Divide and Scale: Formalization of Distributed Ledger Sharding Protocols

ANONYMOUS AUTHOR(S)

Sharding distributed ledgers is the most promising on-chain solution for scaling blockchain technology. In this work, we define and analyze the properties a sharded distributed ledger should fulfill. More specifically, we show that a sharded blockchain with n participants cannot be scalable under a fully adaptive adversary, but it can scale up to $O(n/\log n)$ under an epoch-adaptive adversary. This is possible only if the distributed ledger creates succinct proofs of the valid state updates at the end of each epoch. Our model builds upon and extends the Bitcoin backbone protocol by defining *consistency* and *scalability*. Consistency encompasses the need for atomic execution of cross-shard transactions to preserve safety, whereas scalability encapsulates the speedup a sharded system can gain in comparison to a non-sharded system. We introduce a protocol abstraction and highlight the sufficient components for secure and efficient sharding in our model. In order to show the power of our framework, we analyze the most prominent sharded blockchains (Elastico, Monoxide, OmniLedger, RapidChain) and pinpoint where they fail to meet the desired properties.

1 Introduction

One of the most promising solutions to scaling blockchain protocols is sharding, e.g. [26, 29, 42, 44]. Its high-level idea is to employ multiple blockchains in parallel, the *shards*, that operate under the same consensus. Different sets of participants run consensus and validate transactions, so that the system "scales out".

However, there is no formal definition for a robust sharded ledger (similar to the definition of what a robust transaction ledger is [17]), which leads to multiple problems. First, each protocol defines its own set of goals, which tend to favor the presented protocol design. These goals are then confirmed achievable by experimental evaluations that demonstrate their improvements. Additionally, due to the lack of robust comparisons (which cannot cover all possible byzantine behaviors), sharding is often criticized as some believe that the overhead of transactions between shards cancel out the potential benefits. In order to fundamentally understand sharding, one must formally define what sharding really is, and then see whether different sharding techniques live up to their promise.

In this work, we take up the challenge of providing formal "common grounds" under which we can capture the sharding limitations and fairly compare different sharding solutions. We achieve this by defining a formal sharding framework as well as formal bounds of what a sharded transaction ledger can achieve. Then, we apply our framework to multiple sharded ledgers and identify why they do not satisfy our definition of a robust sharded transaction ledger.

To maintain compatibility with the existing models of a robust transaction ledger, we build upon the work of Garay et al. [17]. We generalize the blockchain transaction ledger properties, originally introduced in [17], namely *Persistence* and *Liveness*, to also apply on sharded ledgers. Persistence essentially expresses the agreement between honest parties on the transaction order, while liveness encompasses that a transaction will eventually be processed and included in the transaction ledger. Further, we extend the model to capture what sharding offers to blockchain systems by defining *Consistency* and *Scalability*. Consistency is a security property that conveys the atomic property of *cross-shard transactions* (transactions that span multiple shards and should either abort or commit in all shards). Scalability, on the other hand, is a performance property that encapsulates the speedup of a sharded blockchain system compared to a non-sharded blockchain system.

Once we define the properties, we explore the boundaries of sharding protocols to maintain the security and performance properties of our framework. We observe a fine balance between consistency and scalability. To truly work, a sharding protocol needs to scale in computation, bandwidth, and storage. The computational cost of each participant in a blockchain system is dominated by verifying the consistency of the updates (either directly executing them or

verifying proofs of consistency). We show that each participant of a sharding protocol must only participate in the verification process of a constant number of shards to satisfy scalability. Additionally, we show that scalable sharding is impossible if participants maintain (and verify) information on all other shards with which they execute cross-shard transactions. This happens because the storage requirements become proportional to that of a non-sharded protocol.

Finally, the high communication complexity of the underlying protocols, i. e., consensus of shards, handling cross-shard transactions, and bootstrapping of participants to shards, can make the bandwidth become the bottleneck. We identify a trade-off between the bandwidth requirements and how adaptive the adversary is, i. e., how "quickly" the adversary can change the corrupted nodes. If the adversary is not static, maintaining security requires shard reassignments and the participants need to bootstrap from the stored and ever-increasing blockchain of the shard. In this work, we prove that *in the long run* there is no sharding protocol that scales better than the blockchain substrate against an adaptive adversary. Furthermore, we show that against a somewhat adaptive adversary, sharding protocols must periodically compact the state updates (e. g., via checkpoints [26], cryptographic accumulators [7], zero-knowledge proofs [6, 31] or other techniques [12, 19–21]); otherwise, the necessary bandwidth resources of each participant increase infinitely in time, eventually exceeding the resources needed in a non-sharded blockchain.

Once we provide solid limits and bounds on the design of sharding protocols, we identify six specific components that are critical in designing a robust permissionless sharded ledger: (a) a core consensus protocol for each shard, (b) an atomic cross-shard communication protocol that enables transferring of value across shards, (c) a Sybil-resistance mechanism that forces the adversary to commit resources in order to participate, (d) a process that guarantees honest and adversarial nodes are appropriately dispersed to the shards to defend security against adversarial adaptivity, (e) a distributed randomness generation protocol that is critical for security, (f) a process to occasionally compact the state in a verifiable manner. We then employ these components to introduce a protocol abstraction, termed the sharding crux, that achieves robust sharding in our model. We explain the design rationale and provide security proofs while we determine which components affect the efficiency of the protocol abstraction, i. e., scalability and throughput.

To demonstrate the power of our framework, we further evaluate the most original and impactful[1] permisionless sharding protocols. Specifically, we describe, abstract, and analyze Elastico [29] (inspiration of Zilliqa), OmniLedger [26] (inspiration of Harmony), Monoxide [42], and RapidChain [44]. We demonstrate that all sharding systems fail to meet the desired properties in our system model. Elastico and Monoxide do not actually (asymptotically) improve on non-sharded blockchains according to our model. OmniLedger is susceptible to a liveness attack where the adaptive adversary can simply delete a shard's state effectively preventing the system's progress. Nevertheless, a simple fix is possible such that OmniLedger satisfies all the desired properties in our model. Last, we prove RapidChain meets the desired properties but only in a weaker adversarial model. For all protocols, we provide elaborate proofs while for OmniLedger and RapidChain we further estimate how much they improve over their blockchain substrate. To that end, we define and use a throughput factor, that expresses the average number of transactions that can be processed per round. We show that both OmniLedger and RapidChain scale optimally, reaching the upper bound $O(\frac{n}{\log n})$.

Our Contribution. In summary, the contribution of this work is the following:

- We introduce a framework where sharded transaction ledgers are formalized and the necessary properties of sharding protocols are defined. Further, we define a throughput factor to estimate the transaction throughput improvement of sharding blockchains over non-sharded blockchains (Section 2).
- We provide the limitations of secure and efficient sharding protocols under our model (Section 3).

[1]We consider all sharding protocols presented in top tier computer science conferences.

- We identify the critical and sufficient ingredients for designing a robust sharded ledger (Section 4). We further present the sharding crux, a protocol abstraction for secure sharding, demonstrating how these ingredients yield a robust sharded ledger.
- We evaluate existing protocols in our model. We show that Elastico, Monoxide, and OmniLedger fail to satisfy some of the defined properties in our model, whereas RapidChain satisfy all the necessary properties to maintain a robust sharded transaction ledger but only under a weaker adversarial model (Section 5, Appendix C).

Omitted proofs can be found in the Appendices.

2 The sharding framework

In this section, we introduce a formal definition of sharded transaction ledgers and define the desired properties of a secure and efficient distributed sharded ledger. Further, we employ the properties of blockchain protocols defined by Garay et al. [17], which we assume to hold in every shard. Given these properties, we later explore the limitations of sharding, introduce a protocol abstraction that satisfies them, and evaluate if the existing sharding protocols maintain secure and efficient sharded transaction ledgers. In addition, we define a metric to evaluate the efficiency of sharding protocols, the transaction throughput.

