OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding

***Abstract*—Designing a secure permissionless distributed ledger (blockchain) that performs on par with centralized payment processors, such as Visa, is a challenging task. Most existing distributed ledgers are unable to scale-out, *i.e.*, to grow their total processing capacity with the number of validators; and those that do, compromise security or decentralization. We present OmniLedger, a novel scale-out distributed ledger that preserves long-term security under permissionless operation. It ensures security and correctness by using a bias-resistant public-randomness protocol for choosing large, statistically representative shards that process transactions, and by introducing an efficient cross-shard commit protocol that atomically handles transactions affecting multiple shards. OmniLedger also optimizes performance via parallel intra-shard transaction processing, ledger pruning via collectively-signed state blocks, and low-latency "trust-but-verify" validation for low-value transactions. An evaluation of our experimental prototype shows that OmniLedger's throughput scales linearly in the number of active validators, supporting Visa-level workloads and beyond, while confirming typical transactions in under two seconds.**

I. Introduction

The scalability of distributed ledgers (DLs), in both total transaction volume and the number of independent participants involved in processing them, is a major challenge to their mainstream adoption, especially when weighted against security and decentralization challenges. Many approaches exhibit different security and performance trade-offs [10], [11], [21], [32], [40]. Replacing the Nakamoto consensus [36] with PBFT [13], for example, can increase throughput while decreasing transaction commit latency [1], [32]. These approaches still require all *validators* or consensus group members to redundantly validate and process all transactions, hence the system's total transaction processing capacity does not increase with added participants, and, in fact, gradually decreases due to increased coordination overheads.

The proven and obvious approach to building "*scale-out*" databases, whose capacity scales horizontally with the number of participants, is by *sharding* [14], or partitioning the state into multiple shards that are handled in parallel by different subsets of participating validators. Sharding could benefit DLs [15] by reducing the transaction processing load on each validator and by increasing the system's total processing capacity proportionally with the number of participants. Existing proposals for sharded DLs, however, forfeit permissionless decentralization [16], introduce new security assumptions, and/or trade performance for security [34], as illustrated in Figure 1 and explored in detail in Sections II and IX.

We introduce OmniLedger, the first DL architecture that provides "scale-out" transaction processing capacity competitive with centralized payment-processing systems, such as Visa, without compromising security or support for permissionless decentralization. To achieve this goal, OmniLedger

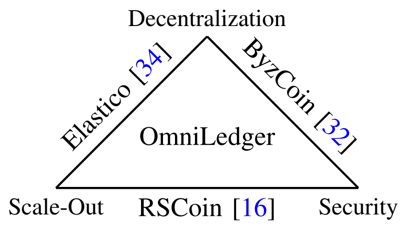

Fig. 1: Trade-offs in current DL systems.

faces three key correctness and security challenges. First, OmniLedger must choose statistically representative groups of validators periodically via permissionless Sybil-attack-resistant foundations such as proof-of-work [36], [38], [32] or proof-of-stake [31], [25]. Second, OmniLedger must ensure a negligible probability that any shard is compromised across the (long-term) system lifetime via periodically (re-)forming shards (subsets of validators to record state and process transactions), that are both sufficiently large and bias-resistant. Third, OmniLedger must correctly and atomically handle *cross-shard transactions*, or transactions that affect the ledger state held by two or more distinct shards.

To choose representative validators via proof-of-work, OmniLedger builds on ByzCoin [32] and Hybrid Consensus [38], using a sliding window of recent proof-of-work block miners as its validator set. To support the more power-efficient alternative of apportioning consensus group membership based on directly invested stake rather than work, OmniLedger builds on Ouroboros [31] and Algorand [25], running a public randomness or cryptographic sortition protocol within a prior validator group to pick a subsequent validator group from the current stakeholder distribution defined in the ledger. To ensure that this sampling of representative validators is both scalable and strongly bias-resistant, OmniLedger uses RandHound [44], a protocol that serves this purpose under standard t-of-n threshold assumptions.

Appropriate use of RandHound provides the basis by which OmniLedger addresses the second key security challenge of securely assigning validators to shards, and of periodically rotating these assignments as the set of validators evolves. OmniLedger chooses shards large enough, based on the analysis in Section VI, to ensure a negligible probability that any shard is ever compromised, even across years of operation.

Finally, to ensure that transactions either commit or abort atomically even when they affect state distributed across multiple shards (*e.g.*, several cryptocurrency accounts), OmniLedger introduces Atomix, a two-phase client-driven "lock/unlock" protocol that ensures that clients can either fully commit a transaction across shards, or obtain "rejection proofs" to abort and unlock state affected by partially completed transactions.

Besides addressing the above key security challenges, OmniLedger also introduces several performance and scalability refinements we found to be instrumental in achieving its usability goals. OmniLedger's consensus protocol, *ByzCoinX*, enhances the PBFT-based consensus in ByzCoin [32] to preserve performance under Byzantine denial-of-service (DoS) attacks, by adopting a more robust group communication pattern. To help new or long-offline miners catch up to the current ledger state without having to download the entire history, OmniLedger adapts classic distributed checkpointing principles [20] to produce consistent, *state blocks* periodically.

Finally, to minimize transaction latency in common cases such as low-value payments, OmniLedger supports optional *trust-but-verify* validation in which a first small tier of validators processes the transactions quickly and then hands them over to a second larger, hence slower, tier that reverifies the correctness of the first tier transactions and ensures long-term security. This two-level approach ensures that any misbehavior within the first tier is detected within minutes, and can be strongly disincentivized through recourse such as loss of deposits. Clients can wait for both tiers to process high-value transactions for maximum security or just wait for the first tier to process low-value transactions.

To evaluate OmniLedger, we implemented a prototype in Go on commodity servers (12-core VMs on Deterlab). Our experimental results show that OmniLedger scales linearly in the number of validators, yielding a throughput of 6,000 transactions per second with a 10-second consensus latency (for 1800 widely-distributed hosts, of which 12.5% are malicious). Furthermore, deploying OmniLedger with two-level, trust-but-verify validation provides a throughput of 2,250 tps with a four-second first-tier latency under a 25% adversary. Finally, a Bitcoin validator with a month-long stale view of the state incurs 40% of the bandwidth, due to state blocks.

In summary, this paper makes the following contributions:

- We introduce the first DL architecture that provides horizontal scaling without compromising either long-term security or permissionless decentralization.
- We introduce Atomix, a Atomic Commit protocol, to commit transactions atomically across shards.
- We introduce ByzCoinX, a BFT consensus protocol that increases performance and robustness to DoS attacks.
- We introduce state blocks, that are deployed along OmniLedger to minimize storage and update overhead.
- We introduce two-tier trust-but-verify processing to minimize the latency of low-value transactions.

II. Background

A. Scalable Byzantine Consensus in ByzCoin

OmniLedger builds on the Byzantine consensus scheme in ByzCoin [32], because it scales efficiently to thousands of consensus group members. To make a traditional consensus algorithm such as PBFT [13] more scalable, ByzCoin uses collective signing or CoSi [45], a scalable cryptographic primitive that implements multisignatures [42]. ByzCoin distributes blocks by using multicast trees for performance, but falls back to a less-scalable star topology for fault tolerance. Although ByzCoin's consensus is scalable, its total processing capacity does not increase with participation *i.e.*, it does not scale-out.

B. Transaction Processing and the UTXO model

Distributed ledgers derive current system state from a *blockchain*, or a sequence of totally ordered blocks that contain transactions. OmniLedger adopts the *unspent transaction output* (UTXO) model to represent ledger state, due to its simplicity and parallelizability. In this model, the outputs of a transaction create new UTXOs (and assign them credits), and inputs completely "spend" existing UTXOs. During bootstrapping, new (full) nodes crawl the entire distributed ledger and build a database of valid UTXOs needed to subsequently decide whether a new block can be accepted. The UTXO model was introduced by Bitcoin [36] but has been widely adopted by other distributed ledger systems.

C. Secure Distributed Randomness Generation

RandHound [44] is a scalable, secure multi-party computation (MPC) protocol that provides unbiasable, decentralized randomness in a Byzantine setting. RandHound assumes the existence of an externally accountable client that wants to obtain provable randomness from a large group of semi-trustworthy servers. To produce randomness, RandHound splits the group of servers into smaller ones and creates a publicly verifiable commit-then-reveal protocol [43] that employs the pigeonhole principle to prove that the final random number includes the contribution of at least one honest participant, thus perfectly randomizing RandHound's output.

Cryptographic sortition [25] is used to select a subset of validators, according to some per-validator weight function. To enable validators to prove that they belong to the selected subset, they need a public/private key pair, (pk_i, sk_i). Sortition is implemented using a verifiable random function (VRF) [35] that takes an input x and returns a random hash (ℓ-bit long string) and a proof π based on sk_i. The proof π enables anyone knowing pk_i to check that the hash corresponds to x.

D. Sybil-Resistant Identities

Unlike permissioned blockchains [16], where the validators are known, permissionless blockchains need to deal with the potential of Sybil attacks [19] to remain secure. Bitcoin [36] suggested the use of Proof-of-Work (PoW), where validators (aka miners) create a valid block by performing an expensive computation (iterating through a nonce and trying to brute-force a hash of a block's header such that it has a certain number of leading zeros). Bitcoin-NG [21] uses this PoW technique to enable a Sybil-resistant generation of identities. There are certain issues associated with PoW, such as the waste of electricity [17] and the fact that it causes recentralization [29] to mining pools. Other approaches for establishing Sybil-resistant identities such as Proof-of-Stake (PoS) [31], [25], Proof-of-Burn (PoB) [46] or Proof-of-Personhood [8] overcome PoW's problems and are compatible with ByzCoins identity (key-block) blockchain, and in turn with OmniLedger

E. Prior Sharded Ledgers: Elastico

OmniLedger builds closely on Elastico [34], that previously explored sharding in permissionless ledgers. In every round, Elastico uses the least-significant bits of the PoW hash to distribute miners to different shards. After this setup, every shard runs PBFT [13] to reach consensus, and a leader shard verifies all the signatures and creates a global block.

