Nodes verify incoming transactions continuously. If an incoming transaction is valid, then the node moves it to its *memory pool*, i.e., the set of pending transactions. For each epoch, nodes in each shard sample some transactions from their memory pools to agree on.

Ledger is the collection of system states jointly maintained by nodes. We use "object" to refer to the irreducible unit of system states. The ledger consists of a number of objects, or a number of transactions recording changes of objects. Each object is owned by a user, and has a unique identifier and a value (i.e., the amount of coins). Each transaction consists of input objects and output objects, plus some transaction fee. Each object can only be the output of a single transaction on the ledger. An object is inactive if the current ledger includes a transaction with this object as input. An object is active if the current ledger includes a transaction creating this object and no transaction takes this object as input.

There are two types of transactions, namely *intra-shard* transactions whose input and output objects belong to a single shard and *cross-shard transactions* whose input and output objects belong to different shards. A cross-shard transaction may be conflicted with transactions in other shards. If conflicted transactions are included in the ledger, then it will cause different kinds of anomalies. For example, two transactions having the same input object are considered conflicted. If the two transactions are included in the ledger, then the object is used twice, leading to a double-spending attack.

2.3. Model of an individual shard

A sharded blockchain includes a number m of shards, each of which works as an independent distributed ledger. We model individual shards based on well-established models of distributed ledger protocols [32]. In each shard, nodes jointly maintain a ledger formed as a blockchain, i.e., a chain of blocks. Each block records a number of transactions. Each transaction records transition of some states. To abstract the process of verifying transactions, we define an oracle $V(tx, L, \ell)$ that, given transaction tx, ledger L and height ℓ, outputs 1 if tx is a valid transaction at height ℓ of ledger L, otherwise 0. The transaction format and definition of $V(\cdot)$ depends on the data layer design, which we will analyse in §5.

Same as distributed ledger protocols [32], a shard has to satisfy two properties, namely *shard-persistence* and *shard-liveness*. Persistence formally states the tamper-resistance of blockchains. It specifies that if a valid transaction tx becomes d-deep (i.e., is followed by d consecutive blocks) in the blockchain of a correct node, then it will be "stable": all correct nodes will include tx in the same position of their blockchains, and the adversary cannot revert tx in their blockchains.

Definition 2.1 (d-Shard-Persistence)**.** A shard satisfies d-Shard-Persistence if the following holds. If a correct node in the shard includes a valid transaction tx at height ℓ on its local ledger which is at least $(\ell + d)$-long, then any correct node in the shard includes tx at the same position as the node's ledger.

Liveness formally states the censorship-resistance of blockchains. It specifies that if a valid transaction tx is submitted to correct nodes for a certain time range of generating u new blocks, then tx will eventually be stable (i.e., become d-deep in blockchains of all correct nodes), and the adversary cannot censor tx, i.e., preventing tx from being confirmed.

Definition 2.2 ((u, d)-Shard-Liveness)**.** A shard satisfies (u, d)-Shard-Liveness if the following holds. If a valid transaction tx is given as input to all correct nodes in a shard for u consecutive blocks, then all correct nodes in the shard will include tx in a block that is at least d-deep on its local ledger.

2.4. Model of a sharded blockchain

Sharded blockchain aims at scaling the blockchain system by running multiple blockchains in parallel. Let $\mathcal{L} = \{L_1, \ldots, L_m\}$ be the set of ledgers in the sharded blockchain, where m is the number of shards. First, every shard in the sharded blockchain should satisfy shard-persistence and shard-liveness.

Definition 2.3 (d-Persistence)**.** A sharded blockchain satisfies d-Persistence iff all shards satisfy d-Shard-Persistence.

Definition 2.4 ((u, d)-Liveness)**.** A sharded blockchain satisfies (u, d)-Liveness iff all shards satisfy (u, d)-Shard-Liveness.

In addition, a sharded blockchain has to satisfy *validity*, meaning that no transaction in the ledger is conflicted with each other. To abstract the process of resolving conflicts introduced by cross-shard transactions, we define oracle $C(tx_x, tx_y)$ that, given two transactions tx_x and tx_y, outputs 1 if tx_x and tx_y are conflicted, otherwise 0.

TABLE 1: The protocol stack of blockchain sharding protocols.

Protocol layer	Sub-protocol	Functionality
Cross-shard layer	Atomic Commit	Protecting correctness of cross-shard transactions
	Concurrency Control	Protecting correctness of concurrent transactions
Intra-shard layer	Consensus	Agreeing on transactions within each shard
	Leader election	Electing a leader for each shard
Membership layer	Shard allocation	Partitioning nodes into different shards
Data layer	-	Defining the ledger format

The definition of transaction conflicts and the specification of $C(\cdot)$ depend on the data layer design, which we will analyse in §5. Cross-shard communication may be required in $C(\cdot)$.

Definition 2.5 (Validity)**.** For any two correct nodes at shard k_x and k_y, let L_x and L_y be their local ledgers, respectively. If they include transaction tx_x and tx_y in the blocks at height ℓ_x and ℓ_y in L_x and L_y, respectively, then the following holds:

- $V(tx_x, L_x, \ell_x) = 1$,
- $V(tx_y, L_y, \ell_y) = 1$, and
- $C(tx_x, tx_y) = 0$.

Note that our evaluation will only show whether a sharded blockchain satisfies the properties *without* calculating the concrete degrees u and d, which depend on system parameters when instantiating the system.

3. Protocol stack

In this section, we deconstruct the sharded blockchain into four protocol layers focusing on orthogonal functionalities. As summarised in Table 1, the protocol stack consists of four layers, including 1) *data layer* that defines the ledger's format; 2) *membership layer* that partitions nodes into different shards; 3) *intra-shard layer* that agrees on transactions within a shard; and 4) *cross-shard layer* that processes cross-shard transactions.

3.1. Data layer

Data layer concerns the ledger format, i.e., how system states are represented, evolved, and stored. Apart from concerns that exist in non-sharded blockchains, the data layer raises some additional concerns on cross-shard transactions, and thus affects the cross-shard layer design in sharded blockchains. First, how transactions are ordered affects the frequency of resolving conflicts between concurrent transactions. In addition, how ledgers are partitioned among shards affects the communication overhead of verifying cross-shard transactions. Concurrency Control and Atomic Commit, the two protocols addressing these two concerns, constitute the cross-shard layer. We will analyse the design spaces and their trade-offs of the data layer in §5.

3.2. Membership layer

Membership layer is responsible for allocating nodes into different shards. This functionality is informally studied in existing blockchain sharding proposals, and has recently been formalised as the *shard allocation* protocol [19]. Shard allocation protocols are non-trivial to design. Specifically, the adversary can launch the *single-shard takeover attack* (aka 1% attack) [33], [34] by gathering its corrupted nodes to the victim shard and compromising the shard's consensus. As voting power is distributed among shards, compromising a single shard requires fewer corrupted nodes than compromising a non-sharded blockchain. To defend against the single-shard takeover attack, the shard allocation protocol has to move some nodes to other shards periodically. However, a node moving to another shard incurs non-negligible overhead, as it needs to synchronise the shard's ledger and find peers in the shard from scratch. The blockchain community refers the issue as the *reshuffling* problem [35], [36].

For completeness, we summarise existing studies of shard allocation below, and refer readers to the paper [19] for detailed analysis. A shard allocation protocol concerns all aspects of system settings in §2.1. It has four correctness properties, namely *liveness*, *allocation-randomness*, *unbiasibility*, and the optional *allocation-privacy*. Liveness specifies that for each epoch, every correct node eventually obtains a valid shard membership; γ-allocation-randomness specifies that for each epoch, every node has probability γ to stay in its current shard, and probability $\frac{1-\gamma}{m-1}$ to move to each other shard; unbiasibility specifies that every node cannot manipulate the shard allocation distribution specified in allocation-randomness; and allocation-privacy specifies that a shard membership cannot be computed by others unless the node itself reveals it. It has four performance metrics, namely *communication complexity*, *self-balance* and *operability*. Communication complexity is the amount of communication for obtaining a shard membership; self-balance is the ability that a sharded blockchain recovers from load imbalance; and operability is the overhead of nodes moving among shards. The evaluation [19] shows that no shard allocation protocol in existing sharded blockchains is fully correct or performant. Based on the evaluation, the authors [19] identify and prove a trade-off between self-balance and operability.