To assist the reader, we provide a glossary of the most frequently used parameters in Table 2 (Appendix A).

2.1 The Model

2.1.1 Network model. We analyze blockchain protocols assuming a synchronous communication network. In particular, a protocol proceeds in *rounds*, and at the end of each round the participants of the protocol are able to synchronize, and all messages are delivered. A set of R consecutive rounds $E = \{r_1, r_2, \ldots, r_R\}$ defines an *epoch*. We consider a fixed number of participants in the system denoted by n. However, this number might not be known to the parties.

2.1.2 Threat model. The adversary is slowly-adaptive, meaning that the adversary can corrupt parties on the fly at the beginning of each epoch but cannot change the malicious set of participants during the epoch, i. e., the adversary is static during each epoch. In addition, in any round, the adversary decides its strategy after receiving all honest parties' messages. The adversary can change the order of the honest parties' messages but cannot modify or drop them. Furthermore, the adversary is computationally bounded and can corrupt at most f parties during each epoch. This bound f holds strictly at every round of the protocol execution. Note that depending on the specifications of each protocol, i. e., which Sybil-attack-resistant mechanism is employed, the value f represents a different manifestation of the adversary's power (e. g., computational power, stake in the system).

2.1.3 Transaction model. In this work, we assume transactions consist of inputs and outputs that can only be spent as a whole. Each transaction input is an unspent transaction output (UTXO). Thus, a transaction takes UTXOs as inputs, destroys them and creates new UTXOs, the outputs. A transaction ledger that handles such transactions is UTXO-based (as opposed to a typical account-based database), similarly to Bitcoin [32]. Note that transactions can have multiple inputs and outputs. We define the average size of a transaction, i.e., the average number of inputs and outputs of a transaction in a transaction set, as a parameter v [2]. Further, we assume a transaction set T follows a distribution D_T (e.g. D_T is the uniform distribution if the sender(s) and receiver(s) of each transaction are chosen uniformly at random from all possible users). All sharding protocols considered in this work are UTXO-based.

[2]This way v correlates to the number of shards a transaction is expected to affect. The actual size in bytes is proportional to v but unimportant for measuring scalability.

2.2 Sharded Transaction Ledgers

In this section, we define what a robust sharded transaction ledger is. We build upon the definition of a robust public transaction ledger introduced in [17]. Then, we introduce the necessary properties a sharding blockchain protocol must satisfy in order to maintain a robust sharded transaction ledger.

A sharded transaction ledger is defined with respect to a set of valid[3] transactions T and a collection of transaction ledgers for each shard $S = \{S_1, S_2, \ldots, S_m\}$. In each shard $i \in [m] = \{1, 2, \ldots, m\}$, a transaction ledger is defined with respect to a set of valid ledgers[4] S_i and a set of valid transactions. Each set possesses an efficient membership test. A ledger $L \in S_i$, is a vector of sequences of transactions $L = \langle x_1, x_2, \ldots, x_l \rangle$, where $tx \in x_j \Rightarrow tx \in T, \forall j \in [l]$.

In a sharding blockchain protocol, a sequence of transactions $x_i = tx_1 \ldots tx_e$ is inserted in a block which is appended to a party's local chain C in a shard. A chain C of length l contains the ledger $L_C = \langle x_1, x_2, \ldots, x_l \rangle$ if the input of the j-th block in C is x_j. The *position* of transaction tx_j in the ledger of a shard L_C is the pair (i, j) where $x_i = tx_1 \ldots tx_e$ (i. e., the block that contains the transaction). Essentially, a party *reports* a transaction tx_j in position i only if its local ledger of a shard includes transaction tx_j in the i-th block. We assume that a block has constant size, meaning there is a maximum constant number of transactions included in each block.

Furthermore, we define a symmetric relation on T, denoted by $M(\cdot, \cdot)$, that indicates if two transactions are conflicting, i. e., $M(tx, tx') = 1 \Leftrightarrow tx, tx'$ are conflicting. Note that valid ledgers can never contain conflicting transactions. Similarly, a valid sharded ledger cannot contain two conflicting transactions even across shards. In our model, we assume there exists a verification oracle denoted by $V(T, S)$, which instantly verifies the validity of a transaction with respect to a ledger. In essence, the oracle V takes as input a transaction $tx \in T$ and a valid ledger $L = \langle x_1, x_2, \ldots, x_l \rangle \in S$ and checks whether the transaction is valid and not conflicting in this ledger; formally, $V(tx, L) = 1 \Leftrightarrow \exists tx' \in L$ s.t. $M(tx, tx') = 1$ or $L' = \langle x_1, x_2, \ldots, x_l, tx \rangle$ is an invalid ledger.

Next, we introduce the security and performance properties a blockchain protocol must uphold to maintain a robust and efficient sharded transaction ledger: persistence, consistency, liveness, and scalability.

Intuitively, *persistence* expresses the agreement between honest parties on the transaction order, whereas *consistency* conveys that cross-shard transactions are either committed or aborted atomically (in all shards). On the other hand, *liveness* indicates that transactions will eventually be included in a shard, i. e., the system makes progress. Last, *scalability* encapsulates the speedup a sharded system can gain in comparison to a non-sharded blockchain system: The blockchain throughput limitation stems from the need for data propagation, maintenance, and verification from every party. Thus, to scale a blockchain protocol via sharding, each party must broadcast, maintain and verify mainly local information.

Definition 1 (Persistence)**.** *Parameterized by $k \in \mathbb{N}$ ("depth" parameter), if in a certain round an honest party reports a shard that contains a transaction tx in a block at least k blocks away from the end of the shard's ledger (such transaction will be called "stable"), then whenever tx is reported by any honest party it will be in the same position in the shard's ledger.*

Definition 2 (Consistency)**.** *Parametrized by k ("depth" parameter), $v \in \mathbb{N}$ (average size of transactions) and D_T (distribution of the input set of transactions), there is no round r in which there are two honest parties P_1, P_2 reporting transactions tx_1, tx_2 respectively as stable (at least in depth k in the respective shards), such that $M(tx_1, tx_2) = 1$.*

To evaluate the system's progress, we assume that the block size is sufficiently large, thus a transaction will never be excluded due to space limitations.

[3]Validity depends on the application using the ledger.
[4]Only one of the ledgers is actually committed as part of the shard's ledger, but before commitment there are multiple potential ledgers.

Definition 3 (Liveness)**.** *Parameterized by u ("wait time"), k ("depth" parameter), $v \in \mathbb{N}$ (average size of transactions) and D_T (distribution of the input set of transactions), provided that a valid transaction is given as input to all honest parties of a shard continuously for the creation of u consecutive rounds, then all honest parties will report this transaction at least k blocks from the end of the shard, i. e., all report it as stable.*

Scaling distributed ledgers depends on three vectors: *communication*, *space*, and *computation*. In particular, to allow high transaction throughput all these vectors should ideally be constant and independent of the number of participants in the system.

First, we define a *communication overhead factor* ω_m as the communication complexity of the protocol scaled over the number of participants. In essence, ω_m represents the average amount of sent or received data per participant in a protocol execution; in other words, the average bandwidth resource requirements of the system's participants. The communication overhead factor expresses both the bandwidth requirements of the protocol execution within an epoch (i. e., within and across shards communication), as well as the bandwidth resources necessary during epoch transitions. We observe that the latter becomes the bottleneck for scalability in the long run as rotating participants must bootstrap to new shards.

We notice, however, that in practise there are protocols that maintain low communication complexity – by employing efficient diffusion mechanisms of data such as gossip protocols – but fail to scale in another dimension. A notable such example is the Bitcoin protocol where the bottleneck is space and computation, while the communication overhead is minimal. For this reason, we introduce the *space overhead factor* ω_s that estimates the space complexity of a sharding protocol, i. e., how much data is stored in total by all the participants of the system in comparison to a single database that only stores the data once. The space overhead factor ranges from $O(1)$ to $O(n)$, where n is the fixed number of participants in the protocol. For instance, the Bitcoin protocol has overhead factor $O(n)$ since every party needs to store all data, while a perfectly scalable blockchain system has a constant space overhead factor.