OmniLedger addresses several challenges that Elastico leaves unsolved. First, Elastico's relatively small shards (*e.g.*, 100 validators per shard in experiments) yield a high failure-probability of 2.76%[1] per shard per block under a 25% adversary, which cannot safely be relaxed in a PoW system [23]. For 16 shards, the failure probability is 97% over only 6 epochs. Second, Elastico's shard selection is not strongly bias-resistant, as miners can selectively discard PoWs to bias results [7]. Third, Elastico does not ensure transaction atomicity across shards, leaving funds in one shard locked forever if another shard rejects the transaction. Fourth, the validators constantly switch shards, forcing themselves to store the global state, which can hinder performance but provides stronger guarantees against adaptive adversaries. Finally, the latency of transaction commitment is comparable to Bitcoin (≈ 10 min.), which is far from OmniLedger's usability goals.

III. System Overview

This section presents the system, network and threat models, the design goals, and a roadmap towards OmniLedger that begins with a strawman design.

A. System Model

We assume that there are n *validators* who process transactions and ensure the consistency of the system's state. Each validator i has a public / private key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$, and we often identify i by pk_i. Validators are evenly distributed across m *shards*. We assume that the configuration parameters of a shard j are summarized in a *shard-policy file*. We denote by an *epoch* e the fixed time (*e.g.*, a day) between global reconfiguration events where a new assignment of validators to shards is computed. The time during an epoch is counted in *rounds* r that do not have to be consistent between different shards. During each round, each shard processes transactions collected from clients. We assume that validators can establish identities through any Sybil-attack-resistant mechanism and commit them to the identity blockchain; to participate in epoch e validators have to register in epoch $e-1$. These identities are added into an identity blockchain as described in Section II-D.

B. Network Model

For the underlying network, we make the same assumption as prior work [31], [34], [36]. Specifically, we assume that (a) the network graph of honest validators is well connected and that (b) the communication channels between honest validators are synchronous, *i.e.*, that if an honest validator broadcasts a message, then all honest validators receive the message within a known maximum delay Δ [39]. However, as Δ is in the scale of minutes, we cannot use it within epochs as we target latencies of seconds. Thus, all protocols inside one epoch use the partially synchronous model [13] with optimistic, exponentially increasing time-outs, whereas Δ is used for slow operations such as identity creation and shard assignment.

C. Threat Model

We denote the number of Byzantine validators by f and assume, that $n = 4f$, *i.e.*, at most 25% [2] of the validators can be malicious at any given moment, which is similar to prior DL's [21], [32], [34]. These malicious nodes can behave arbitrarily, *e.g.*, they might refuse to participate or collude to attack the system. The remaining validators are honest and faithfully follow the protocol. We further assume that the adversary is *mildly adaptive* [31], [34] on the order of epochs, *i.e.*, he can try to corrupt validators, but it takes some time for such corruption attempts to actually take effect.

We further assume that the adversary is computationally bounded, that cryptographic primitives are secure, and that the computational Diffie-Hellman problem is hard.

D. System Goals

OmniLedger has the following primary goals with respect to decentralization, security, and scalability.

1) **Full decentralization.** OmniLedger does not have any single points of failure (such as trusted third parties).
2) **Shard robustness.** Each shard correctly and continuously processes transactions assigned to it.
3) **Secure transactions.** Transactions are committed atomically or eventually aborted, both within and across shards.
4) **Scale-out.** The expected throughput of OmniLedger increases linearly in the number of participating validators.
5) **Low storage overhead.** Validators do not need to store the full transaction history but only a periodically computed reference point that summarizes a shard's state.
6) **Low latency.** OmniLedger provides low latency for transaction confirmations.

E. Design Roadmap

This section introduces SLedger, a strawman DL system that we use to outline OmniLedger's design. Below we describe one epoch of SLedger and show how it transitions from epoch $e-1$ to epoch e.

We start with the secure validator-assignment to shards. Permitting the validators to choose the shards they want to validate is insecure, as the adversary could focus all his validators in one shard. As a result, we need a source of randomness to ensure that the validators of one shard will be a sample of the overall system and w.h.p. will have the same fraction of malicious nodes. SLedger operates a trusted randomness beacon that broadcasts a random value rnd_e to all participants in each epoch e. Validators, who want to participate in SLedger starting from epoch e, have to first register to a global identity blockchain. They create their identities through a Sybil-attack-resistant mechanism in epoch $e-1$ and broadcast them, together with the respective proofs, on the gossip network at most Δ before epoch $e-1$ ends.

Epoch e begins with a leader, elected using randomness rnd_{e-1}, who requests from the already registered and active

[1]Cumulative binomial distribution ($P = 0.25$, $N = 100$, $X \geq 34$)

[2]The system can handle up to $33\% - \epsilon$ with degraded performance

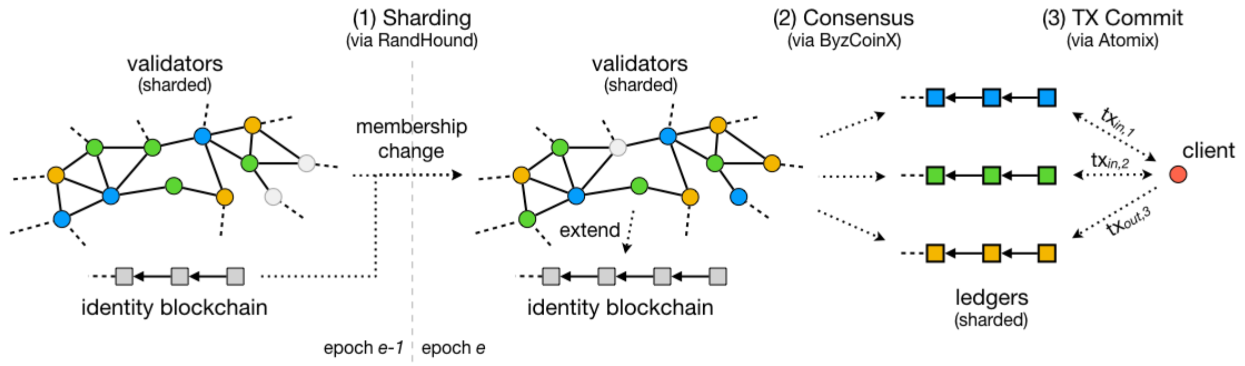

Fig. 2: OmniLedger architecture overview: At the beginning of an epoch e, all validators (shard membership is visualized through the different colors) (1) run RandHound to re-assign randomly a certain threshold of validators to new shards and assign new validators who registered to the identity blockchain in epoch $e-1$. Validators ensure (2) consistency of the shards' ledgers via ByzCoinX while clients ensure (3) consistency of their cross-shard transactions via Atomix (here the client spends inputs from shards 1 and 2 and outputs to shard 3).

validators a (BFT) signature on a block with all identities that have been provably established so far. If at least $\frac{2}{3}$ of these validators endorse the block, it becomes valid, and the leader appends it to the identity blockchain. Afterwards, all registered validators take rnd_e to determine their assignment to one of the SLedger's shards and to bootstrap their internal states from the shards' distributed ledgers. Then, they are ready to start processing transactions using ByzCoin. The random shard-assignment ensures that the ratio between malicious and honest validators in any given shard closely matches the ratio across all validators with high probability.

SLedger already provides a similar functionality to OmniLedger, but it has several significant security restrictions. First, the randomness beacon is a trusted third party. Second, the system stops processing transactions during the global reconfiguration at the beginning of each epoch until enough validators have bootstrapped their internal states and third, there is no support for cross-shard transactions. SLedger's design also falls short in performance. First, due to ByzCoin's failure handling mechanism, its performance deteriorates when validators fail. Second, validators face high storage and bootstrapping overheads. Finally, SLedger cannot provide real-time confirmation latencies and high throughput.

To address the security challenges, we introduce OmniLedger's security design in Section IV:

1) In Section IV-A, we remove the trusted randomness beacon and show how validators can autonomously perform a secure sharding by using a combination of RandHound and VRF-based leader election via cryptographic sortition.
2) In Section IV-B, we show how to securely handle the validator assignment to shards between epochs while maintaining the ability to continuously process transactions.
3) In Section IV-C, we present *Atomix*, a novel two-step atomic commit protocol for atomically processing cross-shard transactions in a Byzantine setting.

To deal with the performance challenges, we introduce OmniLedger's performance and usability design in Section V:

4) In Section V-A, we introduce *ByzCoinX*, a variant of ByzCoin, that utilizes more robust communication patterns to efficiently process transactions within shards, even if some of the validators fail, and that resolves dependencies on the transaction level to achieve better block parallelization.
5) In Section V-C, we introduce *state blocks* that summarize the shards' states in an epoch and that enable ledger pruning to reduce storage and bootstrapping costs for validators.
6) In Section V-D, we show how to enable optimistic real-time transaction confirmations without sacrificing security or throughput by utilizing an intra-shard architecture with *trust-but-verify transaction validation*.

A high-level overview of the (security) architecture of OmniLedger is illustrated in Figure 2.

IV. OmniLedger: Security Design

A. Sharding via Bias-Resistant Distributed Randomness

To generate a seed for sharding securely without requiring a trusted randomness beacon [16] or binding the protocol to PoW [34], we rely on a distributed randomness generation protocol that is collectively executed by the validators.