3.3. Intra-shard layer

Intra-shard layer is responsible for processing intra-shard transactions in this shard and cross-shard transactions involving this shard. The intra-shard layer protocol acts the same ways as a non-sharded blockchain, where nodes jointly maintain a ledger of transactions and keep agreeing on new transactions. The non-sharded blockchain protocol usually involves two subprotocols, namely *leader election* and (leader-based) *consensus*.

Figure 2 describes the execution of the intra-shard layer. For each epoch in a shard, nodes run the *leader election* protocol to elect a leader (aka *primary node*). The leader samples a subset of valid transactions from its memory pool, packs them into a block, and broadcasts this block to other peers in this shard. Nodes in this shard

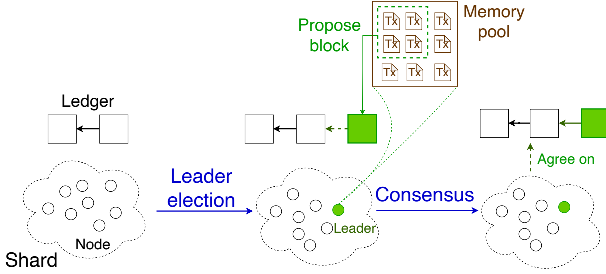
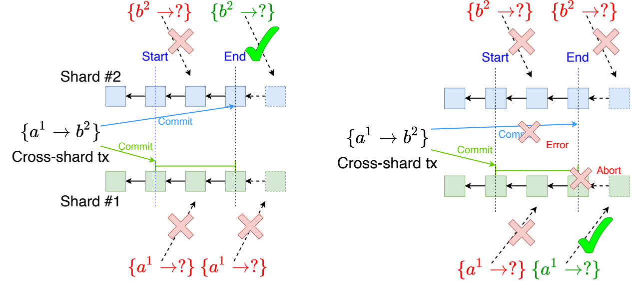


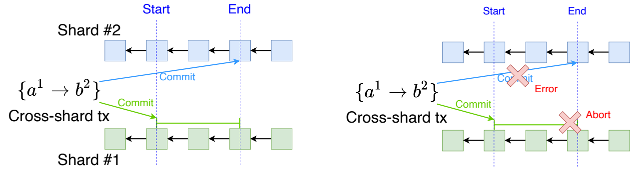
Figure 2: Intra-shard layer. Nodes run a leader election protocol to select a leader. The leader selects a subset of transactions from its memory pool, packs selected transactions into a new block, and broadcasts this block to nodes. Nodes then execute the consensus protocol to agree on this block. If agreed, nodes will append it to the ledger.



(a) Successful case. If tx is included, b^2 will be active and a^1 will be removed.

(b) Failed case. If tx is not included, a^1 will remain active and b^2 will be inactive.

Figure 3: Concurrency Control in sharded blockchains.



(a) Successful case. If shard #1 includes tx, then shard #2 has to include tx as well.

(b) Failed case. If shard #1 omits tx due to some error, then shard #2 should omit tx as well.

Figure 4: Atomic Commit in sharded blockchains.

then execute the *consensus* protocol to agree on these transactions and update states of the ledger accordingly. We will analyse required properties of leader election and consensus and evaluate designs in existing sharded blockchains in §6.

Although blockchain protocols can be leaderless, we still include leader election in our evaluation for two reasons. The first reason is the completeness of the evaluation: all our evaluated sharded blockchains use leader election protocols, except for Chainspace. The second reason is that to exclude leader election, blockchain protocols have to employ *leaderless consensus*, which is currently more of theoretical interest due to the high communication complexity and strong system setting requirements [37].

3.4. Cross-shard layer

Cross-shard layer is responsible for processing cross-shard transactions that involve multiple shards. Processing cross-shard transactions faces two major challenges, namely 1) resolving conflicts between concurrent cross-shard transactions and 2) including cross-shard transactions "atomically": eventually, a cross-shard transaction is either included or omitted in the ledgers of all involved shards. Both challenges also exist in *distributed transactions* that involve multiple computers. Existing distributed systems research [38] suggests to handle the two tasks by using *Concurrency Control* (CC) [23] and *Atomic Commit* (AC) [25], respectively.

To explain CC and AC, we will use an example cross-shard transaction tx. Let $tx = \{a^1 \to b^2\}$ be a cross-shard transaction that takes object a^1 on shard #1 as input and outputs object b^2 on shard #2. As tx involves both shard #1 and #2, it has to be included in both shards.

Concurrency Control. In non-sharded blockchains, a transaction is included in the ledger instantly once the block including it is included in the ledger. In sharded blockchains, including tx is likely to take multiple epochs, as tx are processed by two different shards that are executing independently. Within this time gap, there might be concurrent conflicted transactions attempting to access a and b. To avoid anomalies caused by conflicted transactions, the cross-shard layer has to achieve a property called *isolation* ("I" in ACID [39]), which specifies how and when changes made by a transaction become visible to other transactions. Concurrency Control (CC) [23] is a family of protocols that achieve *isolation* by properly scheduling concurrent-but-conflicted transactions.

Figure 3 describes the functionality of CC for blockchain sharding. Suppose tx is included in shard #1 earlier than in shard #2. If both shards include tx, then b^2 will be active on shard #2. If both shards discard tx, then a^1 will be active on shard #1. During the time gap, neither a^1 nor b^2 can be inputs of other transactions.

Atomic Commit. Shard #1 and #2 should have the same decision on tx: eventually, both shards should include or discard tx. If shard #1 includes but shard #2 omits tx, then a^1 is locked (i.e., cannot be spent) forever. If shard #1 omits but shard #2 includes tx, then a^1 is used as input twice, leading to a double-spending attack. To prevent these two scenarios, a cross-shard communication primitive is required. The primitive has to satisfy two properties, namely *agreement*, i.e., two shards agree on the execution results of tx and *termination*, i.e., tx will eventually be executed on both shards. Otherwise, conflicted transactions can be executed, and the sharded blockchain's validity will be broken. Atomic Commit (AC) is the family of protocols that provide *agreement*. Non-Blocking Atomic Commit (NB-AC) [28] is an AC variant that additionally provides *termination*.

Figure 4 describes the functionality of AC in the context of blockchain shards. For cross-shard transaction tx, if shard #1 includes tx, then shard #2 has to include tx as well. Otherwise, if shard #1 omits tx due to some error, then shard #2 should omit tx as well. We will analyse required properties of CC and AC in §6.

4. Existing sharded blockchains

In this section, we summarise the design of sharded blockchains that we will evaluate. As the paper focuses on permissionless settings, we choose to evaluate seven state-of-the-art permissionless sharded blockchains, including five academic proposals Elastico [11], Omniledger [12], RapidChain [13], Chainspace [14] and Monoxide [15], and two industry projects Zilliqa [16] and Ethereum 2.0 [40].

Elastico and Zilliqa. In Elastico [11], the ledger is formed as a single blockchain. Each node maintains the entire ledger. For each epoch, nodes in a special shard called *final committee* execute a Distributed Randomness Generation (DRG) protocol [41] to generate a random output. Each node solves a PoW with the random output and its public key as input. The node is allocated to a shard according to the prefix of its PoW solution. The first shard becomes the *final committee*. In each shard, nodes execute *Monarchy* [42] to elect a leader, and execute PBFT [43] to agree on the block proposed by the leader. The *final committee* gathers all blocks, merges them into a single block, and appends it to the ledger.

Zilliqa [16] is an industry project that adapts the Elastico protocol with three main optimisations. First, the random output is derived from the last block's hash rather than generated from DRG. Second, instead of using Monarchy for leader election, the node with the smallest PoW solution in a shard is elected as leader. Third, Zilliqa incorporates Collective Signing [44] with PBFT, in order to reduce communication complexity from $O(n^2)$ to $O(n)$.

Omniledger. In Omniledger [12], the ledger is partitioned into different shards. Each part of the ledger is structured as a Directed Acyclic Graph (DAG) of blocks. Each block consists of a number of objects (rather than transactions). Objects are distributed to different shards according to their IDs. For each epoch, all nodes execute the RandHound [45] DRG protocol to produce a random output. Nodes are allocated to shards randomly by a centralised identity authority. In each shard, each node runs Verifiable Random Function (VRF) over the random output and its identity, and the node with the smallest VRF output is elected as leader. The leader proposes a block, and nodes execute ByzCoinX – an optimised version of the ByzCoin [44] consensus protocol – to agree on the block.