To define the space overhead factor we need to introduce the notion of average-case analysis. Typically, sharding protocols scale well when the analysis is optimistic, in essence for transaction inputs that neither contain cross-shard nor multi-input (multi-output) transactions. However, in practice transactions are both cross-shard and multi-input/output. For this reason, we define the space overhead factor as a random variable dependent on an input set of transactions T drawn uniformly at random from a distribution D_T.

We assume T is given well in advance as input to all parties. To be specific, we assume every transaction $tx \in T$ is given at least for u consecutive rounds to all parties of the system. Hence, from the liveness property, all transaction ledgers held by honest parties will report all transactions in T as stable. Further, we denote by $L^{\lceil k}$ the vector L where the last k positions are "pruned", while $|L^{\lceil k}|$ denotes the number of transactions contained in this "pruned" ledger. We note that a similar notation holds for a chain C where the last k positions map to the last k blocks. Each party P_j, maintains a collection of ledgers $SL_j = \{L_1, L_2, \ldots, L_s\}, 1 \leq s \leq m$. We define the space overhead factor for a sharding protocol with input T as the number of stable transactions included in every party's collection of transaction ledgers over the number of transactions given as input in the system[5],

$$\omega_s(T) = \sum_{\forall j \in [n]} \sum_{\forall L \in SL_j} |L^{\lceil k}|/|T|$$

Apart from space and communication complexity, we also consider the verification process, which can be computationally expensive. In our model, we assume the computational cost is constant per verification. Every time a

[5]Without loss of generality, we assume all transactions are valid and thus are eventually included in all honest parties' ledgers.

party checks if a transaction is invalid or conflicting with a ledger, the running time is considered constant since this process can always speed up using efficient data structures (e. g. trees allow for logarithmic lookup time). Therefore, the computational power spent by a party for verification is defined by the number of times the party executes the verification process. For this purpose, we employ a verification oracle V. Each party calls the oracle to verify transactions, pending or included in a block. We denote by q_i the number of times party P_i calls oracle V during the protocol execution. We now define a *computational overhead factor* ω_c that reflects the total number of times all parties call the verification oracle during the protocol execution scaled over the number of input transactions,

$$\omega_c(T) = \sum_{\forall i \in [n]} q_i / |T|$$

Note that similarly to the space overhead factor, the computational overhead factor is also a random variable. Hence, the objective is to calculate the expected value of ω_c, i. e., the probability-weighted average of all possible values, where the probability is taken over the input transactions T.

In an ideally designed protocol, communication, space, and computational overhead factors express the same amount of information. However, there can be poorly designed protocols that reduce the overhead in only one dimension but fail to limit the overhead in the entire system. For this reason, we define the scaling factor of a protocol to be the total number of participants over the maximum of the three overhead factors,

$$\Sigma = \frac{n}{\max(\omega_m, \omega_s, \omega_c)}$$

Intuitively if all the overhead factors are constant, then every node has a constant overhead. Hence, the system can scale linearly without imposing any additional overhead to the parties. Furthermore, a sharding system can consist of multiple protocols; the scaling factor of the sharding system is therefore defined as the the number of participants divided by the maximum overhead factor of all the system's protocols. We can now define the scalability property of sharded distributed ledger protocols as follows.

Definition 4 (Scalability)**.** *Parameterized by n (number of participants), $v \in \mathbb{N}$ (average size of transactions), D_T (distribution of the input set of transactions), any sharding blockchain protocol that scales requires a scaling factor $\Sigma = \omega(1)$.*

In order to adhere to standard security proofs from now on we say that the protocol Π *satisfies* property Q in our model if Q holds with overwhelming probability (in a security parameter). Note that a probability p is overwhelming if $1 - p$ is negligible. A function $negl(k)$ is negligible if for every $c > 0$, there exists an $N > 0$ such that $negl(k) < 1/k^c$ for all $k >\geq N$. Furthermore, we denote by $\mathbb{E}(\cdot)$ the expected value of a random variable.

Definition 5 (Robust Sharded Transaction Ledger)**.** *A protocol that satisfies the properties of persistence, consistency, liveness, and scalability maintains a robust sharded transaction ledger.*

2.3 (Sharding) Blockchain Protocols

In this section, we adopt the definitions and properties of [17] for blockchain protocols, while we slightly change the notation to fit our model. In particular, we assume the parties of a shard of any sharding protocol maintain a chain (ledger) to achieve consensus. This means that every shard internally executes a blockchain (consensus) protocol that has three properties as defined by [17]: chain growth, chain quality, and common prefix. Each consensus protocol satisfies these properties with different parameters.

In this work, we will use the properties of the shards' consensus protocol to prove that a sharding protocol maintains a robust sharded transaction ledger. In addition, we will specifically use the shard growth and shard quality parameters

to estimate the transaction throughput of a sharding protocol. The following definitions follow closely Definitions 3, 4 and 5 of [17].

Definition 6 (Shard Growth Property)**.** *Parametrized by $\tau \in \mathbb{R}$ and $s \in \mathbb{N}$, for any honest party P with chain C, it holds that for any s rounds there are at least $\tau \cdot s$ blocks added to chain C of P.*

Definition 7 (Shard Quality Property)**.** *Parametrized by $\mu \in \mathbb{R}$ and $l \in \mathbb{N}$, for any honest party P with chain C, it holds that for any l consecutive blocks of C the ratio of honest blocks in C is at least μ.*

Definition 8 (Common Prefix Property)**.** *Parametrized by $k \in \mathbb{N}$, for any pair of honest parties P_1, P_2 adopting chains C_1, C_2 (in the same shard) at rounds $r_1 \leq r_2$ respectively, it holds that $C_1^{\lceil k} \preceq C_2$, where $\preceq$ denotes the prefix relation.*

Next, we define the *degree of parallelism* (DoP) of a sharding protocol, denoted m'. To evaluate the DoP of a protocol with input T, we need to determine how many shards are affected by each transaction on average; essentially, estimate how many times we run consensus for each valid transaction until it is stable. This is determined by the mechanism that handles the cross-shard transactions. To that end, we define $m_{i,j} = 1$ if the j-th transaction of set T has either an input or an output that is assigned to the i-th shard; otherwise $m_{i,j} = 0$. Then, the DoP of a protocol execution over a set of transactions T is defined as follows: $m' = \frac{T \cdot m}{\sum_{j=1}^{T} \sum_{i=1}^{m} m_{i,j}}$. The DoP of a protocol execution depends on the distribution of transactions D_T, the average size of transactions v, and the number of shards m. For instance, assuming a uniform distribution D_T, the expected DoP is $\mathbb{E}(m') = m/v$.

We can now define an efficiency metric, the *transaction throughput* of a sharding protocol. Considering constant block size, we define the transaction throughput as follows:

Definition 9 (Throughput)**.** *The expected transaction throughput in s rounds of a sharding protocol with m shards is $\mu \cdot \tau \cdot s \cdot m'$. We define the throughput factor of a sharding protocol $\sigma = \mu \cdot \tau \cdot m'$.*

Intuitively, the throughput factor expresses the average number of blocks that can be processed per round by a sharding protocol. Thus, the transaction throughput (per round) can be determined by the block size multiplied by the throughput factor. The block size is considered constant, however, it cannot be arbitrarily large. The limit on the block size is determined by the bandwidth of the "slowest" party within each shard. At the same time, the constant block size guarantees low latency. If the block size is very large or depends on the number of shards or the number of participants, *bandwidth or latency* becomes the performance bottleneck. Note that our goal is to estimate the efficiency of the parallelism of transactions in the protocol, thus other factors like cross-shard communication latency are omitted.