We require that the distributed-randomness generation protocol provide unbiasability, unpredictability, third-party verifiability, and scalability. Multiple proposals exist [7], [28], [44]. The first approach relies on Bitcoin, whereas the other two share many parts of the design; we focus on RandHound [44] due to better documentation and open-source implementation.

Because RandHound relies on a leader to orchestrate the protocol run, we need an appropriate mechanism to select one of the validators for this role. If we use a deterministic approach to perform leader election, then an adversary might be able to enforce up to f out of n failures in the worst case by refusing to run the protocol, resulting in up to $\frac{1}{4}n$ failures given our threat model. Hence, the selection mechanism must be unpredictable and unbiasable, which leads to a chicken-and-egg problem as we use RandHound to generate randomness with these properties in the first place. To overcome this predicament, we combine RandHound with a VRF-based leader election algorithm [44], [25].

At the beginning of an epoch e, each validator i computes a ticket $\mathsf{ticket}_{i,e,v} = \mathsf{VRF}_{\mathsf{sk}_i}(\text{“leader”} \parallel \mathsf{config}_e \parallel v)$ where

config_e is the configuration containing all properly registered validators of epoch e (as stored in the identity blockchain) and v is a view counter. Validators then gossip these tickets with each other for a time Δ, after which they lock in the lowest-value valid ticket they have seen thus far and accept the corresponding node as the leader of the RandHound protocol run. If the elected node fails to start RandHound within another Δ, validators consider the current run as failed and ignore this validator for the rest of the epoch, even if he returns later on. In this case, the validators increase the view number to $v+1$ and re-run the lottery. Once the validators have successfully completed a run of RandHound and the leader has broadcast rnd_e together with its correctness proof, each of the n properly registered validators can first verify and then use rnd_e to compute a permutation π_e of $1,\dots,n$ and subdivide the result into m approximately equally-sized buckets, thereby determining its assignment of nodes to shards.

Security Arguments: we make the following observations to informally argue the security of the above approach. Each participant can produce only a single valid ticket per view v in a given epoch e, because the VRF-based leader election starts only after the valid identities have been fixed in the identity blockchain. Furthermore, as the output of a VRF is unpredictable as long as the private key sk_i is kept secret, the tickets of non-colluding nodes, hence the outcome of the lottery is also unpredictable. The synchrony bound Δ guarantees that the ticket of an honest leader is seen by all other honest validators. If the adversary wins the lottery, he can decide either to comply and run the RandHound protocol or to fail, which excludes that particular node from participating for the rest of the epoch.

After a successful run of RandHound, the adversary is the first to learn the randomness, hence the sharding assignment, however his benefit is minimal. The adversary can again either decide to cooperate and publish the random value or withhold it in the hope of winning the lottery again and obtaining a sharding assignment that fits his agenda better. However, the probability that an adversary wins the lottery a times in a row is upper bounded by the exponentially decreasing term $(f/n)^a$. Thus, after only a few re-runs of the lottery, an honest node wins with high probability and coordinates the sharding. Finally, we remark that an adversary cannot collect random values from multiple runs and then choose the one he likes best as validators accept only the latest random value that matches their view number v.

In Appendix B, we show how OmniLedger can be extended to probabilistically detect that the expected Δ does not hold and how it can still remain secure with a fall-back protocol.

B. Maintaining Operability During Epoch Transitions

Recall that, in each epoch e, SLedger changes the assignments of all n validators to shards, which results in an idle phase during which the system cannot process transactions until enough validators have finished bootstrapping.

To maintain operability during transition phases, OmniLedger gradually swaps in new validators to each shard per epoch. This enables the remaining operators to continue providing service (in the honest scenario) to clients while the recently joined validators are bootstrapping. In order to achieve this continued operation we can swap-out at most $\frac{1}{3}$ of the shard's size ($\approx \frac{n}{m}$), however the bigger the batch is, the higher the risk gets that the number of remaining honest validators is insufficient to reach consensus and the more stress the bootstrapping of new validators causes to the network.

To balance the chances of a temporary loss of liveness, the shard assignment of validators in OmniLedger works as follows. First, we fix a parameter $k < \frac{1}{3}\frac{n}{m}$ specifying the swap-out batch, *i.e.*, the number of validators that are swapped out at a given time. For OmniLedger, we decided to work in batches of $k = \log \frac{n}{m}$. Then for each shard j, we derive a seed $H(j \parallel \mathsf{rnd}_e)$ to compute a permutation $\pi_{j,e}$ of the shard's validators, and we specify the permutation of the batches. We also compute another seed $H(0 \parallel \mathsf{rnd}_e)$ to permute and scatter the validators who joined in epoch e and to define the order in which they will do so (again, in batches of size k). After defining the random permutations, each batch waits Δ before starting to bootstrap in order to spread the load on the network. Once a validator is ready, he sends an announcement to the shard's leader who then swaps the validator in.

Security Arguments: During the transition phase, we ensure the safety of the BFT consensus in each shard as there are always at least $\frac{2}{3}\frac{n}{m}$ honest validators willing to participate in the consensus within each shard. And, as we use the epoch's randomness rnd_e to pick the permutation of the batches, we keep the shards' configurations a moving target for an adaptive adversary. Finally, as long as there are $\frac{2}{3}\frac{n}{m}$ honest and up-to-date validators, liveness is guaranteed. Whereas if this quorum is breached during transition (the new batch of honest validators has not yet updated), the liveness is lost only temporarily, until the new validators update.

C. Cross-Shard Transactions

To enable value transfer between different shards thereby achieving shard interoperability, support for secure cross-shard transactions is crucial in any sharded-ledger system. We expect that the majority of transactions to be cross-shard in the traditional model where UTXOs are randomly assigned to shards for processing [16], [34], see Appendix C.

A simple but inadequate strawman approach to a cross-shard transaction, is to concurrently send a transaction to several shards for processing because some shards might commit the transaction while others might abort. In such a case, the UTXOs at the shard who accepted the transactions are lost as there is no straightforward way to roll back a half-committed transaction, without adding exploitable race conditions.

To address this issue, we propose a novel *Byzantine Shard Atomic Commit (Atomix)* protocol for *atomically* processing transactions across shards, such that each transaction is either committed or eventually aborted. The purpose is to ensure consistency of transactions between shards, to prevent double spending and to prevent unspent funds from being locked forever. In distributed computing, this problem is known as atomic commit [47] and atomic commit protocols [27], [30] are deployed on honest but unreliable processors. Deploying such protocols in OmniLedger is unnecessarily complex, because the shards are collectively honest, do not crash infinitely, and run ByzCoin (that provides BFT consensus). Atomix improves the strawman approach with a lock-then-unlock process. We intentionally keep the shards' logic simple and make any direct shard-to-shard communication unnecessary by tasking the client with the responsibility of driving the unlock process while permitting any other party (*e.g.*, validators or even other clients) to fill in for the client if a specific transaction stalls after being submitted for processing.

Atomix uses the UTXO state model, see Section II-B for an overview, which enables the following simple and efficient three-step protocol, also depicted in Figure 3.

1) **Initialize.** A client creates a *cross-shard transaction* (cross-TX for short) whose inputs spend UTXOs of some *input shards* (ISs) and whose outputs create new UTXOs in some *output shards* (OSs). The client gossips the cross-TX and it eventually reaches all ISs.
2) **Lock.** *All* input shards associated with a given cross-TX proceed as follows. First, to decide whether the inputs can be spent, each IS leader validates the transaction within his shard. If the transaction is valid, the leader marks within the state that the input is spent, logs the transaction in the shard's ledger and gossips a *proof-of-acceptance*, a signed Merkle proof against the block where the transaction is included. If the transaction is rejected, the leader creates an analogous *proof-of-rejection*, where a special bit indicates an acceptance or rejection. The client can use each IS ledger to verify his proofs and that the transaction was indeed locked. After all ISs have processed the lock request, the client holds enough proofs to either commit the transaction or abort it and reclaim any locked funds, but not both.
3) **Unlock.** Depending on the outcome of the lock phase, the client is able to either commit or abort his transaction.
   a) **Unlock to Commit.** If *all* IS leaders issued proofs-of-acceptance, then the respective transaction can be committed. The client (or any other entity such as an IS leader after a time-out) creates and gossips an *unlock-to-commit transaction* that consists of the lock transaction and a proof-of-acceptance for each input UTXO. In turn, each involved OS validates the transaction and includes it in the next block of its ledger in order to update the state and enable the expenditure of the new funds.
   b) **Unlock to Abort.** If, however, *even* one IS issued a proof-of-rejection, then the transaction cannot be committed and has to abort. In order to reclaim the funds locked in the previous phase, the client (or any other entity) must request the involved ISs to unlock that particular transaction by gossiping an *unlock-to-abort transaction* that includes (at least) one proof-of-rejection for one of the input UTXOs. Upon receiving a request to unlock, the ISs' leaders follow a similar procedure and mark the original UTXOs as spendable again.

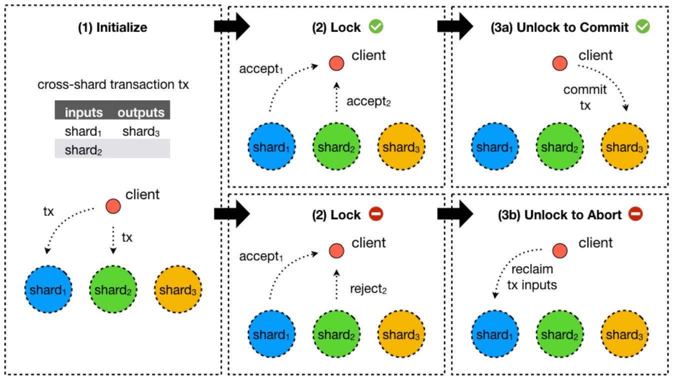

Fig. 3: Atomix protocol in OmniLedger.