To process cross-shard transactions, Omniledger employs the *Atomix* Atomic Commit (AC) protocol. We describe Atomix by using the cross-shard transaction $tx = \{a^1 \rightarrow b^2\}$ in §3.4 as an example. First, the user sends tx to shard #1, and requests shard #1 to lock a^1. Nodes in shard #1 verify tx, perform a ByzCoinX consensus on locking a^1, and send a *proof-of-acceptance* of a^1 to the user. The user then constructs a *unlock-to-commit* transaction consisting of the *proof-of-acceptance* of a^1, tx, and object b^2, then sends this transaction to shard #2. Nodes in shard #2 execute ByzCoinX consensus to agree on the *unlock-to-commit* transaction. If accepting this transaction, shard #2 executes another ByzCoinX consensus on adding b^2 as an active object.

RapidChain. In RapidChain [13], the ledger is partitioned into different shards. Each part of the ledger is formed as a blockchain. Each block consists of a set of transactions. Each transaction is allocated to a shard according to the input object's ID. Each node should solve a PoW to join the system. Nodes are allocated to different shards following the Commensal Cuckoo rule [46]. For each epoch, nodes in each shard elect a leader (RapidChain does not specify the leader election protocol), and execute a synchronous BFT consensus protocol [47] to agree on the block proposed by the leader. The leader is also responsible for coordinating cross-shard transactions. Given cross-shard transaction $tx = \{a^1 \rightarrow b^2\}$, the leader of shard #2 splits tx to two intra-shard transactions: $tx_a^1 = \{a^1 \rightarrow x\}$ and $tx_b^2 = \{x \rightarrow b^2\}$, then sends tx_a^1 and tx_b^2 to nodes in shard #1 and shard #2, respectively. After nodes in shard #1 agree on tx_a^1, the leader of shard #2 proposes tx_b^2.

Chainspace. Chainspace [14] is a sharded smart contract platform. In Chainspace, the ledger is partitioned into different shards. Each part of the ledger consists of a Directed Acyclic Graph (DAG) of objects, as well as transaction hashes. Objects are distributed to shards according to their IDs. Chainspace does not specify how newly joined nodes are allocated to different shards. If an existing node wants to move to another shard, then it should send a request to the system by using a transaction, and nodes vote to approve its request. Chainspace does not elect leaders for intra-shard consensus. Instead, once a node receives a transaction, this node will initiate a consensus within its shard (plus a cross-shard AC if the transaction is cross-shard) on that transaction.

To process cross-shard transactions, Chainspace employs the *S-BAC* protocol, which combines an optimistic concurrency control (OCC) protocol and an AC protocol. Unlike AC protocols in other sharded blockchains, S-BAC requires input shards to communicate with each other. Thus, we describe S-BAC using a transaction with multiple inputs in different shards. Let $tx = \{a^1, b^2 \rightarrow c^3\}$ be a transaction with two inputs a^1 on shard #1 and b^2 on shard #2, and an output c^3 on shard #3. In S-BAC, the user first sends $tx = \{a^1, b^2 \rightarrow c^3\}$ to shard #1 and #2, and the two shards verify tx with an intra-shard consensus. If valid, the two shards exchange a^1 and b^2, and execute another consensus to inactivate a^1 and b^2. Note that if a^1 or b^2 is inactivated by another transaction before the two shards, tx will be aborted. After that, the two shards send tx to shard #3, and meanwhile exchange the status of a^1 and b^2 and respond to the user. When shard #3 receives tx, shard #3 performs an intra-shard consensus on creating c^3.

Monoxide. In Monoxide [15], the ledger is partitioned into different shards. Each part of the ledger is formed as a blockchain. Each block contains a number of transactions. Each transaction can only have one input object and one output object. A transaction is allocated to a shard according to its input object's ID. Nodes can join any shards. Similar to Bitcoin, Monoxide employs PoW-based leader election and Nakamoto consensus. Existing analysis proves that Nakamoto consensus works under partially synchronous networks [48]. Nodes are allowed to do Chu-ko-nu mining, i.e., mine on multiple shards simultaneously. Monoxide takes a similar approach with RapidChain for cross-shard transactions: each cross-shard transaction is split to multiple intra-shard transactions that

are submitted individually. Monoxide assumes nodes are incentivised to process cross-shard transactions, as they want to earn the fee in these transactions. Monoxide calls this guarantee as *eventual atomicity*.

Ethereum 2.0. Ethereum 2.0 is the next generation of the Ethereum project [49], aiming at scaling Ethereum via sharding. In Ethereum 2.0 [40], the ledger is partitioned into different shards, including a beacon chain and a number of shard chains. The beacon chain is the main chain that stores cross-shard transactions, manages validators who produce and verify blocks, and generates randomness periodically using the RANDAO protocol [50]. Nodes in shard chains execute consensus and append blocks independently. Transactions belong to different smart contracts, and smart contracts are allocated to different shards according to their IDs. Each node stores a part of the ledger as well as block headers of the entire ledger. Each shard samples a subset of nodes called *validators* via the "custody game", a deposit-based weighted sortition. Nodes first deposit some coins in a special *deposit contract* to join the validator registry. With the random output from the beacon chain as entropy, the deposit contract randomly samples a number of validators and a leader from all nodes. The leader proposes a block, and validators execute the Casper [51] consensus protocol to agree on the block. To process cross-shard transactions, users submit transactions on both input shards and output shards. Given cross-shard transaction $tx = \{a^1 \rightarrow b^2\}$ between a sender and a receiver, the sender splits tx to two intra-shard transactions: $tx_a^1 = \{a^1 \rightarrow x\}$ and $tx_b^2 = \{x \rightarrow b\}$. Then, the sender sends tx_a^1 to shard #1. Once tx_a^1 is included, the shard will create a receipt consisting of the block's Merkle branch with tx_a^1, a^1, and the receiver of tx_b^2. With this receipt and block headers of shard #1, any node can verify the status of a^1. The receiver verifies the receipt, and sends tx_b^2 together with the receipt to shard #2. Shard #2 then validates tx_b^2 using the receipt, and will include tx_b^2 if valid.

5. Data layer

We consider the following design choices for the data layer.

- *Ledger unit*: the irreducible unit in a ledger (e.g., transaction or object).
- *Unit allocation*: how a ledger unit is allocated to a shard.
- *Consensus unit*: the item appended to the ledger for each consensus epoch (e.g., block or transaction).
- *Ledger partitioning*: whether each node stores a part of the ledger (sharded) or the entire ledger (replicated).
- *Ordering*: how ledger units are ordered (e.g., partial ordering or total ordering).

Table 2 summarises design choices of the data layer made by blockchain sharding proposals. Based on the evaluation, we analyse the design space and trade-offs inside these design choices.

Ledger unit: transaction v.s. object. Among these sharding protocols, only Omniledger uses object as ledger unit, and other protocols use transaction. The only advantage of storing objects is to reduce the ledger size, as the ledger does not need to record historical objects. However, without storing transactions, the history of modifications on objects will be lost, making it hard to roll back transactions. Rollback may happen in two scenarios. First, with Atomic Commit in the cross-shard layer, cross-shard transactions may be aborted halfway and rolled back. Second, when the intra-shard layer uses a probabilistic consensus protocol, e.g., Nakamoto consensus, transactions may be first included but later reverted. To add the rollback support without storing all transactions, the sharded blockchain has to store some coarse-grained historical states of objects. For example, Omniledger labels inactive objects rather than removing them from the ledger. Other approaches such as versioning objects and checkpointing [52], [53] can also be used for storing historical states of objects.

TABLE 2: Design choices of the data layer.