3 Limitations of sharding protocols

In this section, we explore the limits of sharding protocols. First, in Section 3.1, we focus on the limitations that stem from the nature of the transaction workload. In particular, sharding protocols are affected by two characteristics of the input transaction set: the number of inputs and outputs of each transaction or transaction size v, and more importantly the number of cross-shard transactions.

The number of inputs and outputs (UTXOs) of each transaction is in practice fairly small. For example, an average Bitcoin transaction has 2 inputs and 3 outputs with a small deviation (e. g. 1) [1]. Hence, we consider the number of UTXOs participating in each transaction fixed, thus the transaction size v is a small constant. Furthermore, the largest the transaction size, the more shards are affected by each transaction on expectation, hence the largest the ratio of cross-shard transactions. Thus, to meaningfully upper bound the ratio of cross-shard transactions, we consider the minimum transaction size $v = 2$.

The number of cross-shard transactions depends on the distribution of the input transactions D_T, as well as the process that partitions transactions to shards. First, we assume each ledger interacts (i. e., has a cross-shard transaction) with γ other ledgers on average, where γ is a function dependent on the number of shards m. We consider protocols that require parties to maintain information on shards other than their own and derive an upper bound for the expected value of γ such that scalability holds. Then, we prove there is no protocol that maintains a robust sharded ledger against an adaptive adversary.

Later, in Section 3.2, we assume shards are created with a uniformly random process, i. e., the UTXO space is partitioned uniformly at random into shards. Under this assumption (which most sharding systems make), we show a constant fraction of the transactions are expected to be cross-shard, and there is no sharded ledger that satisfies scalability if participants have to maintain any information on ledgers others than their own. Note that our results hold for *any distribution* where the expected number of cross-shard transactions is *proportional* to the number of shards.

Last, in Section 3.3, we examine the case where parties are assigned to shards independently and uniformly at random. This assumption is met by almost all known sharding systems to guarantee security against non-static adversaries. We show that any sharding protocol can scale at most by a factor of $n/\log n$ in our model. Finally, we demonstrate the importance of periodical compaction of the valid state-updates in sharding protocols; we prove that any sharding protocol that satisfies scalability in our security model requires a state-compaction process such as checkpoints [26], cryptographic accumulators [7], zero-knowledge proofs [6], non-interactive proofs of proofs-of-work [12, 21], proof of necessary work [20], or erasure codes [19].

3.1 General Bounds

First, we prove there is no robust sharded transaction ledger that has a constant number of shards. Then, we show that there is no protocol that maintains a robust sharded transaction ledger against an adaptive adversary.

Lemma 10. *In any robust sharded transaction ledger the number of shards (parametrized by n) is* $m = \omega(1)$.

Suppose a party is participating in shard x_i. If the party maintains information (e. g. the headers of the chain for verification purposes) on the chain of shard x_j, we say that the party is a *light node* for shard x_j. In particular, a light node for shard x_j maintains information at least proportional to the length of the shard's chain x_j. Sublinear light clients [12, 21] verifiably compact the shard's state, thus are not considered as light nodes but are discussed later.

Lemma 11. *For any robust sharded transaction ledger that requires every participant to be a light node for all the shards affected by cross-shard transactions, it holds* $\mathbb{E}(\gamma) = o(m)$.

Next, we show that there is no protocol that maintains a robust transaction ledger against an adaptive adversary in our model. We highlight that our result holds because we assume *any node is corruptible* by the adversary. If we assume more restrictive corruption sets, e. g. each shard has at least one honest well-connected node, sharding against an adaptive adversary may be possible if we employ other tools, such as fraud and data availability proofs [3].

Theorem 12. *There is no protocol maintaining a robust sharded transaction ledger against an adaptive adversary with* $f \geq n/m$.

Proof. (Towards contradiction) Suppose there exists a protocol Π that maintains a robust sharded ledger against an adaptive adversary that corrupts $f = n/m$ parties. From the pigeonhole principle, there exists at least one shard x_i with at most n/m parties (independent of how shards are created). The adversary is adaptive, hence at any round can

corrupt all parties of shard x_i. In a malicious shard, the adversary can perform arbitrary operations, thus can spend the same UTXO in multiple cross-shard transactions. However, for a cross-shard transaction to be executed it needs to be accepted by the output shard, which is honest. Now, suppose Π allows the parties of each shard to verify the ledger of another shard. For Lemma 11 to hold, the verification process can affect at most $o(m)$ shards. Note that even a probabilistic verification, i. e., randomly select some transactions to verify, can fail due to storage requirements and the fact that the adversary can perform arbitrarily many attacks. Therefore, for each shard, there are at least 2 different shards that do not verify the cross-shard transactions. Thus, the adversary can simply attempt to double-spend the same UTXO across every shard and will succeed in the shards that do not verify the validity of the cross-shard transaction. Hence, consistency is not satisfied. □

3.2 Bounds under Uniform Shard Creation

In this section, we assume that the creation of shards is UTXO-dependent; transactions are assigned to shards independently and uniformly at random. This assumption is in sync with the proposed protocols in the literature. In a non-randomized process of creating shards, the adversary can precompute and thus bias the process in a permissionless system. Hence, all sharding proposals employ a random process for shard creation. Furthermore, all shards validate approximately the same amount of transactions; otherwise the efficiency of the protocol would depend on the shard that validates most transactions. For this reason, we assume the UTXO space is partitioned to shards uniformly at random. Note that we consider UTXOs to be random strings.

Under this assumption, we prove a constant fraction of transactions are cross-shard on expectation. As a result, we show there is no sharding protocol that maintains a robust sharded ledger when participants act as light clients on all shards involved in cross-shard transactions. Our observations hold for any transaction distribution D_T that results in a constant fraction of cross-shard transactions.

Lemma 13. *The expected number of cross-shard transactions is* $\Theta(|T|)$.

Lemma 14. *For any protocol that maintains a robust sharded transaction ledger, it holds* $\gamma = \Theta(m)$.

Theorem 15. *There is no protocol that maintains a robust sharded transaction ledger when participants are light nodes on the shards involved in cross-shard transactions.*

Proof. Immediately follows from Lemmas 11 and 14. □

3.3 Bounds under Random Party to Shard Assignment

In this section, we assume parties are assigned to shards uniformly at random. Any other shard assignment strategy yields equivalent or worse guarantees since we have no knowledge on which parties are byzantine. Our goal is to upper bound the number of shards for a protocol that maintains a robust sharded transaction ledger in our security model. To satisfy the security properties, we demand each shard to contain at least a constant fraction of honest parties $1 - a$ ($< 1 - \frac{f}{n}$). This is due to classic lower bounds of consensus protocols [28].

The *size* of a shard is the number of the parties assigned to the shard. We say shards are *balanced* if all shards have approximately the same size. We denote by $p = f/n$ the (constant) fraction of the byzantine parties. A shard is *a-honest* if at least a fraction of $1 - a$ parties in the shard are honest.

Lemma 16. *Given n parties are assigned uniformly at random to shards and the adversary corrupts at most $f = pn$ parties (p constant), all shards are a-honest (a constant) with probability $\frac{1}{n^c}$ only if the number of shards is at most $m = \frac{n}{c' \ln n}$, where $c' \geq c\frac{2+a-p}{(a-p)^2}$.*

Corollary 17. *A sharding protocol maintains a robust sharded transaction ledger against an adversary with $f = pn$, only if $m = \frac{n}{c' \ln n}$, where $c' \geq c\frac{5/2-p}{(1/2-p)^2}$.*

Next, we prove that any sharding protocol may scale at most by an $n/\log n$ factor. This bound refers to independent nodes. If, for instance, we shard per authority, with all authorities represented in each shard, the bound of the theorem does not hold and the actual system cannot be considered sharded since every authority holds all the data.