We remark that, although the focus of OmniLedger is on the UTXO model, Atomix can be extended with a locking mechanism for systems where objects are long-lived and hold state (*e.g.*, smart contracts [48]), see Appendix D for details.

Security Arguments: We informally argue the previously stated security properties of Atomix, based on the following observations. Under our assumptions, shards are honest, do not fail, eventually receive all messages and reach BFT consensus. Consequently, (1) all shards always faithfully process valid transactions; (2) if all input shards issue a proof-of-acceptance, then every output shard unlocks to commit; (3) if even one input shard issues a proof-of-rejection, then all input shards unlocks to abort; and (4) if even one input shard issues a proof-of-rejection, then no output shard unlocks to commit.

In Atomix, each cross-TX eventually commits or aborts. Based on (1), each input shard returns exactly one response: either a proof-of-acceptance or a proof-of-rejection. Consequently, if a client has the required number of proofs (one per each input UTXO), then the client either only holds proofs-of-acceptance (allowing the transaction to be committed as (2) holds) or not (forcing the transaction to abort as (3) and (4) holds), but not both simultaneously.

In Atomix, no cross-TX can be spent twice. As shown above, cross-shard transactions are atomic and are assigned to specific shards who are solely responsible for them. Based on (1), the assigned shards do not process a transaction twice and no other shard attempts to unlock to commit.

In Atomix, if a transaction cannot be committed, then the locked funds can be reclaimed. If a transaction cannot be committed, then there must exist at least one proof-of-rejection issued by an input shard, therefore (3) must hold. Once all input shards unlock to abort, the funds become available again.

We remark that funds are not automatically reclaimed and a client or other entity must initiate the unlock to abort process. Although this approach poses the risk that if a client crashes indefinitely his funds remain locked, it enables a simplified protocol with minimal logic that requires no direct shard-to-shard communication. A client who crashes indefinitely is equivalent to a client who loses his private key, which prevents him from spending the corresponding UTXOs. Furthermore, any entity in the system, for example a validator in exchange for a fee, can fill in for the client to create an unlock transaction, as all necessary information is gossiped.

To ensure better robustness, we can also assign the shard of the smallest-valued input UTXO to be a coordinator responsible for driving the process of creating unlock transactions. Because a shard's leader might be malicious, $f+1$ validators of the shard need to send the unlock transaction to guarantee that all transactions are eventually unlocked.

Size of Unlock Transactions: In Atomix, the unlock transactions are larger than regular transactions as appropriate proofs for input UTXOs need to be included. OmniLedger relies on ByzCoinX (a novel BFT-consensus described in Section V-A) for processing transactions within each shard. When the shard's validators reach an agreement on a block that contains committed transactions, they produce a collective signature whose size is independent of the number of validators. This important feature enables us to keep Atomix proofs (and consequently the unlock transactions) short, even though the validity of each transaction is checked against the signed blocks of all input UTXOs. If ByzCoinX did not use collective signatures, the size of unlock transactions would be impractical. For example, for a shard of 100 validators a collective signature would only be 77 bytes, whereas a regular signature would be 9KB, almost two order's of magnitude larger than the size of a simple transaction (500 bytes).

V. Design Refinements for Performance

In this section, we introduce the performance sub-protocols of OmniLedger. First, we describe a scalable BFT-consensus called *ByzCoinX* that is more robust and more parallelizable than ByzCoin. Then, we introduce state-blocks that enable fast bootstrapping and decrease storage-costs. Finally, we propose an optional trust-but-verify validation step to provide real-time latency for low-risk transactions

A. Fault Tolerance under Byzantine Faults

The original ByzCoin design offers good scalability, partially due to the usage of a tree communication pattern. Maintaining such communication trees over long time periods can be difficult, as they are quite susceptible to faults. In the event of a failure, ByzCoin falls back on a more robust all-to-all communication pattern, similarly to PBFT. Consequently, the consensus's performance deteriorates significantly, which the adversary can exploit to hinder the system's performance.

To achieve better fault tolerance in OmniLedger, without resorting to a PBFT-like all-to-all communication pattern, we introduce for ByzCoinX a new communication pattern that trades-off some of ByzCoin's high scalability for robustness, by changing the message propagation mechanism within the consensus group to resemble a two-level tree. During the setup of OmniLedger in an epoch, the generated randomness is not only used to assign validators to shards but also to assign them evenly to groups within a shard. The number of groups g, from which the maximum group size can be derived by taking the shard size into account, is specified in the shard policy file. At the beginning of a ByzCoinX roundtrip, the protocol leader randomly selects one of the validators in each group to be the *group leader* responsible for managing communication between the protocol leader and the respective group members. If a group leader does not reply before a predefined timeout, the protocol leader randomly chooses another group member to replace the failed leader. As soon as the protocol leader receives more than $\frac{2}{3}$ of the validators' acceptances, he proceeds to the next phase of the protocol. If the protocol leader fails, all validators initiate a PBFT-like view-change procedure.

B. Parallelizing Block Commitments

We now show how ByzCoinX parallelizes block commitments in the UTXO model by carefully analyzing and handling dependencies between transactions.

We observe that transactions that do not conflict with each other can be committed in different blocks and consequently can be safely processed in parallel. To identify conflicting transactions, we need to analyze the dependencies that are possible between transactions. Let $\texttt{tx}_A$ and $\texttt{tx}_B$ denote two transactions. Then, there are two cases that need to be carefully handled: (1) both $\texttt{tx}_A$ and $\texttt{tx}_B$ try to spend the same UTXO and (2) an UTXO created at the output of $\texttt{tx}_A$ is spent at the input of $\texttt{tx}_B$ (or vice versa). To address (1) and maintain consistency, only one of the two $\texttt{tx}$ can be committed. To address (2), $\texttt{tx}_A$ has to be committed to the ledger before $\texttt{tx}_B$, *i.e.*, $\texttt{tx}_B$ has to be in a block that depends (transitively) on the block containing $\texttt{tx}_A$. All transactions that do not exhibit these two properties can be processed safely in parallel. In particular we remark that transactions that credit the same address do not produce a conflict, because they generate different UTXOs

To capture the concurrent processing of blocks, we adopt a *block-based directed acyclic graph (blockDAG)* [33] as a data structure, where every block can have multiple parents. The ByzCoinX protocol leader enforces that each pending block includes only non-conflicting transactions and captures UTXO dependencies by adding the hashes of former blocks (*i.e.*, backpointers) upon which a given block's transactions depend. To decrease the number of such hashes, we remark that UTXO dependencies are transitive, enabling us to relax the requirement that blocks have to capture all UTXO dependencies directly. Instead, a given block can simply add backpointers to a set of blocks, transitively capturing all dependencies.

C. Shard Ledger Pruning

Now we tackle the issues of an ever-growing ledger and the resulting costly bootstrapping of new validators; this is particularly urgent for high-throughput DL systems. For example, whereas Bitcoin's blockchain grows by $\approx 144\,\mathrm{MB}$ per day and has a total size of about 133 GB, next-generation systems with Visa-level throughput (*e.g.*, 4000 tx/sec and 500 B/tx) can easily produce over 150 GB per day.

To reduce the storage and bootstrapping costs for validators (whose shard assignments might change periodically), we introduce *state blocks* that are similar to stable checkpoints in PBFT [13] and that summarize the entire state of a shard's ledger. To create a state block $\mathsf{sb}_{j,e}$ for shard j in epoch e, the shard's validators execute the following steps: At the end of e, the shard's leader stores the UTXOs in an ordered Merkle tree and puts the Merkle tree's root hash in the header of $\mathsf{sb}_{j,e}$. Afterwards, the validators run consensus on the header of $\mathsf{sb}_{j,e}$ (note that each validator can construct the same ordered Merkle tree for verification) and, if successful, the leader stores the approved header in the shard's ledger making $\mathsf{sb}_{j,e}$ the genesis block of epoch $e+1$. Finally, the body of $\mathsf{sb}_{j,e-1}$ (UTXOs) can be discarded safely. We keep the regular blocks of epoch e, however, until after the end of epoch $e+1$ for the purpose of creating transaction proofs.

As OmniLedger's state is split across multiple shards and as we store only the state blocks' headers in a shard's ledger, a client cannot prove the existence of a past transaction to another party by presenting an inclusion proof to the block where the transaction was committed. We work around this by moving the responsibility of storing transactions' proofs-of-existence to the clients of OmniLedger. During epoch $e+1$ clients can generate proofs-of-existence for transactions validated in epoch e using the normal block of epoch e and the state block. Such a proof for a given transaction tx contains the Merkle tree inclusion proof to the regular block B that committed tx in epoch e and a sequence of block headers from the state block $\mathsf{sb}_{j,e}$ at the end of the epoch to block B. To reduce the size of these proofs, state blocks can include several multi-hop backpointers to headers of intermediate (regular) blocks similarly to skipchains [37].

Finally, if we naively implement the creation of state blocks, it stalls the epoch's start, hence the transaction processing until $\mathsf{sb}_{j,e}$ has been appended to the ledger. To avoid this downtime, the consistent validators of the shard in epoch $e+1$ include an empty state-block at the beginning of the epoch as a place-holder; and once $\mathsf{sb}_{j,e}$ is ready they commit it as a regular block, pointing back to the place-holder and $\mathsf{sb}_{j,e-1}$.

D. Optional Trust-but-Verify Validation

There exists an inherent trade-off between the number of shards (and consequently the size of a shard), throughput and latency, as illustrated in Figure 4. A higher number of smaller shards results in a better performance but provides less resiliency against a more powerful attacker (25%). Because the design of OmniLedger favors security over scalability, we pessimistically assume an adversary who controls 25% of the validators and, accordingly, choose large shards at the cost of higher latency but guarantee the finality of transactions. This assumption, however, might not appropriately reflect the priorities of clients with frequent, latency-sensitive but low-value transactions (*e.g.*, checking out at a grocery store, buying gas or paying for coffee) and who would like to have transactions processed as quickly as possible.