	Ledger unit	Unit alloc.	Consensus unit	Partitioning	Ordering
Elastico	Tx	Arbitrary	Block	Replicated	Total
Omniledger	Object	ID	Block	Sharded	Partial
RapidChain	Tx	Input ID	Block	Sharded	Partial
Chainspace	Tx	Input ID	Tx	Sharded	Partial
Monoxide	Tx	Input ID	Block	Sharded	Partial
Zilliqa	Tx	Arbitrary	Block	Replicated	Total
Ethereum 2.0	Tx	SC. ID	Block	Sharded	Partial

Consensus unit: transaction v.s. block. While in Chainspace nodes execute consensus over transactions, in other sharded blockchains nodes execute consensus over blocks. These design choices are basically parameterised by the consensus unit, i.e., block size limit, the number of transactions a block can include. Apart from existing concerns on the block size limit in non-sharded blockchains [54], the block size limit introduces some other concerns related to the cross-shard communication in sharded blockchains. On the positive side, specifying a small consensus unit increases the throughput limit. As a pair of blocks are more likely to be conflicted than a pair of transactions, small consensus units can be processed at a higher concurrency level, leading to higher throughput limit. On the negative side, specifying a small consensus unit reduces the actual throughput. With smaller consensus units, invocations of consensus and Atomic Commit are more frequent, introducing extra communication overhead. When the network capacity is limited, it may become the performance bottleneck and limit the throughput.

Thus, there is a trade-off between the throughput limit and the actual throughput, parameterised by the consensus unit size. While our evaluated proposals experiment on the relationship between actual throughput and consensus unit size in their own models, a unified analysis regarding the trade-off is still missing, and we consider it as future work.

Partitioning: sharded v.s. replicated. Within these sharding protocols, Elastico and Zilliqa replicate the ledger among shards, while other protocols divide the ledger into different shards. The ledger partitioning has a direct impact on the construction of oracle $C(tx_x, tx_y)$. Replicating ledgers is similar to *parallel chains* [55]–[58],

where the ledger consists of multiple shards, and nodes execute consensus on these shards in parallel. With all shards, nodes can verify cross-chain transactions locally, and $C(tx_x, tx_y)$ does not involve cross-shard communication [20], which we will show is challenging to solve in §7. However, replicating ledgers inevitably introduces $O(m)$ more overhead on communication and storage, where m is the number of shards.

Thus, there exists a trade-off between the complexity of cross-shard communication and the overhead of storing/synchronising ledgers, parameterised by the portion of the ledger a node should synchronise with. Ethereum 2.0 employs an in-between solution that minimises the overhead while bypassing the cross-chain communication problem: each node stores a part of the ledger as well as block headers of the entire ledger, so that any node can verify cross-shard transactions locally. Other possibilities over this trade-off are considered as future work.

Total ordering v.s. partial ordering. Among these blockchain sharding protocols, Elastico and Zilliqa enforce total ordering on transactions, and other protocols only enforce partial ordering. We observe that total ordering is not always desired for sharded blockchains. First, not all transactions need to be ordered. In permissionless blockchains, different users hold different accounts and active objects. Transactions involving different sets of users are independent of each other, and therefore can be unordered. Second, ordering transactions requires shards to synchronise with each other, which inevitably introduces extra communication overhead. For example, Elastico and Zilliqa allow nodes in a special shard to be the *final committee*. The *final committee* collects blocks from all other shards, merges them to a single block, and appends this block to the blockchain. The final committee works as a *barrier* [59] that synchronises all shards: all shards should wait before every shard produces a block.

6. Intra-shard layer

In this section, we analyse the required properties of the intra-shard layer, and evaluate existing intra-shard layer designs. As mentioned in §3, the intra-shard layer consists of two protocols, namely *leader election* and *consensus*. Given the rich literature in consensus [64]–[67], we build our analysis on consensus upon existing studies. Our evaluation shows that, leader election is overlooked by existing designs, but it usually introduces strong assumptions.

6.1. Leader election

In each shard, nodes execute the leader election protocol to elect a leader, who is responsible for proposing the next block and/or coordinating intra-shard consensus.

System setting. In addition to aspects mentioned in §2, we also evaluate the *weight* for leader election. A node's *weight* is in proportion to the chance that it is elected as the leader. For example, the weight can be computational power and financial stake in PoW-based and PoS-based leader election, respectively.

Correctness and performance metrics. We model the leader election for sharded blockchains based on existing models [22], [68]. Leader election for blockchain sharding should satisfy the following five properties:

- *Public verifiability*: given a node's public key, its leadership proof and the system state, anyone can verify whether this node is the leader at this system state.
- *Uniqueness*: after election, only one node (in a shard) can provide a valid leadership proof.
- *Unpredictability*: for any epoch t, the probability of making an accurate guess on the leader of any shard at the $(t+1)$-th epoch is in proportion to the ratio between the guessed node's weight and its shard's total weight at epoch t.
- *Fairness*: no node can manipulate its probability of being elected.
- *Termination*: for every epoch, eventually, there will be a node elected as leader.

Public verifiability allows nodes to verify the leader's identity. Uniqueness ensures that only a single node can become the leader and initiate the consensus protocol. Unpredictability and fairness prevent the adversary from corrupting the leader throughout the protocol execution. Termination prevents nodes in the sharded blockchain to lose liveness forever. The performance metric of the leader election protocol is the communication complexity.

Evaluation and analysis. We evaluate the strength of the system setting, in the understanding, from a systems theory viewpoint, that strength of a system setting is inversely proportional to its coverage. Table 3a summarises our evaluation results. Our evaluation shows that leader election is overlooked by existing proposals. Specifically, RapidChain and Ethereum 2.0 require leader election protocols but do not provide detailed specifications. In Ethereum 2.0, the leader election protocol executes upon a smart contract, which is assumed to be secure. Elastico and Zilliqa's leader election protocols cannot tolerate any Byzantine fault. Elastico uses *Monarchy* [42] for leader election, where a single Byzantine node can stall the protocol forever by withholding messages. Zilliqa elects the node with the smallest PoW solution in a shard as leader. A Byzantine node can withhold its PoW solution, allow another node to be the leader, and publish its PoW solution to revert the current consensus execution. Omniledger's leader election protocol relies on a trusted third party (TTP).

Previous studies show that leader election is challenging in permissionless settings. Calzado et al. [68] model and evaluate leader election under crash faults and dynamic mobile networks. They evaluate several leader election protocols, and show that no protocol resists against all failures. Boneh et al. [22] formally study Single Secret Leader Election (SSLE) and propose two protocols, but both protocols rely on randomness beacon and complex cryptographic primitives such as Fully Homomorphic Encryption and Indistinguishable Obfuscation.

6.2. Consensus

Given the rich literature in consensus [64]–[67], we summarise correctness properties and performance metrics for consensus protocols, and mention previously identified issues of consensus protocols used by these blockchain

TABLE 3: Evaluation of intra-shard layer. N/A means the protocol is not specified, and symbol "-" means the protocol is not needed.

(a) Leader election.

	Protocol	System setting				Correctness					Performance
		Network sync.	Trust	Fault tolerance	Weight	Public verif.	Uniqueness	Unpredictability	Fairness	Termination	Comm. compl.
Elastico	[42]	Sync.	-	0	-	✓	✓	✓	✓	✓	$O(n^2)$
Omniledger	VRF-based	Sync.	Leader	N/A	-	✓	✓	✓	✓	✓	$O(1) \sim O(n)$
RapidChain	N/A	N/A	N/A	N/A	-	N/A	N/A	N/A	N/A	N/A	N/A
Chainspace	-	-	-	-	-	-	-	-	-	-	-
Monoxide	PoW-based	Part. Sync.	-	1/2	Comp.	✓	✗**	✓	✗†	✓	$O(n)$
Zilliqa	VRF-based	Sync.	-	0	Comp.	✓	✓	✓	✓	✓	$O(n)$
Ethereum 2.0	PoS-based	N/A	Smart contracts	N/A	Stake	✓	N/A	N/A	N/A	N/A	$O(1)$

* Solved by the underlying membership layer ** Solved by Nakamoto consensus

† With selfish mining [60], the adversary can increase its chance of being elected as leader.

(b) Consensus protocols.