Theorem 18. *Any protocol that maintains a robust sharded transaction ledger in our security model has scaling factor at most $\Sigma = O(\frac{n}{\log n})$.*

Proof. In our security model, the adversary can corrupt $f = pn$ parties, p constant. Hence, from Corollary 17, $m = O(\frac{n}{\log n})$. Each party stores at least T/m transactions on average and thus the expected space overhead factor is $\omega_s \geq n\frac{T/m}{T} = \frac{n}{m}$. Therefore, the scaling factor is at most $m = O(\frac{n}{\log n})$. □

Next, we show that any sharding protocol that satisfies scalability requires some process of *verifiable compaction of state*, such as checkpoints [26], cryptographic accumulators [7], zero-knowledge proofs [6], non-interactive proofs of proofs-of-work [12, 21], proof of necessary work [20], erasure codes [19]. Such a process allows the state of the distributed ledger (e. g., stable transactions) to be compressed significantly while users can verify the correctness of the state. Intuitively, in any sharding protocol secure against a slowly adaptive adversary parties must periodically shuffle in shards. To verify new transactions the parties must receive a verifiably correct UTXO pool for the new shard without downloading the full shard history; otherwise the communication overhead of the bootstrapping process eventually exceeds that of a non-sharded blockchain. Although existing evaluations typically ignore this aspect with respect to bandwidth, we stress its importance in the long-term operation: *the bootstrap cost will eventually become the bottleneck due to the need for nodes to regularly shuffle.*

Theorem 19. *Any protocol that maintains a robust sharded transaction ledger in our security model employs verifiable compaction of the state.*

Proof. (Towards contradiction) Suppose there is a protocol that maintains a robust sharded ledger without employing any process that verifiably compacts the blockchain. To guarantee security against a slowly-adaptive adversary, the parties change shards at the end of each epoch. At the beginning of each epoch, the parties must process a new set of transactions. To check the validity of this new set of transactions, each (honest) shard member downloads and maintains the corresponding ledger. Note that even if the party only maintains the hash-chain of a ledger, the cost is equivalent to maintaining the list of transactions given that the block size is constant. We will show that the communication overhead factor increases with time, eventually exceeding that of a non-sharded blockchain; thus scalability is not satisfied from that point on.

In each epoch transition, a party changes shards with probability $1 - 1/m$, where m is the number of shards. As a result, a party changing a shard in epoch k must download the shard's ledger of size $\frac{k \cdot T}{m}$. Therefore, the expected communication overhead factor of bootstrapping during the k-th epoch transition is $\frac{k \cdot T}{m} \cdot (1 - \frac{1}{m})$. We observe the

communication overhead grows with the number of epochs k, hence it will eventually become the scaling bottleneck. For instance, for $k > m \cdot n$, the communication factor is greater than linear to the number of parties in the system n, thus the protocol does not satisfy scalability. □

Theorem 19 holds even if parties are not assigned to shards uniformly at random but follow some other shuffling strategy like in [34]. *As long as a significant fraction of honest parties change shards from epoch to epoch, verifiable compaction of state is necessary* to restrict the bandwidth requirements during bootstrapping in order to satisfy scalability.

4 The Sharding Crux

In this section, we first discuss our design rationale for robust sharding. Using the bounds provided in Section 3, we deduce some sufficient components for robust sharding in our model. We then introduce a *protocol abstraction* for robust sharding, termed *the sharding crux*, employing the introduced sharding components. We prove our protocol abstraction to be secure in our model (assuming the components are secure) and evaluate its efficiency with respect to the choices of the individual components.

4.1 Sharding Components

We explain our design rationale and introduce the ingredients of a protocol that maintains a robust sharded ledger.

(a) **Consensus protocol of shards or `Consensus`:** A sharding protocol either runs consensus in every shard separately (multi-consensus) or provides a single total ordering for all the blocks generated in each shard (uni-consensus). Since uni-consensus takes polynomial cost per block, such a protocol can only scale if the block size is also polynomial (e.g., includes $\Omega(n)$ transactions). However, in such a case, the resources of each node generating an $\Omega(n)$-sized block must also grow with n, and therefore scalability would not be satisfied. Hence, in our protocol abstraction we chose a multi-consensus approach.

The consensus protocol run per shard must satisfy the properties defined by Garay et al. [17], namely common prefix, chain quality and chain growth. These properties are necessary (but not sufficient) to ensure the persistence, liveness, and consistency of the sharding system.

(b) **Cross-shard mechanism or `CrossShard`:** The cross-shard transacting mechanism is the protocol that handles the transactions that span across multiple shards. The cross-shard mechanism is critical to the security of the sharding system as it guarantees consistency. Furthermore, it is also crucial for scalability as a naively designed cross-shard mechanism may induce high storage or communication overhead on the nodes when handling several cross-shard transactions. To that end, the limitations from Section 3 apply for any protocol designed to maintain a robust sharded transaction ledger. The cross-shard transaction mechanism should provide the ACID properties (just like in database transactions). Durability and Isolation are provided directly by the blockchains of the shards, hence, the cross-shard transaction protocol should provide Consistency (every transaction that commits produces a semantically valid state) and Atomicity (transactions are committed and aborted atomically - all or nothing). Typically the cross-shard protocol runs on top of the consensus of the shards, meaning that it invokes the consensus protocol to guarantee Consistency.

(c) **Sybil-resistance mechanism or `Sybil`:** Necessary to the global consensus on the new set of Sybil resistant identities in a permissionless setting, which guarantees the security bounds of the consensus protocol (e. g., $f < 1/3$ for BFT). Given the slowly adaptive adversary, the Sybil-resistance mechanism needs to depend on the unknown randomness of the previous epoch. The exact protocol (PoW, PoS, etc.) is irrelevant.

(d) **Division of nodes to shards or `Divide2Shards`:** From Theorem 12 we know that no system can shard the state securely if the adversary is fully adaptive. Therefore, we design our protocol abstraction in a slowly-adaptive adversarial

model – static adversaries are an easier subcase. To guarantee the security properties against a slowly adaptive adversary, nodes must be divided to shards appropriately.

To satisfy transaction finality (i. e., liveness and persistence), either the consensus security bounds must hold for each shard, or the protocol must guarantee with high probability that if the adversary compromises a shard then the security violation will be restored within a specific (small) number of rounds. To be specific, if an adversary completely or partially compromises a shard, effectively violating the consensus bounds, then the adversary can double spend within the shard (violates persistence), as well as across shards (because nodes cannot verify cross-shard transactions from Lemma 11). Therefore, the transactions included in these blocks can only be executed when honest participants have verified them. Partial solutions towards this direction have been proposed such as fraud proofs that allow an honest party to later prove a misbehavior. Another problem with this approach is the need to guarantee data availability.

Due to the complexity of such solutions and their implications on the transactions' finality, we design our protocol assuming the security bounds of consensus are maintained when nodes are divided into shards. Specifically, we assume that at the beginning of each epoch, the nodes are shuffled among the shards to guarantee the consensus bounds. A secure shuffling process typically requires a source of randomness (i. e., see the DRG protocol below). When assigned to a shard, nodes need to update their local state with the state of the new shard they are asked to secure.

(e) **Randomness generation protocol or `DRG`:** Necessary for any setting and should provide unbiasable randomness [8, 11, 13, 16, 27, 36, 40] such that both the Sybil-resistance mechanism and the protocol that divides nodes to shards result in shards that maintain the security bounds for the consensus protocol. Due to the slowly-adaptive adversary assumption, the DRG protocol must be executed (at least) once per epoch, while its high communication complexity can be amortized over the rounds of an epoch such that the system remains scalable.

(f) **Verifiable compaction of state or `CompactState`:** At the end of each epoch, the history of the shards' ledgers should be summarized such that new parties can bootstrap with minimal effort (Theorem 19). To this end, a process for verifiable compaction of the state-updates must be employed. Then, the compacted state must be broadcast to all the nodes in the network via reliable broadcast [10] to ensure data integrity and data availability to the node that will be assigned to each shard in the next epoch. Note that any protocol that ensures data binding and data availability can be used. This step must take place before the new epoch begins as the slowly adaptive adversary is allowed to compromise the nodes of a shard in the previous epoch, violating the liveness of the system.