In response to the clients' needs, we augment the intra-shard architecture (see Figure 4) by following a "trust but verify" model, where *optimistic validators* process transactions quickly, providing a provisional but unlikely-to-change commitment and *core validators* subsequently verify again

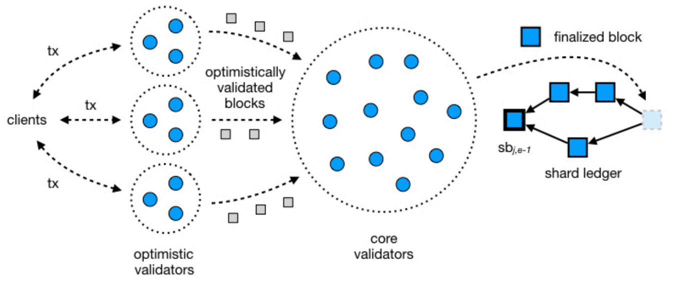

Fig. 4: Trust-but-Verify Validation Architecture

the transactions to provide finality and ensure verifiability. Optimistic validators follow the usual procedures for deciding which transactions are committed in which order; but they form much smaller groups, even as small as one validator per group. Consequently, they produce smaller blocks with real-time latencies but are potentially less secure as the adversary needs to control a (proportionally) smaller number of validators to subvert their operation. As a result, some bad transactions might be committed, but ultimately core validators verify all provisional commitments, detecting any inconsistencies and their culprits, which makes it possible to punish rogue validators and to compensate the defrauded customers for the damages. The trust-but-verify approach strikes a balance for processing small transactions in real-time, as validators are unlikely to misbehave for small amounts of money.

At the beginning of an epoch e, all validators assign themselves to shards by using the per-epoch randomness, and then bootstrap their states from the respective shard's last state block. Then, OmniLedger assigns each validator randomly to one of multiple optimistic processing groups or a single core processing group. The shard-policy file specifies the number of optimistic and core validators, as well as the number of optimistic groups. Finally, in order to guarantee that any misbehavior will be contained inside the shard, it can also define the maximum amount of optimistic validated transactions to be equal to the stake or revenue of the validators.

Transactions are first processed by an optimistic group that produces optimistically validated blocks. These blocks serve as input for re-validation by core validators who run concurrently and combine inputs from multiple optimistic processing groups, thus maximizing the system's throughput (Figure 4). Valid transactions are included in a finalized block that is added to the shard's ledger and are finally included in the state block. However, when core validators detect an inconsistency, then the respective optimistically validated transaction is excluded and the validators who signed the invalid block are identified and held accountable, *e.g.*, by withholding any rewards or by excluding them from the system. We remark that the exact details of such punishments depend on the incentive scheme that are out of scope of this paper. Given a minimal incentive to misbehave and the quantifiable confidence in the security of optimistic validation (Figure 5), clients can choose, depending on their needs, to take advantage of real-time processing with an optimistic assurance of finality or to wait to have their transaction finalized.

VI. Security Analysis

Our contributions are mainly pragmatic rather than theoretical and in this section we provide an informal security analysis supplementing the arguments in Sections IV and V.

A. Randomness Creation

RandHound assumes an honest leader who is responsible for coordinating the protocol run and for making the produced randomness available to others. In OmniLedger, however, we cannot always guarantee that an honest leader will be selected. Although a dishonest leader cannot affect the unbiasability of the random output, he can choose to withhold the randomness if it is not to his liking, thus forcing the protocol to restart. We economically disincentivize such behavior and bound the bias by the randomized leader-election process.

The leader-election process is unpredictable as the adversary is bound by the usual cryptographic hardness assumptions and is unaware of (a) the private keys of the honest validators and (b) the input string x to the VRF function. Also, OmniLedger's membership is unpredictable at the moment of private key selection and private keys are bound to identities. As a result, the adversary has at most $m = 1/4$ chance per round to control the elected leader as he controls at most 25% of all nodes. Each time an adversary-controlled leader is elected and runs RandHound the adversary can choose to accept the random output, and the sharding assignment produced by it, or to forfeit it and try again in hopes of a more favorable yet still random assignment. Consequently, the probability that an adversary controls n consecutive leaders is upper-bounded by $P[X \geq n] = \frac{1}{4^n} < 10^{-\lambda}$. For $\lambda = 6$, the adversary will control at most 10 consecutive RandHound runs. This is an upper bound, as we do not include the exclusion of the previous leader from the consecutive elections.

B. Shard-Size Security

We previously made the assumption that each shard is collectively honest. This assumption holds as long as each shard has less than $c = \lfloor \frac{n}{3} \rfloor$ malicious validators, because ByzCoinX requires $n = 3f + 1$ to provide BFT consensus.

The security of OmniLedger's validator assignment mechanism is modeled as a random sampling problem with two possible outcomes (honest or malicious). Assuming an infinite pool of potential validators, we can use the binomial distribution (Eq. 1). We can assume random sampling due to RandHound's unpredictability property that guarantees that each selection is completely random; this leads to the adversarial power of at most $m = 0.25$.

$$P\left[X \leq \lfloor \frac{n}{3} \rfloor\right] = \sum_{k=0}^{n} \binom{n}{k} m^k (1-m)^{n-k} \tag{1}$$

To calculate the failure rate of one shard, *i.e.*, the probability that a shard is controlled by an adversary, we use the cumulative distributions over the shard size n, where X is the random variable that represents the number of times we pick a malicious node. Figure 5 (right) illustrates the proposed shard size, based on the power of the adversary. In a similar fashion we calculate the confidence a client can have that an optimistic validation group is honest (left).

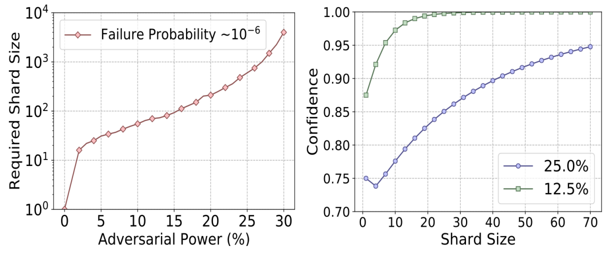

Fig. 5: Left: Shard size required for 10^{-6} system failure probability under different adversarial models. Right: Security of an optimistic validation group for 12.5% and 25% adversaries.

C. Epoch Security

In the last section, we modeled the security of a single shard as a random selection process that does, however, not correspond to the system's failure probability within on epoch. Instead, the total failure rate can be approximated by the union bound over the failure rates of individual shards.

We argue that, given an adequately large shard size, the epoch-failure probability is negligible. We can calculate an upper bound on the total-failure probability by permitting the adversary to run RandHound multiple times and select the output he prefers. This is a stronger assumption than what RandHound permits, as the adversary cannot go back to a previously computed output if he chose to re-run RandHound. An upper bound of the epoch failure event X_E is given by

$$P[X_E] \leq \sum_{k=0}^{l} \frac{1}{4^k} \cdot n \cdot P[X_S] \tag{2}$$

where l is the number of consecutive views the adversary controls, n is the number of shards and $P[X_S]$ is the failure probability of one shard as calculated in Section VI-B. For $l \rightarrow \infty$, we get $P[X_E] \leq \frac{4}{3} \cdot n \cdot P[X_S]$. More concretely, the failure probability, given a 12.5%-adversary and 16 shards, is $4 \cdot 10^{-5}$ or one failure in 68.5 years for one-day epochs

D. Group Communication

We now show that OmniLedger's group-communication pattern has a high probability of convergence under faults. We assume that there are N nodes that are split in $\sqrt{N}$ groups of $\sqrt{N}$ nodes each.

1) Setting the Time-Outs: In order to ensure that the shard leader will have enough time to find honest group leaders, we need to setup the view-change time-outs accordingly. OmniLedger achieves this by having two time-outs. The first timeout T_1 is used by the shard leader to retry the request to non-responsive group members. The second timeout T_2 is used by the group members to identify a potential failure of a shard leader and to initiate a view-change [13]. To ensure that the shard leader has enough time to retry his requests, we have a fixed ratio of $T_1 = 0.1T_2$. However, if the T_2 is triggered, then in the new view T_2 doubles (as is typical [13]) in order to contemplate for increase in the network's asynchrony, hence T_1 should double to respect the ratio.

2) Reaching Consensus: We calculate the probability for a group size $N = 600$ where $\sqrt{N} = 25$: Given a population of 600 nodes and a sampling size of 25, we use the hypergeometric distribution for our calculation which yields a probability of 99.93% that a given group will have less than $25 - 10 = 15$ malicious nodes. A union bound over 25 groups yields a probability of 98.25% that no group will have more than 15 malicious nodes. In the worst case, where there are exactly $\frac{1}{3}$ malicious nodes in total, we need all of the honest validators to reply. For a group that contains exactly 15 malicious nodes, the shard's leader will find an honest group leader (for ByzCoinX) after 10 tries with a probability of $1 - ((15/24)^{10}) = 98.6\%$. As a result, the total probability of failure is $1 - 0.986 * 0.9825 = 0.031$.

We remark that this failure does not constitute a compromise of security of OmniLedger. Rather, it represents the probability of a failure for the shard leader who is in charge of coordinating the shard's operation. If a shard leader indeed fails, then a new shard leader will be elected having 97% probability of successfully reaching consensus.