	Protocol	System setting			Correctness				Performance
		Network sync.	Trust	Fault tolerance	Agreement	Validity	Termination	Finality	Comm. compl.
Elastico	PBFT	Part. Sync.	-	1/3	✓	✓	✓	✓	$O(n^2)$
Omniledger	ByzCoinX	Part. Sync.	-	1/3	✓	✓	✗ [61]	✓	$O(n)$
RapidChain	[47]	Sync.	-	1/2	✓	✓	✓	✓	$O(n^2)$
Chainspace	PBFT	Part. Sync.	-	1/3	✓	✓	✓	✓	$O(n^2)$
Monoxide	Nakamoto	Part. Sync.	-	1/2	✓	✓	✓	✗	$O(n)$
Zilliqa	PBFT + CoSi	Part. Sync.	-	1/3	✓	✓	✓	✓	$O(n)$
Ethereum 2.0	Casper FFG	Async.	Smart contracts	1/3	✓	✓	✗ [62], [63]	✓	$O(n)$

sharding protocols. We consider system settings mentioned in §2, and evaluate consensus protocols against the following three properties [24]:

- *Agreement*: no two honest nodes decide differently.
- *Validity*: the value decided must be a value proposed.
- *Termination*: all honest nodes eventually decide.
- *(Optional) Finality* [69]: if a correct node appends a block B before another block B' to its local blockchain, then no correct node appends B' before B to its local blockchain.

We consider a single performance metric, namely the *communication complexity*. Table 3b summarises our evaluation on consensus. Note that fault tolerance capacity is quantified by voting power, of which the definition depends on protocol designs. It shows that all consensus protocols either require quadratic communication complexity or fail to satisfy *termination*, except for Nakamoto consensus used by Monoxide. To summarise, Omniledger's ByzCoinX consensus protocol does not satisfy termination, as it can lose liveness when the leader is Byzantine, as pointed out by Yu et al. [61]. Ethereum 2.0's Casper FFG consensus protocol executes upon a special smart contract, where nodes (aka validators) post their votes to agree on the next block. Casper FFG does not satisfy termination, as it will never terminate when none of the blocks at a certain height reaches the finalisation threshold [62], [63].

7. Cross-shard layer

In this section, we evaluate the cross-shard layer. As mentioned in §3, the cross-shard layer consists of *Concurrency Control* (CC) and *Atomic Commit* (AC). We evaluate and analyse CC and AC separately.

For CC, we find that existing sharded blockchains do not separate CC and AC, and focus less on designing and analysing CC compared to AC. We isolate CC from AC for them and analyse these CC protocols based on well-defined models in database literature [25]–[27]. Our evaluation shows that existing sharded blockchains choose different types of Concurrency Control protocols with different isolation levels. We thus narrow down the design space and analyse the trade-offs of Concurrency Control for sharded blockchains.

For AC, we show that cross-shard transactions should employ a variant of AC called Non-Blocking AC (NB-AC) [28]. We evaluate AC protocols of sharded blockchains in the lens of NB-AC. Based on our evaluation, we analyse the challenges of achieving *agreement* and *termination*, and discuss the performance through two criteria, namely communication complexity and timing of consistency.

7.1. Concurrency Control

Background. Transactional systems such as databases have to satisfy the *isolation* property [25]: concurrent transactions do not affect each other. Concurrency Control (CC) is a family of protocols that provide isolation by specifying how and when modifications made by a transaction become visible to other transactions.

There are four types of CC protocols [70], [71], namely coordinator-based CC, timestamp-based CC (T/O, aka deterministic scheduling) pessimistic concurrency

control (PCC), and optimistic concurrency control (OCC). In coordinator-based CC, there exists a centralised coordinator who receives transactions from users, orders transactions and resolves conflicts between transactions. In timestamp-based CC, transactions are timestamped by a global clock and are processed chronologically according to the timestamps. PCC and OCC are both implemented by using locks, and they make different assumptions on the conflict rate of transactions. PCC assumes most transactions are conflicted, and follows the two-phase locking (2PL) approach: a transaction locks its accessed objects, then modifies objects, and finally releases locks on the objects. OCC assumes few transactions are conflicted, and follows the modify-validate-commit/rollback approach: a transaction modifies objects while saving a copy of original objects, then verifies if other transactions modify these objects, and finally commits modifications if no other transaction does this, otherwise rolls back modifications.

System setting. Concurrency Control (CC) does not rely on any of the system setting aspects discussed in §2. A CC protocol specifies a set of rules that should be followed when appending a transaction to the ledger. Any node can verify whether a transaction satisfies these rules for a ledger, without relying on communication with other nodes or trusted third parties.

Correctness and performance. CC's correctness includes *safety* and *liveness*. *Safety* defines the *isolation* guarantee of transactions. There are various isolation levels [25]. With higher isolation level, concurrent transactions have less impact on each other, but fewer transactions can be executed concurrently. There are two widely accepted isolation levels, namely *serialisability (I-S)* and *snapshot isolation (I-SI)*. Serialisability is the strongest isolation guarantee, and snapshot isolation is a relatively weaker one.

- *Serialisability (I-S)* [26]: for any set of transactions (that might be executed concurrently) and their execution result r, there exists a sequence of them such that its execution result is equivalent to r.
- *Snapshot isolation (I-SI)* [27]: for any transaction tx, 1) all read operations on an object in tx return the same result (e.g., the result of the first read operation), and 2) iff no other concurrent transactions modify objects read by tx, tx will commit, otherwise tx will rollback.

The key difference between them is that snapshot isolation does not prevent the *write skew* anomaly, where two transactions (tx_1, tx_2) simultaneously read the same set of objects a and b, simultaneously make disjoint updates (e.g., tx_1 updates a and tx_2 updates b), and simultaneously commit, without noticing the latest updates made by each other. Serialisability is usually achieved by PCC, while snapshot isolation is usually achieved by OCC.

Liveness is defined the same way as *termination* [72]: transactions will eventually terminate rather than halting halfway. To conclude, we evaluate CC against the following two properties:

- *Safety*: the protocol guarantees a certain isolation level.
- *Liveness*: the execution of any transaction will not halt halfway.

TABLE 4: Evaluation of Concurrency Control.

	Protocol	Correctness		Performance
		Safety	Liveness	No cross-shard comm.
Elastico	Coordinator	I-S†	✓	✗
Omniledger	2PL	I-S	✓*	✓
RapidChain	2PL	I-S	✓*	✓
Chainspace	OCC	I-SI‡	✓	✗
Monoxide	2PL	I-S	✓*	✓
Zilliqa	Coordinator	I-S	✓	✗
Ethereum 2.0	2PL	I-S	✓*	✓

* Users have incentive to finish cross-shard transactions.

† I-S means serialisability. ‡ I-SI means snapshot isolation.

For CC's performance, we evaluate whether nodes in different shards need to communicate with each other.

Evaluation and analysis. Table 4 summarises our evaluation results on CC. Elastico and Zilliqa use coordinated-based CC that achieves serialisability and liveness, and requires cross-shard communication. The final committee receives blocks from all shards, and merge all blocks to a single one where transactions are ordered. Omniledger, RapidChain, Monoxide and Ethereum 2.0 use the two-phase locking (2PL) protocol that achieves serialisability and liveness, and requires no cross-shard communication. Omniledger, RapidChain, Monoxide, and Ethereum 2.0 achieve liveness by using *incentive*: users need to submit their cross-shard transactions in order to receive money. Chainspace's S-BAC protocol implements an OCC protocol that achieves snapshot isolation and liveness. The OCC protocol forbids concurrent write operations, and detecting them requires cross-shard communication.

Design space of CC. According to Table 4, existing sharded blockchains use any of them except for T/O. The reason why T/O is not used is that there is no global clock in permissionless networks. Without a global clock, the adversary can forge transactions with any timestamps, in order to re-order transactions arbitrarily and break the isolation guarantee.

Coordinator-based CC is less complex than PCC and OCC, as the coordinator has a complete view of all transactions. However, coordinator-based CC achieves lower throughput limit than PCC and OCC. In coordinator-based CC, the coordinator has to wait for all shards to produce blocks, then merge all blocks to a single block and publish it. The waiting process reduces the concurrency level, and thus the throughput limit.

PCC and OCC make different assumptions on the distribution of conflicted transactions: PCC assumes more transactions are conflicted compared to OCC's assumption. Existing studies on evaluating and comparing PCC and OCC show a consistent result: with more conflicted transactions, PCC outperforms OCC, and vice versa [73]. Apart from the conflict rate assumption, OCC requires shards to communicate with each other in order to detect conflicted write operations before including transactions, while PCC does not require cross-shard communication. In addition, OCC allows to roll back transactions, and the rolling back mechanism incurs more complexity in protocol design.