For the purpose of our analysis, we consider that the protocol that partitions the state (e. g., transaction space) into shards yields the worst possible outcome. In other words, we consider all transactions to be cross-shard, because any secure protocol that can perform well in the most pessimistic case, can obviously also perform well where transactions are intra-shard. We further note that designing a secure sharding system when most transactions are intra-shard is trivial, hence not the study of this work. Last, we assume that all protocols satisfy liveness in our security model.

4.2 The Sharding Crux

In this section, we introduce a protocol abstraction that achieves robust sharding in our model. In the protocol design we employ the components of sharding as black boxes, and later prove security and analyze the performance of the protocol depending on the choice of the components.

4.3 Analysis

We show that the sharding crux abstraction is secure in our model (i. e., satisfies persistence, consistency, and liveness), while its efficiency (i. e., scalability and throughput factor) depends on the chosen subprotocols.

Protocol Abstraction 1: The Sharding Crux

Data: N_0 nodes are participating in the system at round 0 (genesis block). $m(N_E)$ denotes the function that determines the number of shards in epoch E. The transactions of epoch E are T_E. i denotes the block round (its relation to the communication rounds depends on the employed components).
Result: Shard state $T = \{T_0, T_1, \dots\}$.

/* Initialization */
1 $i \leftarrow 1$
2 $E \leftarrow 0$
/* Beginning of epoch: retrieve identities from Sybil resistant protocol, execute the DRG protocol to create the new epoch randomness, and assign nodes to shards */
3 **if** $i \mod R = 1$ **:**
4 $E \leftarrow E + 1$
5 **if** $i \neq 1$ **:**
6 $N_E \leftarrow \texttt{Sybil}(r_{E-1})$
7 $r_E \leftarrow \texttt{DRG}(N_E)$
8 Call `Divide2Shards`$(N_E, m(N_E), r_E)$
/* End of epoch: compact the state of the shard */
9 **elif** $i \mod R = 0$ **:**
10 Call `CompactState(i)`
/* During epoch: run the consensus protocol for intra-shard and cross-shard transactions */
11 **else:**
12 **if** *If transaction* $t \in T_E$ *is cross-shard* **:**
13 Call `CrossShard(t)` ; // Invokes Consensus in multiple shards
14 **else:**
15 Call `Consensus(t)`
16 $i \leftarrow i + 1$
17 Go to step 3

Theorem 20. *The sharding crux satisfies persistence in our system model assuming at most f byzantine nodes.*

Proof. Assuming `Divide2Shard` respects the security bounds of `Consensus`, the common prefix property is satisfied in each shard, so persistence is satisfied. □

Theorem 21. *The sharding crux satisfies consistency in our system model assuming at most f byzantine nodes.*

Proof. Transactions can either be intra-shard (all UTXOs within a single shard) or cross-shard. Consistency is satisfied for intra-shard transactions as long as `Divide2Shard` respects the security bounds of `Consensus`, hence the common prefix property is satisfied. Furthermore, consistency is satisfied for cross-shard transactions from the `CrossShard` protocol as long as it correctly provides atomicity (provided by assumption). □

Theorem 22. *The sharding crux satisfies liveness in our system model assuming at most f byzantine nodes.*

Proof. Follows from the assumption that all subprotocols satisfy liveness , as well as the `CompactState` protocol that ensures data availability between epochs. □

Scalability. The scaling factor of sharding crux depends on the maximum overhead factor of all the components it employs. The maximum overhead factor for `DRG, Divide2Shards, Sybil`, and `CompactState` can be amortized over

the rounds of an epoch because these protocols are executed once each epoch. Thus, the size of an epoch is critical for scalability. Intuitively, this implies that *if the size of the epoch is small, hence the adversary highly-adaptive, sharding is not that beneficial as the protocols that are executed on the epoch transaction are as resource demanding as the consensus in a non-sharded system.*

Throughput factor. Similarly to scalability, the throughput factor also depends on the employed subprotocols, and in particular, `Consensus` and `CrossShards`. To be specific, the throughput factor depends on the shard growth and shard quality parameters which are determined by `Consensus`. In addition, given a transaction input, the degree of parallelism, which is the last component of the throughput factor, is determined by the maximum number of shards possible and the way cross-shard transactions are handled. The maximum number of shards depends on `Consensus` and `Divide2Shards`, while `CrossShard` determines how many shards are affected by a single transaction. For instance, if the transactions are divided in shards uniformly at random, the sharding crux can scale at most by $n/\log n$ as stated in Corollary 17. We further note that the minimum number of affected shards for a specific transaction is the number of UTXOs that map to different shards; otherwise security cannot be guaranteed.

We demonstrate in Appendix C how to calculate the scaling factor and the throughput factor for OmniLedger and RapidChain.

5 Evaluation of sharding protocols

We evaluate existing sharding protocols with respect to the desired properties defined in Section 2.2. In particular, we analyze Elastico, Monoxide, OmniLedger, and RapidChain. The analysis can be found in Appendix C, while a summary of our evaluation is illustrated in Table 1.

Table 1. Summarizing sharding protocol properties under our model

Protocol	Persistence	Consistency	Liveness	Scalability	Permissionless	Slowly-adaptive
Elastico	✓	✗	✓	✗	✓	✓
Monoxide	✓	✓	✓	✗	✓	✓
OmniLedger	✓	✓	✗	✓	✓	✓
RapidChain	✓	✓	✓	✓	✓	~
Chainspace	✓	✓	✓	✓	✗	✗

We first show that Elastico does not satisfy consistency due to its `CrossShard` protocol (as long as the workload does have cross-shard transactions). Additionally, Elastico does not satisfy scalability as all epoch-transition protocols (e. g., `Divide2Shards`,`CompactState`) are executed for every block (epoch in Elastico). We highlight that Elastico does not satisfy scalability by design regardless the transaction distribution; that is, even with a few cross-shard transactions, Elastico maintains a global hash chain which is broadcast to all the system's participants, and thus the scaling factor is constant at best (determined by the block size).

We then show that Monoxide does not satisfy scalability because the state is not partitioned when security properties are satisfied. We observe that the lack of scalability stems from the protocol's mechanism to defend against adversaries in the PoW setting; hence it cannot scale even in an optimistic transaction distribution with no cross-shard transactions.

Third, we prove that OmniLedger maintains all properties but liveness. Specifically, OmniLedger uses checkpoints of the UTXO pool at the end of each epoch, but the state is not broadcast to the network. Therefore, OmniLedger is vulnerable to slowly adaptive adversaries that can corrupt a shard from the previous epoch before the new nodes of the shard bootstrap to the state during epoch transition. This attack violates the liveness property of the system.

Nevertheless, simply adding a reliable broadcast step at the end of each epoch restores the liveness of OmniLedger, as all other components satisfy liveness. The overhead of this step can be amortized over the rounds of the epoch hence scalability is also maintained.

Fourth, we prove RapidChain maintains a robust sharded ledger but only under a weaker model than the one defined in Section 2. Specifically, the protocol only allows a constant number of parties to join or leave and the adversary can at most corrupt a constant number of additional parties with each epoch transition. Another shortcoming of RapidChain is the synchronous consensus mechanism it employs. In case of temporary loss of synchrony in the network, the consensus of cross-shard transactions is vulnerable, hence consistency might break [44]. However, most of these drawbacks can be addressed with simple solutions, such as changing the consensus protocol (trade-off performance with security), replacing the epoch transition process with one similar to (fixed) OmniLedger, etc. Although OmniLedger (with the proposed fix) maintains a robust sharded ledger in a stronger model (as defined in Section 2), RapidChain introduces practical speedups on specific components of the system. These improvements are not asymptotically important – and thus not captured by our framework – but might be significant for the performance of deployed sharding protocols.