VII. Implementation

We implemented OmniLedger and its subprotocols for sharding, consensus, and processing of cross-shard transactions in Go [26]. For sharding, we combined RandHound's code, available on GitHub, with our implementation of a VRF-based leader-election mechanism by using a VRF construction similar to the one of Franklin and Zhang [22]. Similarly, to implement ByzCoinX, we extended ByzCoin's code, available on GitHub as well, by the parallel block commitment mechanism as introduced in Section V-B. We also implemented the Atomix protocol, see Section IV-C, on top of the shards and a client that dispatches and verifies cross-shard transactions.

VIII. Evaluation

In this section, we experimentally evaluate our prototype implementation of OmniLedger. The primary questions we want to evaluate concern the overall performance of OmniLedger and whether it truly scales out (Section VIII-B), the cost of epoch transitions (Section VIII-C), the client-perceived latency when committing cross-shard transactions (Section VIII-D), and the performance differences between ByzCoinX and ByzCoin with respect to throughput and latency (Section VIII-E).

A. Experimental Setup

We ran all our experiments on DeterLab [18] using 60 physical machines, each equipped with an Intel E5-2420 v2 CPU, 24 GB of RAM, and a 10 Gbps network link. To simulate a realistic, globally distributed deployment, we restricted the bandwidth of all connections between nodes to 20 Mbps and impose a latency of 100 ms on all communication links. The basis for our experiments was a data set consisting of the first 10,000 blocks of the Bitcoin blockchain.

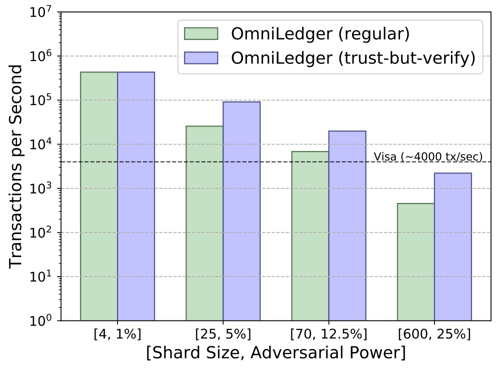

Fig. 6: OmniLedger throughput for 1800 hosts, varying shard sizes s, and adversarial power f/n.

TABLE I: OmniLedger transaction confirmation latency in seconds for different configurations with respect to the shard size s, adversarial power f/n, and validation types.

$[s, f/n]$	$[4, 1\%]$	$[25, 5\%]$	$[70, 12.5\%]$	$[600, 25\%]$
Regular val.	1.38	5.99	8.04	14.52
1st lvl. val.	1.38	1.38	1.38	4.48
2nd lvl. val.	1.38	55.89	41.84	62.96

B. OmniLedger Performance

In this experiment, we evaluate the performance of OmniLedger in terms of throughput and latency in different situations: we distinguish the cases where we have a fixed shard size and varying adversarial power (in particular 1%, 5%, 12.5%, and 25%) or the other way round. We also distinguish between configurations with regular or trust-but-verify validations where we use 1 MB blocks in the former case and 500 KB for optimistically validated blocks and 16 MB for final blocks in the latter case. In order to provide enough transactions for the final blocks, for each shard, there are 32 optimistic validation groups concurrently running; they all feed to one core shard, enabling low latency for low-risk transactions (Table I) and high throughput of the total system.

Figure 6 shows OmniLedger's throughput for 1800 hosts in different configurations and, for comparison, includes the average throughput of Visa at ≈ 4000 tx/sec. Additionally, Table I shows the confirmation latency in the above configuration.

We observe that OmniLedger's throughput with trust-but-verify validation is almost an order of magnitude higher than with regular validation, at the cost of a higher latency for high-risk transactions that require both validation steps. For low-risk transactions, OmniLedger provides an optimistic confirmation in a few seconds after the first validation step, with less than 10% probability that the confirmation was vulnerable to a double-spending attack due to a higher-than-average number of malicious validators. For high-risk transactions, the latency to guarantee finality is still less than one minute.

TABLE II: OmniLedger scale-out throughput in transactions per second (tps) for a adversarial power of $f/n = 12.5\%$ shard size of $s = 70$, and a varying number of shards m.

m	1	2	4	8	16
tps	439	869	1674	3240	5850

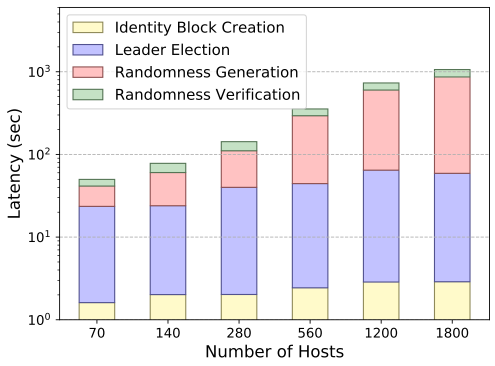

Fig. 7: Epoch transition latency.

Table II shows the scale-out throughput of OmniLedger with a 12.5% adversary, a shard size of 70, and a number of shards m between 1 and 16. As we can see, the throughput increases almost linearly in the number of shards.

In Figure 6, with a 12.5% adversary and a total number of 1800 hosts, we distributed the latter across 25 shards for which we measured a throughput of $13,000$ tps corresponding to 3 times the level of Visa. If we want to maintain OmniLedger's security against a 25% adversary and still achieve the same average throughput of Visa, *i.e.*, 4000 tps, then we estimate that we need to increase the number of hosts to about 4200 (which is less than the number of Bitcoin full nodes [4]) and split them into 7 shards. Unfortunately, our experimental platform could not handle such a high load, therefore, we mention here only an estimated value.

C. Epoch-Transition Costs

In this experiment, we evaluate the costs for transitioning from an epoch $e-1$ to epoch e. Recall, that at the end of epoch $e-1$ the new membership configuration is first collectively signed, then used for the VRF-based leader-election. Once the leader is elected, he runs RandHound with a group-size of 16 hosts (which is secure for a 25% adversary [44]) and broadcasts it to all validators, who then verify the result and connect to their peers. We assume that validators already know the state of the shard they will be validating. It is important to mention that this process is not on the critical path, but occurs concurrently with the previous epoch. Once the new groups have been setup, the new shard leaders enforce a view-change.

As we can see in Figure 7, the cost of bootstrapping is mainly due to RandHound that takes up more than 70% of the total run time. To estimate the worst-case scenario, we refer to our security analysis in Section VI-A and see that, even in the case with 1800 hosts, an honest leader is elected after 10 RandHound runs with high probability, which takes approximately 3 hours. Given an epoch duration of one day, this worst-case overhead is acceptable.

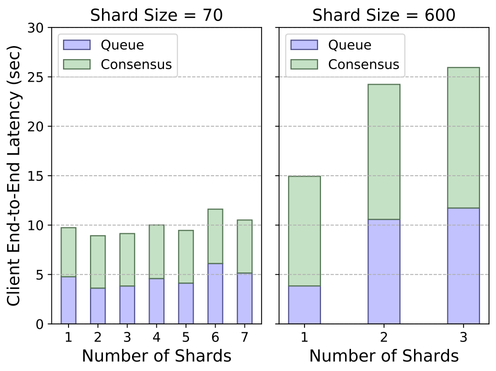

Fig. 8: Client-perceived, end-to-end latency for cross-shard transactions via Atomix.

TABLE III: ByzCoinX latency in seconds for different concurrency levels and data sizes.

	Concurrency			
Data Size	1	2	4	8
1 MB	15.4	13.5	12.6	11.4
8 MB	32.2	27.7	26.0	23.2
32 MB	61.6	58.0	50.1	50.9

D. Client-Perceived End-to-End Latency with Atomix

In this experiment we evaluate in different shard configurations, the client-perceived, end-to-end latency when using Atomix. As shown in Figure 8, the client-perceived latency is almost double the value of the consensus latency as there are already other blocks waiting to be processed in the common case. Consequently, the inclusion of the transaction in a block is delayed. This latency increases slightly further when multiple shards validate a transaction. The overall end-to-end latency would be even higher if a client had to wait for output shards to run consensus that, however, is not required.

If the client wants to directly spend the new funds, he can batch together the proof-of-acceptance and the expenditure transaction in order to respect the input-after-output constraint.

Overall, the client-perceived end-to-end latency for cross-shard transactions is not significantly affected when increasing the number of shards.

E. ByzCoinX Performance

In this experiment, we measure the performance improvements of ByzCoinX over the original ByzCoin. To have a fair comparison, each data-series corresponds to the total size of data concurrently in the network, meaning that if the concurrency level is 2 then there are 2 blocks of 4 MB concurrently, adding to a total of 8 MB, whereas a concurrency level of 4 means 4 blocks of 2 MB each.

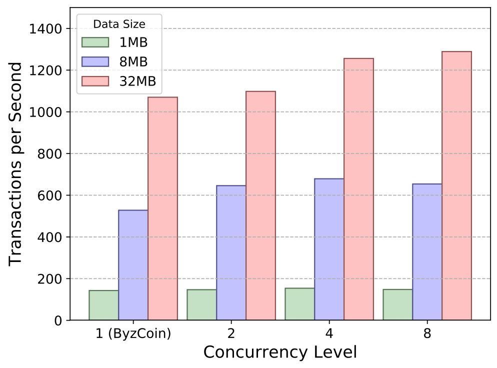

Fig. 9: ByzCoinX throughput in transactions per second for different levels of concurrency.

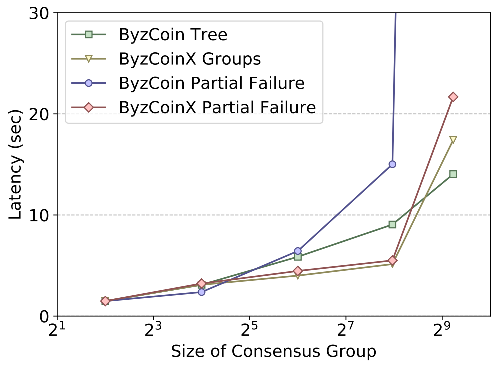

Fig. 10: ByzCoinX communication pattern latency.