7.2. Atomic Commit

System setting. We consider aspects mentioned in §2.

Correctness. Apart from the previously known replay attack [74], we identify three new attacks on the Atomic Commit (AC) for sharded blockchains, allowing us to introduce the required correctness properties. The four attacks are as follows.

1) **Transaction forging attack (Figure 5a)** The adversary creates a fake cross-shard transaction for a shard, then convinces other shards that this fake transaction has been included on that shard. Without cross-shard communication and the knowledge of the shard's ledger, other shards cannot determine whether this cross-shard transaction is valid or not.
2) **Message withholding attack (Figure 5b)** The adversary withholds some messages that should be sent to shards. This can stop cross-shard transactions from being processed.
3) **Replay attack [74] (Figure 5c)** The adversary probes a cross-shard transaction, then replays it to the involved shards. In permissionless networks, nodes cannot distinguish whether their received messages are honest but delayed, or malicious. Such replayed messages can lead to two scenarios. The first scenario (e.g., in Chainspace) is that the victim shard considers the replayed transaction to be malicious so rejects it. The second scenario (e.g., in Omniledger, Rapidchain, and Monoxide) is that shards will have conflicted views on the replayed transaction.
4) **Publish-revert attack (Figure 5d)** In probabilistic consensus protocols such as Nakamoto consensus, a transaction might be accepted first and reverted later. Given a sharded blockchain with probabilistic consensus, it is possible that a cross-shard transaction is valid in some shards but has been reverted in other shards. Consequently, shards have conflicted views on the transaction.

To resist against these four attacks, AC in sharded blockchains should provide the same guarantee as Non-Blocking AC (NB-AC) [28], an AC variant that additionally satisfies termination. NB-AC satisfies the following four properties.

- *Agreement*: for any cross-shard transaction, all involved shards have the same decision on it.
- *Termination*: for any cross-shard transaction, all involved shards eventually decide on it.
- *Abort-validity*: a cross-shard transaction will be aborted iff at least one involved shard votes to abort it.
- *Commit-validity*: a cross-shard transaction will be included iff all involved shards vote to include it.

Agreement, abort-validity and commit-validity jointly provide resistance against transaction forging attacks. Termination provides resistance against message withholding attacks. Agreement and termination jointly provide resistance against replay attacks. Agreement provides resistance against publish-revert attacks.

Performance. We evaluate two performance metrics, namely *communication complexity* and *timing of consistency*. We consider three levels of timing of consistency [75] for sharded blockchains:

- *Strict consistency*: transactions will be seen by all nodes once included;
- *Casual consistency*: transactions will be seen only by relevant nodes once included; and
- *Eventual consistency*: transactions will be seen by relevant nodes but without any timing guarantee.

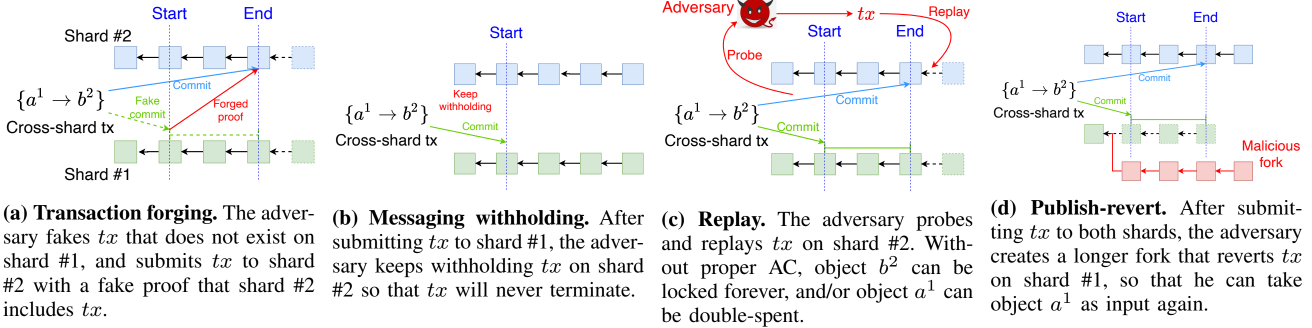
Evaluation and analysis. Table 5 summarises our evaluation results on AC. Omniledger, RapidChain and Monoxide are vulnerable to transaction forging attacks, as the adversary can act as a user and forge transactions to proceed AC on output shards and abort AC on input shards. Omniledger, RapidChain, Chainspace and Monoxide are vulnerable to replay attacks as analysed in the paper [74]. Monoxide is vulnerable to publish-revert attacks, as Monoxide's PoW-based consensus protocol does not provide finality and the adversary may revert cross-shard transactions in input shards. Thus, Omniledger, RapidChain, Chainspace and Monoxide do not achieve agreement. Omniledger, RapidChain, Monoxide and Ethereum 2.0's AC protocols are vulnerable to message withholding attacks, as the adversary can act as a user and withhold cross-shard transactions on output shards. Thus, they do not satisfy *termination*.

Elastico and Chainspace's AC protocols require quadratic communication complexity. Elastico's AC requires nodes in the final committee to execute a PBFT consensus, and Chainspace's S-BAC protocol requires nodes to execute PBFT consensus for multiple times, leading to quadratic communication complexity. Omniledger and Zilliqa's AC protocols require nodes in involved shards to execute the CoSi-based PBFT consensus, leading to linear communication complexity. In RapidChain, Monoxide and Ethereum's AC protocols, nodes do not need to execute extra intra-shard consensus, leading to constant communication complexity.

In Elastico and Zilliqa, after all shards produce their blocks and send them to the final committee, the final committee merges blocks from all shards and publishes the block to all nodes, leading to strict consistency. In Chainspace, each cross-shard transaction invokes an AC among only involved shards, leading to casual consistency. In Monoxide and Chainspace, users split cross-shard transactions and shards process splitted transactions independently without cross-shard communication, leading to eventual consistency.

Challenges of achieving agreement. Our evaluation shows that achieving *agreement* is challenging, due to transaction forging and replay attacks. In fact, when shards maintain different parts of the ledger, achieving *agreement* between shards is proven to be *impossible* without a trusted third party [20]. In a nutshell, when blockchains control disjoint sets of information, solving cross-chain communication, i.e., making blockchains to agree on something, is equivalent to solve fair exchange, which is proven impossible without a trusted third party [76]. The impossibility also applies to cross-shard communication when shards maintain different parts of the ledger.

To workaround the impossibility, sharded blockchains have to either employ a trusted party or enforce nodes to store redundant information of other shards. Elastico and



(a) Transaction forging. The adversary fakes tx that does not exist on shard #1, and submits tx to shard #2 with a fake proof that shard #2 includes tx.

(b) Messaging withholding. After submitting tx to shard #1, the adversary keeps withholding tx on shard #2 so that tx will never terminate.

(c) Replay. The adversary probes and replays tx on shard #2. Without proper AC, object b^2 can be locked forever, and/or object a^1 can be double-spent.

(d) Publish-revert. After submitting tx to both shards, the adversary creates a longer fork that reverts tx on shard #1, so that he can take object a^1 as input again.

Figure 5: Four possible attacks on Atomic Commit (AC). We use cross-shard transaction $tx = \{a^1 \to b^2\}$ as an example.

TABLE 5: Evaluation of Atomic Commit. Symbol "-" means the protocol has no name.

		System setting			Correctness				Performance	
	Protocol	**Network sync.**	**Trust**	**Fault tolerance**	**Agreement**	**Termination**	**Abort-Validity**	**Commit-Validity**	**Comm. compl.**	**Timing**
Elastico	-	Part. Sync.	-	⅓	✓	✓	✓	✓	$O(n^2)$	Strict
Omniledger	Atomix	Async.	-	⅓	✗^{fr}	✗^{w}	✓	✓	$O(n)$	Eventual
RapidChain	-	Async.	-	-	✗^{fr}	✗^{w}	✓	✓	$O(1)$	Eventual
Chainspace	S-BAC	Part. Sync.	-	⅓	✗^{r}	✓	✓	✓	$O(n^2)$	Casual
Monoxide	-	Async.	-	-	✗^{frp}	✗^{w}	✓	✓	$O(1)$	Eventual
Zilliqa	-	Part. Sync.	-	⅓	✓	✓	✓	✓	$O(n)$	Strict
Ethereum 2.0	-	Async.	-	-	✓	✗^{w}	✓	✓	$O(1)$	Eventual

^{f} vulnerable to transaction forging attacks. ^{w} vulnerable to message withholding attacks.
^{r} vulnerable to replay attacks. ^{p} vulnerable to publish-revert attacks.