Finally, we include Chainspace in the comparison of sharding protocols, which maintains a robust sharded transaction ledger but only in the permissioned setting against a static adversary. Chainspace could be secure in our model in the permissioned setting if it adopts OmniLedger's epoch transition protocols and the proposed fix for data availability in the verifiable compaction of state. We omit the security proofs for Chainspace since they are either included in [2] or are similar to OmniLedger. We include in the evaluation the "permissionless" and "slowly-adaptive" property, in addition to the security and efficiency properties, namely persistence, consistency, liveness and scalability.

Lastly, we note that the cross-shard communication protocols of some of these sharding systems suffer from replay attacks [39]. In our analysis, we consider the fixes proposed in [39].

6 Related work

The Bitcoin backbone protocol [17] was the first to formally define and prove a blockchain protocol, more specifically Bitcoin in a PoW setting. Later, Pass et al. [33] showed that there is no PoW protocol that can be robust under asynchrony. With Ouroboros [23] Kiayias et al. extended the ideas of backbone to the Proof-of-Stake (PoS) setting, where they showed that it is possible to have a robust transaction ledger in a semi-synchronous environment as well [15]. Our work is also extending the backbone framework to define a robust sharded ledger. We also analyze state-of-the-art sharding protocols under our framework.

Recently, a few systemization of knowledge papers on sharding [41] as well as consensus [5] and cross-shard communication [45] which have also discussed part of sharding, have emerged. Unlike these, our focus is not a complete list of existing protocols. Instead, we formally define what a secure and efficient sharded ledger is, evaluate existing sharding protocols under this lens, by either proving or disproving their claims in our framework, and provide a roadmap to robust sharding.

7 Conclusion

In this paper we provided the first complete analysis of sharded blockchains to enable "apples-to-apples" comparison. To this end we first proved several impossibility results and lower bounds on the security and scalability of sharded distributed ledgers. Equipped with the knowledge of these limitations we formalized the "sharding crux", a protocol composition that maintains a robust sharded distributed ledger. Finally, we investigated existing sharding proposals and showed that their lack of formal treatment leads to all of them being subpar (or clearly broken) to the sharding crux.

References

[1] [n.d.]. Bitcoin Statistics on Transaction UTXOs. https://bitcoinvisuals.com/. Accessed: 2020-11-20.

[2] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2018. Chainspace: A sharded smart contracts platform. *25th Annual Network and Distributed System Security Symposium* (2018).

[3] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. 2018. Fraud and Data Availability Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities. *arXiv preprint: 1809.09044* (2018).

[4] Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. 2018. Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *European Symposium on Research in Computer Security*. Springer, 111–131.

[5] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the Age of Blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. ACM, 183–198.

[6] Eli Ben-Sasson. 2020. A Cambrian Explosion of Crypto Proofs. https://nakamoto.com/cambrian-explosion-of-crypto-proofs/.

[7] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*. Springer, 561–586.

[8] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. 2015. On Bitcoin as a public randomness source. *IACR Cryptology ePrint Archive, Report 2015/1015* (2015).

[9] Maria Borge, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. 2017. Proof-of-personhood: Redemocratizing permissionless cryptocurrencies. In *IEEE European Symposium on Security and Privacy Workshops*. 23–26.

[10] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. *J. ACM* 32, 4 (1985), 824–840.

[11] Benedikt Bünz, Steven Goldfeder, and J. Bonneau. 2017. Proofs-of-delay and randomness beacons in Ethereum. In *IEEE Security and Privacy on the Blockchain*.

[12] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. 2020. FlyClient: Super-Light Clients for Cryptocurrencies. In *IEEE Symposium on Security and Privacy*. 928–946.

[13] Ignacio Cascudo and Bernardo David. 2017. SCRAPE: Scalable randomness attested by public entities. In *International Conference on Applied Cryptography and Network Security*. Springer, 537–556.

[14] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. 173–186.

[15] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. 2018. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 66–98.

[16] Paul Feldman. 1987. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 427–438.

[17] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 281–310.

[18] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 51–68.

[19] Swanand Kadhe, Jichan Chung, and Kannan Ramchandran. 2019. SeF: A secure fountain architecture for slashing storage costs in blockchains. *arXiv preprint: 1906.12140* (2019).

[20] Assimakis Kattis and Joseph Bonneau. 2020. Proof of Necessary Work: Succinct State Verification with Fairness Guarantees. *IACR Cryptology ePrint Archive, Report 2020/190* (2020).

[21] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. 2020. Non-interactive proofs of proof-of-work. In *International Conference on Financial Cryptography and Data Security*. Springer, 505–522.

[22] Aggelos Kiayias and Giorgos Panagiotakos. 2017. On Trees, Chains and Fast Transactions in the Blockchain. In *5th International Conference on Cryptology and Information Security in Latin America*. 327–351.

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

[24] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium*. 279–296.

[25] Eleftherios Kokoris-Kogias. 2019. Robust and Scalable Consensus for Sharded Distributed Ledgers. *IACR Cryptology ePrint Archive, Report 2019/676* (2019).

[26] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *39th IEEE Symposium on Security and Privacy*. IEEE, 583–598.

[27] Eleftherios Kokoris-Kogias, Dahlia Malkhi, and Alexander Spiegelman. 2020. Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures. In *27th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1751–1767.

[28] Leslie Lamport, Robert Shostak, and Marshall Pease. 2019. The Byzantine generals problem. In *Concurrency: The Works of Leslie Lamport*. 203–226.

[29] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 17–30.

[30] Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*. Springer, 53–65.
[31] Izaak Meckler and Evan Shapiro. 2018. Coda: Decentralized cryptocurrency at scale. https://cdn.codaprotocol.com/static/coda-whitepaper-05-10-2018-0.pdf.
[32] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.
[33] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 643–673.
[34] Ranvir Rana, Sreeram Kannan, David Tse, and Pramod Viswanath. 2020. Free2Shard: Adaptive-adversary-resistant sharding via Dynamic Self Allocation. *arXiv preprint: 2005.09610* (2020).
[35] Ling Ren, Kartik Nayak, Ittai Abraham, and Srinivas Devadas. 2017. Practical synchronous byzantine consensus. *arXiv preprint: 1704.02397* (2017).
[36] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. 2018. HydRand: Practical Continuous Distributed Randomness. *IACR Cryptology ePrint Archive, Report 2018/319* (2018).
[37] Siddhartha Sen and Michael J Freedman. 2012. Commensal cuckoo: Secure group partitioning for large-scale services. *ACM SIGOPS Operating Systems Review* 46, 1 (2012), 33–39.
[38] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*. Springer, 507–527.
[39] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. 2019. Replay Attacks and Defenses Against Cross-shard Consensus in Sharded Distributed Ledgers. *arXiv preprint: 1901.11218* (2019).
[40] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. 2017. Scalable bias-resistant distributed randomness. In *IEEE Symposium on Security and Privacy*. 444–460.
[41] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. 2019. SoK: Sharding on Blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. ACM, 41–61.
[42] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale out Blockchains with Asynchronous Consensus Zones. In *16th USENIX Symposium on Networked Systems Design and Implementation*. 95–112.
[43] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*. ACM, 347–356.
[44] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 931–948.
[45] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. 2019. SoK: Communication Across Distributed Ledgers. *IACR Cryptology ePrint Archive, Report 2019/1128* (2019).

A Glossary

B Limitations of Sharding Protocols

Lemma 10. *In any robust sharded transaction ledger the number of shards (parametrized by n) is* $m = \omega(1)$.