In Figures 9 and Table III, we see that there is a 20% performance increase when moving from one big block to four smaller concurrently running blocks, with a *concurrent* 35% decrease in the per-block consensus latency. This can be attributed to the higher resource utilization of the system, when blocks arrive more frequently for validation. When the concurrency further increases, we can see a slight drop in performance, meaning that the overhead of the parallel consensus outweighs the parallelization benefits, due to the constant number of cryptographic operations per block.

Figure 10 illustrates the scalability of ByzCoin's [32] tree and fall-back flat topology, versus ByzCoinX's more fault-tolerant (group-based) topology and its performance when failures occur. As expected the tree topology scales better, but only after the consensus is run among more than 600 nodes, which assumes an adversary stronger than usual (see Figure 5).

For a group size below 600, ByzCoinX's communication pattern actually performs better than ByzCoin's. This is due to ByzCoinX's communication pattern that can be seen as a shallow tree where the roundtrip from root to leaves is faster than in the tree of ByzCoin. Hence, ByzCoin has a fixed branching factor and an increasing depth, whereas ByzCoinX has a fixed depth and an increasing branching factor. The effect of these two choices leads to better latencies for a few hundred nodes for fixed depth. The importance of the group topology, however, is that it is more fault tolerant because when failures occur the performance is not seriously affected. This is not true for ByzCoin; it switches to a flat topology in case of failure that does not scale after a few hundred nodes, due to the huge branching factor. This experiment was run with 1 MB blocks, the non-visible data point is at 300 seconds.

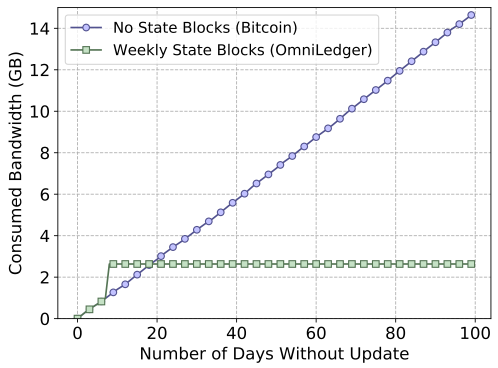

Fig. 11: Bootstrap bandwidth consumption with state blocks.

F. Bandwidth Costs for State Block Bootstrapping

In this experiment, we evaluate the improvements that state blocks offer to new validators during bootstrapping. Recall, that during an epoch transition, a new validator first crawls the identity blockchain, after which he needs to download only the latest state block instead of replaying the full blockchain to create the UTXO state. For this experiment, we reconstructed Bitcoin's blockchain [5], [41] and created a parallel OmniLedger blockchain with weekly state blocks.

Figure 11 depicts the bandwidth overhead of a validator that did not follow the state for the first 100 days. As we can see, the state block approach is better if the validator is outdated for more than 19 days or 2736 Bitcoin blocks.

The benefit might not seem substantial for Bitcoin, but in OmniLedger, 2736 blocks are created in less than 8 hours, meaning that for one day-long epochs, the state block approach is significantly better. If a peak throughput is required and 16 MB blocks are deployed, we expect reduced bandwidth consumption close to two orders of magnitude.

IX. Related Work

The growing interests in scaling blockchains have produced a number of prominent systems that we compare in Table IV. ByzCoin [32] is a first step to scalable BFT consensus, but cannot scale-out. Elastico is the first open scale-out DL, however, it suffers from performance and security challenges that we have already discussed in Section II. RSCoin [16] proposes sharding as a scalable approach for centrally banked cryptocurrencies. RSCoin relies on a trusted source of randomness for sharding and auditing, making its usage problematic in trustless settings. Furthermore, to validate transactions, each shard has to coordinate with the client and instead of running BFT, RSCoin uses a simple two-phase commit, assuming that safety is preserved if the majority of validators is honest. This approach, however, does not protect from double spending attempts by a malicious client colluding with a validator.

TABLE IV: Comparison of Distributed Ledger Systems

System	Scale-Out	Cross-Shard Transaction Atomicity	State Blocks	Measured Scalability (# of Validators)	Estimated Time to Fail	Measured Latency
RSCoin [16]	In Permissioned	Partial	No	30	N/A	1 sec
Elastico [34]	In PoW	No	No	1600	1 hour	800 sec
ByzCoin [32]	No	N/A	No	1008	19 years	40 sec
Bitcoin-NG [21]	No	N/A	No	1000	N/A	600 sec
PBFT [9], [11]	No	N/A	No	16	N/A	1 sec
Nakamoto [36]	No	N/A	No	4000	N/A	600 sec
OmniLedger	Yes	Yes	Yes	2400	68.5 years	1.5 sec

In short, prior solutions [16], [32], [34] achieve only two out of the three desired properties; decentralization, long-term security, and scale-out, as illustrated in Figure 1. OmniLedger overcomes this issue by scaling out, as far as throughput is concerned, and by maintaining consistency to the level required for safety, without imposing a total order.

Bitcoin-NG scales Bitcoin without changing the consensus algorithm by observing that the PoW process does not have to be the same as the transaction validation process; this results in two separate timelines: one slow for PoW and one fast for transaction validation. Although Bitcoin-NG significantly increases the throughput of Bitcoin, it is still susceptible to the same attacks as Bitcoin [24], [3].

Other efforts to scale blockchains include: Tendermint [9], a protocol similar to PBFT for shard-level consensus that does not scale due to its similarities to PBFT, and the Lightning Network [40], an off-chain payment protocol for Bitcoin (also compatible to OmniLedger); it limits the amount of information committed to the blockchain.

Chainspace [2], enhances RSCoin with a more general smart-contract capability. Chainspace also recognizes the need for cross-shard atomic commit but devises a rather complicated algorithm because it chooses to have the shards run the protocol without the use of a client, which increases the cross-shard communication. Our approach is synergistic to Chainspace, as we focus on an open scalable UTXO style DL, whereas Chainspace focuses on sharded smart-contracts and small-scale shards that can be deployed only under weak adversaries. However, combining OmniLedger and Chainspace has great potential to create an open, scalable smart-contract platform that provides scalability and security under strong adversaries.

X. Limitation and Future Work

OmniLedger is still a proof of concept and has limitations that we want to address in future work. First, even if the epoch bootstrap does not interfere with the normal operation, its cost (in the order of minutes) is significant. We leave to future work the use of advanced cryptography, such as BLS [6] for performance improvements. Additionally, the actual throughput is dependent on the workload (see Appendix C). If all transactions touch all the shards before committing, then the system is better off with only one shard. We leave to future work the exploration of alternative ways of sharding, *e.g.*using locality measures. Furthermore, we rely on the fact that honest validators will detect that transactions are unfairly censored and change the leader in the case of censorship. But, further anti-censorship guarantees are needed. We provide a protocol sketch in Appendix A and leave to future work its implementation and further combination with secret sharing techniques for providing stronger guarantees. Another shortcoming of OmniLedger is that it does not formally reason around incentives of participants and focus on the usual honest or malicious devide, which can be proven unrealistic in anonymous open cryptocurrencies. Finally, the system is not suitable for highly adaptive adversaries, as the bootstrap time of an epoch is substantial and scales only moderately, thus leading to the need for day-long epochs.

XI. Conclusion

OmniLedger is the first DL that securely scales-out to offer a Visa-level throughput and a latency of seconds while preserving full decentralization and protecting against a Byzantine adversary. OmniLedger achieves this through a novel approach consisting of three steps. First, OmniLedger is designed with concurrency in mind; both the full system (through sharding) and each shard separately (through ByzCoinX) validate transactions in parallel, maximizing the resource utilization while preserving safety. Second, OmniLedger enables any user to transact safely with any other user, regardless of the shard they use, by deploying *Atomix*, an algorithm for cross-shard transactions as well as real-time validation with the introduction of a trust-but-verify approach. Finally, OmniLedger enables validators to securely and efficiently switch between shards, without being bound to a single anti-Sybil attack method and without stalling between reconfiguration events.

We implemented and evaluated OmniLedger and each of its sub-components. ByzCoinX improves ByzCoin both in performance, with 20% more throughput and 35% less latency, and in robustness against failures. Atomix offers a secure processing of cross-shard transactions and its overhead is minimal compared to intra-shard consensus. Finally, we evaluated the OmniLedger prototype thoroughly and showed that it can indeed achieve Visa-level throughput.

References

[1] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman. Solidus: An Incentive-compatible Cryptocurrency Based on Permissionless Byzantine Consensus. *CoRR*, abs/1612.02916, 2016.

[2] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A Sharded Smart Contracts Platform. *arXiv preprint arXiv:1708.03778*, 2017.