Zilliqa employ a final committee that stores the entire ledger and coordinates all cross-shard transactions. This requires the final committee to be trustworthy, and introduces non-negligible storage and communication overhead. Ethereum 2.0 requires nodes to store block headers and work as lightweight clients of all shards, so that every node can verify all cross-shard transactions. This can be seen as a trade-off between storage and security guarantee.

Challenges of achieving termination. When users are responsible for submitting transactions on output shards, the adversary can play as a user and keep withholding transactions, breaking AC's *termination*. Nevertheless, with proper CC, such withholding can be discouraged. For example, in Omniledger, RapidChain and Monoxide, if the adversary withholds transactions on output shards, then its money is locked (i.e., cannot be spent) forever, and the system still works correctly.

Trade-off between timing of consistency and performance. Elastico and Zilliqa achieve *strict consistency* by enforcing all shards to synchronise with each other. Chainspace achieves *casual consistency* by enforcing shards involved in a cross-shard transaction to synchronise with each other. Monoxide achieves *eventual consistency* by allowing shards to process cross-shard transactions independently. To achieve better timing of consistency, the sharded blockchain has to either enforce more shards to execute AC, or enforce shards to execute AC more frequently. Both approaches incur significant communication overhead and downgrade performance. This implies a trade-off between timing of consistency and performance: achieving better timing of consistency usually sacrifices performance.

8. System-level analysis

In this section, we provide the system-level evaluation on sharded blockchains based on individual layer evaluation in §5-7. The evaluation results in Table 7 show that existing sharded blockchains either assume strong system settings or fail to preserve all correctness properties. We attribute the issue to the design choice that different protocol layers assume different system settings.

8.1. Evaluation

System setting. If a system consists of multiple protocols, then it remains secure only when all assumptions made by its protocols hold, otherwise some protocols cannot achieve all correctness properties [30], compromising the entire system. Therefore, the system remains secure under the strongest assumptions made by its protocol layers. Given this observation, we derive the system settings of the sharded blockchains from the individual layer evaluation. Table 6 summarises system settings of sharded blockchains analysed in §5-7. It shows that except for Monoxide and Ethereum 2.0, all evaluated sharded blockchains assume incoherent system settings on different layers. In addition, Elastico and Zilliqa cannot tolerate a single Byzantine node, otherwise the system will lose liveness. Moreover, Omniledger, Chainspace and Ethereum 2.0 rely on various trusted parties for some protocol layers.

Correctness properties. We consider correctness properties defined in §2, namely *persistence*, *liveness*, and *validity*. All sharded blockchains satisfy persistence. Omniledger and Ethereum 2.0 do not satisfy *liveness* as their

TABLE 6: System settings of sharded blockchains across layers.

		Network sync.	Trust	Fault tolerance
Elastico	Shard allocation	Sync.	-	1/3
	Leader election	Sync	-	0
	Consensus	Part. Sync.	-	1/3
	Atomic Commit	Part. Sync.	-	1/3
	Overall	**Sync.**	**-**	**0**
Omniledger	Shard allocation	Async.	Iden. auth.	1/3
	Leader election	Sync.	Protocol leader	N/A
	Consensus	Part. Sync.	-	1/3
	Atomic Commit	Async.	-	1/3
	Overall	**Sync.**	**Iden. auth. Protocol leader**	**1/3**
RapidChain	Shard allocation	Sync.	-	0
	Leader election	N/A	N/A	N/A
	Consensus	Sync.	-	1/2
	Atomic Commit	Async.	-	-
	Overall	**Sync.**	**-**	**0**
Chainspace	Shard allocation	Async.	Smart contracts	1
	Leader election	-	-	-
	Consensus	Part. Sync.	-	1/3
	Atomic Commit	Part. Sync.	-	1/3
	Overall	**Part. Sync.**	**Smart contracts**	**1/3**
Monoxide	Shard allocation	Async.	-	1
	Leader election	Async.	-	1/2
	Consensus	Async	-	1/2
	Atomic Commit	Async.	-	-
	Overall	**Async.**	**-**	**1/2**
Zilliqa	Shard allocation	Sync.	-	1
	Leader election	Sync.	-	0
	Consensus	Part. Sync.	-	1/3
	Atomic Commit	Part. Sync.	-	1/3
	Overall	**Sync.**	**-**	**0**
Ethereum 2.0	Shard allocation	Async.	-	1
	Leader election	N/A	Smart contracts	N/A
	Consensus	Async.	Smart contracts	1/3
	Atomic Commit	Async.	-	-
	Overall	**Async.**	**Smart contracts**	**1/3**

TABLE 7: System-level evaluation.

	System setting			Correctness		
	Network sync.	Trust	Fault tolerance	Persistance	Liveness	Validity
Elastico	Sync.	-	0	✓	✓	✓
Omniledger	Sync.	iden. auth. + leader	1/3	✓	✗	✗
RapidChain	Sync.	-	0†	✓	✓	✗
Chainspace	Part. Sync.	Smart contracts	1/3	✓	✓	✗
Monoxide	Async.	-	1/2	✓	✓	✗
Zilliqa	Sync.	-	0	✓	✓	✓
Ethereum 2.0	Async.	Smart contracts	1/3	✓	✗	✓

† RapidChain's shard allocation protocol cannot tolerate any fault, as analysed in [19].

used consensus protocols do not satisfy termination, as analysed in §6.2. Omniledger, RapidChain, Chainspace and Monoxide do not satisfy *validity* as their Atomic Commit protocols do not satisfy agreement, as analysed in §7.2.

8.2. Coherence of system settings

Based on the evaluation, we observe an important design consideration, namely the coherence of system settings across layers. In particular, we attribute the weak security guarantee of sharded blockchains to their design choice that different protocol layers make different system settings, without corresponding architectural guarantees. Consequently, unless a hybrid architecture were used, assuming differentiated system settings across system layers is a serious vulnerability and can introduce subtle but severe failure syndromes [30]. In the context of blockchain sharding, to remain correct on all layers, the sharded blockchain can only work in the environment that satisfies the strongest system setting made by protocol layers. Otherwise, some protocol layers cannot be fully correct, and the entire sharded blockchain can be compromised.

If the sharded blockchains are deployed in the strongest system settings assumed in Table 7, then by replacing protocols relying on weaker system settings with those relying on the strongest system setting, the sharded blockchain can achieve better performance without compromising security. For Elastico, Omniledger and Zilliqa, by replacing the partially synchronous PBFT consensus protocol with a synchronous consensus protocol, the performance can be significantly improved. For Omniledger, by using the trusted identity authority rather than a trusted protocol leader to nominate consensus leaders, the need of the trusted protocol leader can be removed. By using the trusted identity authority to coordinate cross-shard transactions rather than using the Atomix Atomic Commit protocol, the $O(n)$ communication overhead of Atomix can be reduced, and the timing of consistency can be improved.

If the sharded blockchains are deployed in weaker system settings than those in Table 7, then they cannot achieve some correctness properties. For example, the network may be partially synchronous or even asynchronous. If the network is partially synchronous, then some sharded blockchains lose some correctness properties: Elastico, Omniledger, and Zilliqa's leader election protocols cannot achieve termination, breaching the system's liveness; and RapidChain's consensus protocol cannot achieve agreement, breaching the system's persistence. If the network is asynchronous, then Chainspace also loses liveness, as its consensus protocol and Atomic Commit protocol cannot achieve termination.

9. Conclusion

In this paper, we provided a comprehensive evaluation for state-of-the-art blockchain sharding protocols. Our evaluation reveals a considerable number of new attacks, design trade-offs and open challenges in sharded blockchains. Based on the analysis of individual layers, we also observed and analysed the importance of assuming coherent system settings across protocol layers. We hope that the blockchain community will benefit from our findings and avoid our identified pitfalls in designing secure and scalable blockchain sharding protocols.

References

[1] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*, Springer, 2017, pp. 357–388.

[3] P. Daian, R. Pass, and E. Shi, "Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake," in *International Conference on Financial Cryptography and Data Security*, Springer, 2019, pp. 23–41.