[3] M. Apostolaki, A. Zohar, and L. Vanbever. Hijacking Bitcoin: Large-scale Network Attacks on Cryptocurrencies. *38th IEEE Symposium on Security and Privacy*, May 2017.
[4] Bitnodes. Bitcoin Network Snapshot, April 2017.
[5] Blockchain.info. Blockchain Size, Feb. 2017.
[6] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.
[7] J. Bonneau, J. Clark, and S. Goldfeder. On Bitcoin as a public randomness source. IACR eprint archive, Oct. 2015.
[8] M. Borge, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, and B. Ford. Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies. In *1st IEEE Security and Privacy on the Blockchain*, Apr. 2017.
[9] E. Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains, 2016.
[10] V. Buterin, J. Coleman, and M. Wampler-Doty. Notes on Scalable Blockchain Protocols (verson 0.3), 2015.
[11] C. Cachin. Architecture of the Hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.
[12] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *19th ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000.
[13] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.
[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.
[15] K. Croman, C. Decke, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. S. an, and R. Wattenhofer. On Scaling Decentralized Blockchains (A Position Paper). In *3rd Workshop on Bitcoin and Blockchain Research*, 2016.
[16] G. Danezis and S. Meiklejohn. Centrally Banked Cryptocurrencies. *23rd Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2016.
[17] S. Deetman. Bitcoin Could Consume as Much Electricity as Denmark by 2020, May 2016.
[18] DeterLab Network Security Testbed, September 2012.
[19] J. R. Douceur. The Sybil Attack. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002.
[20] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47. IEEE, 1992.
[21] I. Eyal, A. E. Gencer, E. G. Sirer, and R. van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, Mar. 2016. USENIX Association.
[22] M. K. Franklin and H. Zhang. A Framework for Unique Ring Signatures. *IACR Cryptology ePrint Archive*, 2012:577, 2012.
[23] A. Gervais, G. O. Karame, S. Capkun, and V. Capkun. Is Bitcoin a Decentralized Currency? *IEEE Security and Privacy Magazine*, 12(3):54–60, 2014.
[24] A. Gervais, H. Ritzdorf, G. O. Karame, and S. Capkun. Tampering with the Delivery of Blocks and Transactions in Bitcoin. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 692–705. ACM, 2015.
[25] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies, Oct. 2017.
[26] The Go Programming Language, Feb. 2018.
[27] R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.
[28] T. Hanke and D. Williams. Intoducing Random Beascons Using Threshold Relay Chains, Sept. 2016.
[29] E. G. S. Ittay Eyal. It's Time For a Hard Bitcoin Fork, June 2014.
[30] I. Keidar and D. Dolev. Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 245–254. ACM, 1995.
[31] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. Cryptology ePrint Archive, Report 2016/889, 2016.
[32] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.
[33] Y. Lewenberg, Y. Sompolinsky, and A. Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.
[34] 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*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.
[35] S. Micali, S. Vadhan, and M. Rabin. Verifiable Random Functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 120–130. IEEE Computer Society, 1999.
[36] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.
[37] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1271–1287. USENIX Association, 2017.
[38] R. Pass and E. Shi. Hybrid Consensus: Efficient Consensus in the Permissionless Model. Cryptology ePrint Archive, Report 2016/917, 2016.
[39] R. Pass, C. Tech, and L. Seeman. Analysis of the Blockchain Protocol in Asynchronous Networks. *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2017.
[40] J. Poon and T. Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, Jan. 2016.
[41] Satoshi.info. Unspent Transaction Output Set, Feb. 2017.
[42] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
[43] B. Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *IACR International Cryptology Conference (CRYPTO)*, pages 784–784, 1999.
[44] E. Syta, P. Jovanovic, E. Kokoris-Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. Scalable Bias-Resistant Distributed Randomness. In *38th IEEE Symposium on Security and Privacy*, May 2017.
[45] 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 *37th IEEE Symposium on Security and Privacy*, May 2016.
[46] B. Wiki. Proof of burn , Sept. 2017.
[47] Wikipedia. Atomic commit, Feb. 2017.
[48] G. Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper*, 2014.

Appendix A
Censorship Resistance Protocol

One issue existing in prior work [32], [34] that OmniLedger partially addresses is when a malicious shard leader censors transactions. This attack can be undetectable from the rest of the shard's validators. A leader who does not propose a transaction is acceptable as far as the state is concerned, but this attack can compromise the fairness of the system or be used as a coercion tool.

For this reason, we enable the validators to request transactions to be committed, because they think the transactions are censored. They can either collect those transactions via the normal gossiping process or receive a request directly from a client. This protocol can be run periodically (*e.g.*, once every 10 blocks). We denote $N = 3f+$ validators exist where at most f are dishonest.

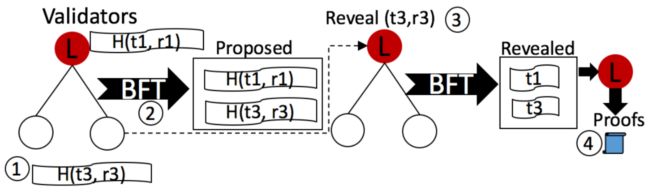

Fig. 12: Anti-censorship mechanism OmniLedger

The workflow (Figure 12), starts ① with each validator proposing a few (e.g 100) blinded transactions for anti-censorship, which initiates a consensus round. The leader should add in the blocks all the proposals, however he can censor f of the honest proposers. Nevertheless, he is blind on the f inputs he has to add from the honest validators he will reach consensus with. Once the round ends, there is a list ② of transactions that are eligible for anti-censorship, which is a subset of the proposed. As the transactions are blinded, no other validator knows which ones are proposed before the end of the consensus. Each validators reveals ③ his chosen transactions, the validators check that the transactions are valid and run consensus on which ones they expect the leader to propose. The leader is then obliged to include ④ the transactions that are consistent with the state, otherwise the honest validators will cause a view-change [13].

Appendix B
Breaking the Network Model

Although DL protocols that assume a non-static group of validators have similar synchrony [34], [36] assumptions, in this section we discuss what can happen if the adversary manages to break them [3]. In such a case we can detect the attack and provide a back-up randomness generation mechanism which is not expected to scale but guarantees safety even in asynchrony.

Given that RandHound guarantees safety without the need for synchrony an adversary manipulates the network can at most slow down any validator he does not control, winning the leadership all the time. However this does not enable the adversary to manipulate RandHound, it just gives him the advantage of being able to restart the protocol if he does not like the random number. This restart will be visible to the network, and the participants can suspect a bias-attempt, when multiple consecutive RandHound rounds start to fail.

OmniLedger can provide a "safety valve" mechanism in order to mitigate this problem. When 5 RandHound views fail in a row, which under normal circumstances could happen with less than 1% probability, the validators switch from RandHound to running a fully asynchronous coin-tossing protocol [12] that uses Publicly Verifiable Secret Sharing [43], in order to produce the epoch's randomness. This protocol scales poorly ($O(n^3)$), but it will be run when the network is anyway under attack and liveness is not guaranteed, in which case safety is more important.

Appendix C
Probability of Cross-Shard Transactions

This section explores the limitations cross-shard transactions pose to the performance of the system. When splitting the state into disjoint parts, the common practice [16], [34] is to assign UTXOs to shards, based on the first bits of their hash. For example, one shard manages all UTXOs whose first bit is 0, and the second shard all UTXOs whose first bit is 1. Then each shard is managed by a group of validators who keep the state consistent and commit updates.

For an intrashard transaction we want all the inputs and outputs of the transaction to be assigned at the shame shard. The probability of assigning a UTXO in a shard is uniformly random from the randomness guarantees of cryptographic hash functions. Let m be the total number of shards, n the sum of input and output UTXOs and k the number of shards that need to participate in the cross-shard validation of the transaction. The probability can be calculated as:

$$P(n,k,m) = \begin{cases} 1, n=1, k=1 \\ 0, n=1, k\neq 1 \\ \frac{m-k}{m}P(n-1,k-1,m)+ \\ \frac{k}{m}P(n-1,k,m), n\neq 1, k>0 \end{cases} \quad (3)$$

For a typical transaction with two inputs and one output and a three-shard setup, the probability of a transaction being intra-shard is $P(3,1,3) = 3.7\%$, rendering the assumption that transactions touch only one shard [34] problematic. As a result if all transactions had this format the expected speed-up from an 1-shard to a 4-shard configuration would be $4*(0.015+\frac{0.328}{2}+\frac{0.56}{3}+\frac{0.09}{4}) = 1.56$ Generalizing this we should expect that the speed-up will be lower than the one of experiment VIII, depending on the average amount of inputs and outputs and the total number of shards.

Appendix D
Atomix for State-full Objects

The original Atomix protocol in Section IV implements a state machine as depicted in Figure 13

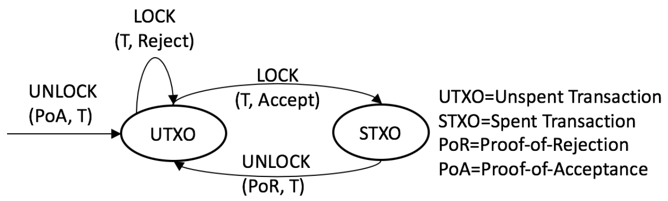

Fig. 13: State-Machine for state-full objects. Pessimistic locking is necessary

To enable the use of such an algorithm in smart contracts we need to account on the fact that a smart-contract object is mutable and can be accessed concurrently for a legitimate reason. As a result we need to modify the algorithm in two ways: a) the Unlock transactions should be send to both Input and Output shards and b) the state machine should have one more state as the shards need to wait for confirmation before unlocking. This is necessary because there is the chance that the (state-full) object will be accessed again and this could violate the input-after-output dependency if Atomix decides to abort.

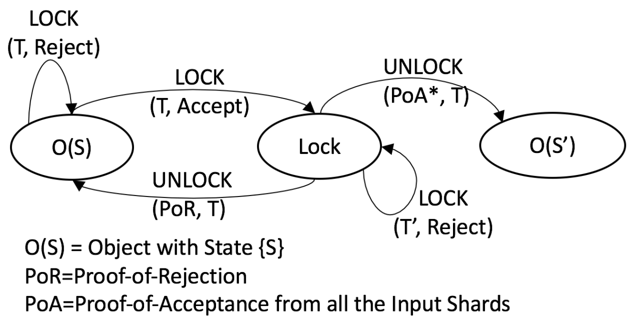

Fig. 14: State-Machine for state-full objects. Pessimistic locking is necessary

In Figure 14, we can see that an object will Lock for a specific transaction (T) and will reject any concurrent T', until T is committed and the new state S' is logged, or aborted and the old state S is open for change again.