[4] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable off-chain instant payments," 2016.

[5] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43.

[7] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 International Conference on Management of Data*, ACM, 2019, pp. 123–140.

[8] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "Resilientdb: Global scale resilient blockchain fabric," *Proceedings of the VLDB Endowment*, vol. 13, no. 6,

[9] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Sharper: Sharding permissioned blockchains over network clusters," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 76–88.

[10] J. Hellings and M. Sadoghi, "Byshard: Sharding in a byzantine environment," *Proc. VLDB Endow*, 2021.

[11] 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*, ACM, 2016, pp. 17–30.

[12] 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)*, IEEE, 2018, pp. 583–598.

[13] 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*, ACM, 2018, pp. 931–948.

[14] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[15] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 95–112.

[16] Z. Team *et al.*, "The Zilliqa technical whitepaper," *Retrieved September*, vol. 16, p. 2019, 2017.

[17] *Ethereum/wiki*. [Online]. Available: https://github.com/ethereum/wiki.

[18] G. Wang, Z. J. Shi, M. Nixon, and S. Han, "Sok: Sharding on blockchain," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*, 2019.

[19] R. Han, J. Yu, and R. Zhang, "Analysing and Improving Shard Allocation Protocols for Sharded Blockchains," 2020, https://eprint.iacr.org/2020/943.

[20] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt, "Sok: Communication across distributed ledgers," in *International Conference on Financial Cryptography and Data Security*, 2021.

[21] G. Avarikioti, E. Kokoris-Kogias, and R. Wattenhofer, "Divide and scale: Formalization of distributed ledger sharding protocols," *arXiv preprint arXiv:1910.10434*, 2019.

[22] D. Boneh, S. Eskandarian, L. Hanzlik, and N. Greco, "Single secret leader election," in *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*.

[23] P. A. Bernstein, P. A. Bernstein, and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.

[24] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[25] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987, vol. 370.

[26] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, 1979.

[27] M. J. Franklin, *Concurrency control and recovery.* 1997.

[28] R. Guerraoui, "Non-blocking atomic commit in asynchronous distributed systems with failure detectors," *Distributed Computing*, vol. 15, no. 1, pp. 17–25, 2002.

[29] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7, 2000.

[30] P. Sousa, N. F. Neves, and P. Verissimo, "How resilient are distributed f fault/intrusion-tolerant systems?" In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, IEEE, 2005, pp. 98–107.

[31] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[32] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2015, pp. 281–310.

[33] "The zilliqa design story piece by piece: Part 1 (network sharding)," https://blog.zilliqa.com/https-blog-zilliqa-com-the-zilliqa-design-story-piece-by-piece-part1-d9cb32ea1e65.

[34] "Ethereum sharding: Overview and finality," https://medium.com/@icebearhww/ethereum-sharding-and-finality-65248951f649.

[35] V. Buterin, *Serenity design rationale*. [Online]. Available: https://notes.ethereum.org/@vbuterin/rkhCgQteN?type=view.

[36] (2020). On sharding blockchains faqs. https://eth.wiki/sharding/Sharding-FAQs.

[37] K. Antoniadis, A. Desjardins, V. Gramoli, R. Guerraoui, and M. I. Zablotchi, "Leaderless consensus," Tech. Rep., 2021.

[38] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports, "Building consistent transactions with inconsistent replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, p. 12, 2018.

[39] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM computing surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

[40] *Ethereum/eth2.0-specs*. [Online]. Available: https://github.com/ethereum/eth2.0-specs.

[41] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[42] J. Aspnes, C. Jackson, and A. Krishnamurthy, "Exposing computationally-challenged byzantine impostors," Technical Report YALEU/DCS/TR-1332, Yale University Department of Computer, Tech. Rep., 2005.

[43] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[44] 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)*, 2016, pp. 279–296.

[45] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," in *2017 IEEE Symposium on Security and Privacy (SP)*, Ieee, 2017, pp. 444–460.

[46] S. Sen and M. J. Freedman, "Commensal cuckoo: Secure group partitioning for large-scale services," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 1, pp. 33–39, 2012.

[47] L. Ren, K. Nayak, I. Abraham, and S. Devadas, "Brief announcement: Practical synchronous byzantine consensus," in *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, 2017.

[48] J. A. Garay, A. Kiayias, and N. Leonardos, "Full analysis of nakamoto consensus in bounded-delay networks.," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 277, 2020.

[49] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[50] *Randao: A dao working as rng of ethereum*. [Online]. Available: https://github.com/randao/randao.

[51] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[52] D. Karakostas and A. Kiayias, "Securing proof-of-work ledgers via checkpointing," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, 2021, pp. 1–5.

[53] S. Sankagiri, X. Wang, S. Kannan, and P. Viswanath, "The checkpointed longest chain: User-dependent adaptivity and finality," in *International Conference on Financial Cryptography and Data Security*, 2021.

[54] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.

[55] M. Fitzi, P. Gazi, A. Kiayias, and A. Russell, "Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition," *IACR Cryptology ePrint Archive*, vol. 2018, p. 1119, 2018.

[56] H. Yu, I. Nikolic, R. Hou, and P. Saxena, “OHIE: Blockchain Scaling Made Simple,” in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, 2018.
[57] J. Niu, “Eunomia: A permissionless parallel chain protocol based on logical clock,” *arXiv preprint arXiv:1908.07567*, 2019.
[58] S. Forestier and D. Vodenicarevic, “Blockclique: Scaling blockchains through transaction sharding in a multithreaded block graph,” *arXiv preprint arXiv:1803.09029*, 2018.
[59] D. Hensgen, R. Finkel, and U. Manber, “Two algorithms for barrier synchronization,” *International Journal of Parallel Programming*, vol. 17, no. 1, pp. 1–17, 1988.
[60] I. Eyal and E. G. Sirer, “Majority is not enough: Bitcoin mining is vulnerable,” in *International conference on financial cryptography and data security*, Springer, 2014, pp. 436–454.
[61] J. Yu, D. Kozhaya, J. Decouchant, and P. Verissimo, “Repucoin: Your reputation is your power,” *IEEE Transactions on Computers*, 2019.
[62] Y. Wang, “Byzantine fault tolerance in partial synchronous networks,” 2020.
[63] *Formal analysis of the cbc casper consensus algorithm with tla+*. [Online]. Available: https://blog.trailofbits.com/2019/10/25/formal-analysis-of-the-cbc-casper-consensus-algorithm-with-tla.
[64] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, “Sok: Consensus in the age of blockchains,” in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*, 2019.
[65] J. A. Garay and A. Kiayias, “SoK: A Consensus Taxonomy in the Blockchain Era.,” in *Topics in Cryptology - CT-RSA 2020 - The Cryptographers’ Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*.
[66] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, “A survey of distributed consensus protocols for blockchain networks,” *IEEE Commun. Surv. Tutorials*, 2020.
[67] C. Natoli, J. Yu, V. Gramoli, and P. Esteves-Verissimo, “Deconstructing blockchains: A comprehensive survey on consensus, membership and structure,” *arXiv preprint arXiv:1908.08316*, 2019.
[68] C. Gómez-Calzado, A. Lafuente, M. Larrea, and M. Raynal, “Fault-tolerant leader election in mobile dynamic distributed systems,” in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, IEEE, 2013, pp. 78–87.
[69] M. Vukolić, “The quest for scalable blockchain fabric: Proof-of-work vs. bft replication,” in *International workshop on open problems in network security*, Springer, 2015, pp. 112–125.
[70] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, “Staring into the abyss: An evaluation of concurrency control with one thousand cores,” 2014.
[71] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, “An evaluation of distributed concurrency control,” *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 553–564, 2017.
[72] F. Pedone and R. Guerraoui, “On transaction liveness in replicated databases,” in *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*, IEEE, 1997, pp. 104–109.
[73] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, “Extracting more concurrency from distributed transactions,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 479–494.
[74] A. Sonnino, S. Bano, M. Al-Bassam, and G. Danezis, “Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers,” in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2020, pp. 294–308.
[75] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.
[76] H. Pagnia and F. C. Gärtner, “On the impossibility of fair exchange without a trusted third party,” Technical Report TUD-BS-1999-02, Darmstadt University of Technology, Tech. Rep., 1999.