Proof. Suppose there is a protocol that maintains a constant number m of sharded ledgers, denoted by $x_1, x_2, \ldots, x_m$. Let n denote the number of parties and T the number of transactions to be processed (wlog assumed to be valid). A transaction is processed only if it is stable, i. e., is included deep enough in a ledger (k blocks from the end of the ledger where k a security parameter). Each ledger will include T/m transactions on expectation. Now suppose each party participates in only one ledger (best case), thus broadcasts, verifies and stores the transactions of that ledger only. Hence, every party stores T/m transactions on expectation. The expected space overhead factor is $\omega_s = \sum_{\forall i \in [n]} \sum_{\forall x \in L_i} |x^{\lceil k}|/|T| = \sum_{\forall x \in L_i} \frac{T}{mT} = \frac{n}{m} = \Theta(n)$. Thus, $\Sigma = \Theta(1)$ and scalability is not satisfied. □

Lemma 11. *For any robust sharded transaction ledger that requires every participant to be a light node for all the shards affected by cross-shard transactions, it holds* $\mathbb{E}(\gamma) = o(m)$.

Proof. We assumed that every ledger interacts on average with γ different ledgers, i. e., the cross-shard transactions involve γ many different shards on expectation. The block size is considered constant, meaning each block includes at most e transactions where e is constant. Thus, each party maintaining a ledger and being a light node to γ other ledgers

Table 2. (Glossary) The parameters in our analysis.

n	number of parties
f	number of byzantine parties
m	number of shards
v	average transaction size (number of inputs and outputs)
E	epoch, i. e., a set of consecutive rounds
T	set of transactions (input)
k	"depth" security parameter (persistence)
u	"wait" time (liveness)
Σ	scaling factor
ω_m	communication overhead factor
ω_s	space overhead factor
ω_c	computational overhead factor
σ	throughput factor
μ	chain quality parameter
τ	chain growth parameter
v	average transaction size
m'	degree of parallelism
γ	average number of a shard's interacting shards (cross-shard)

must store on expectation $(1 + \frac{\gamma}{e})\frac{T}{m}$ information. Hence, the expected space overhead factor is

$$\mathbb{E}(\omega_s) = \sum_{\forall i \in [n]} \sum_{\forall x \in L_i} |x^{\lceil k}|/|T| = n\frac{(1+\frac{\gamma}{e})\frac{T}{m}}{T} = \Theta\Big(\frac{\gamma n}{m}\Big)$$

where the second equation holds due to linearity of expectation. To satisfy scalability, we demand $\Sigma = \omega(1)$, hence $\mathbb{E}(\omega_s) = o(n)$, thus $\gamma = o(m)$. □

Lemma 13. *The expected number of cross-shard transactions is* $\Theta(|T|)$.

Proof. Let Y_i be the random variable that shows if a transaction is cross-shard; $Y_i = 1$ if $tx_i \in T$ is cross-shard, and 0 otherwise. Since UTXOs are assigned to shards uniformly at random, $Pr[i \in x_k] = \frac{1}{m}$, for all $i \in v$ and $k \in [m] = \{1, 2, \ldots, m\}$. The probability that all UTXOs in a transaction $tx \in T$ belong to the same shard is $\frac{1}{m^{v-1}}$ (where v is the cardinality of UTXOs in tx). Hence, $Pr[Y_i = 1] = 1 - \frac{1}{m^{v-1}}$. Thus, the expected number of cross-shard transactions is $\mathbb{E}(\sum_{\forall tx_i \in T} Y_i) = |T|\big(1 - \frac{1}{m^{v-1}}\big)$. Since, $m(n) = \omega(1)$ (Lemma 10) and v constant, the expected cross-shard transactions converges to T for n sufficiently large. □

Lemma 14. *For any protocol that maintains a robust sharded transaction ledger, it holds* $\gamma = \Theta(m)$.

Proof. We assume each transaction has a single input and output, hence $v = 2$. This is the worst-case input for evaluating how many shards interact per transaction; if $v >> 2$ then each transaction would most probably involve more than two shards and thus each shard would interact with more different shards for same set on transactions.

For $v = 2$, we can reformulate the problem as a graph problem. Suppose we have a random graph G with m nodes, each representing a shard. Now let an edge between nodes u, w represent a transaction between shards u, w. Note that in this setting we allow self-loops, which represent the intra-shard transactions. We create the graph G with the following random process: We choose an edge independently and uniformly at random from the set of all possible edges including self-loops, denoted by E'. We repeat the process independently $|T|$ times, i. e., as many times as the

cardinality of the transaction set. We note that each trial is independent and the edges chosen uniformly at random due to the corresponding assumptions concerning the transaction set and the shard creation. We now will show that the average degree of the graph is $\Theta(m)$, which immediately implies the statement of the lemma.

Let the random variable Y_i represent the existence of edge i in the graph, i. e., $Y_i = 1$ if edge i was created at any of the T trials, 0 otherwise. The set of all possible edges in the graph is E, $|E| = \binom{m}{2} = \frac{m(m-1)}{2}$. Note that this is not the same as set E' which includes self-loops and thus $|E'| = \binom{m}{2} + m = \frac{m(m+1)}{2}$. For any vertex u of G, it holds

$$\mathbb{E}[deg(u)] = \frac{2\mathbb{E}[\sum_{\forall i \in E} Y_i]}{m}$$

where $deg(u)$ denotes the degree of node u. We have,

$$Pr[Y_i = 1] = 1 - Pr[Y_i = 0] =$$

$$1 - Pr[Y_i = 0 \text{ at trial 1}]Pr[Y_i = 0 \text{ at trial 2}] \ldots$$

$$Pr[Y_i = 0 \text{ at trial T}] = 1 - \left(1 - \frac{2}{m(m+1)}\right)^{|T|}$$

Thus,

$$\mathbb{E}[deg(u)] = \frac{2m(m-1)}{2}\left[1 - \left(1 - \frac{2}{m(m+1)}\right)^{|T|}\right]$$

$$= (m-1)\left[1 - \left(1 - \frac{2}{m(m+1)}\right)^{|T|}\right]$$

Therefore, for many transactions we have $|T| = \omega(m^2)$ and consequently $\mathbb{E}[deg(u)] = \Theta(m)$. □

Lemma 16. *Given n parties are assigned uniformly at random to shards and the adversary corrupts at most $f = pn$ parties (p constant), all shards are a-honest (a constant) with probability $\frac{1}{n^c}$ only if the number of shards is at most $m = \frac{n}{c' \ln n}$, where $c' \geq c\frac{2+a-p}{(a-p)^2}$.*

Proof. To prove the lemma, it is enough to show that if we only assign the $f = pn$ byzantine parties independently and uniformly at random to the m shards, then all shards will have less than $a\frac{n}{m}$ parties.

From this point on, let n denote the byzantine parties. We will show that for $m = \frac{n}{c' \ln n}$ and a carefully selected constant c', all shards will have less than $(1 + (a-p))\frac{n}{m}$ parties.

Let Y_i denote the size of shard i. Then, $\mathbb{E}[Y_i] = \frac{n}{m}$. By the Chernoff bound, we have

$$Pr\left[Y_i \geq (1 + (a-p))\frac{n}{m}\right] \leq e^{\frac{(a-p)^2}{2+a-p}}\frac{n}{m}$$

The probability that all shards have less than $(1 + (a-p))\frac{n}{m}$ parties is given by the union bound on the number of shards,

$$p = 1 - \left(Pr\left[Y_1 \geq (1 + (a-p))\frac{n}{m}\right] + Pr\left[Y_2 \geq (1 + (a-p))\frac{n}{m}\right] + \cdots + Pr\left[Y_m \geq (1 + (a-p))\frac{n}{m}\right]\right)$$

$$\Rightarrow p \geq 1 - me^{\frac{(a-p)^2}{2+a-p}}\frac{n}{m} \geq 1 - \frac{1}{n^c}, c > 1$$

The last inequality holds for $m \leq \frac{n}{c' \ln n}$, where $c' \geq c\frac{2+a-p}{(a-p)^2}$, in which case all shards are a-honest with high probability $\frac{1}{n^c}$. □

Corollary 17. *A sharding protocol maintains a robust sharded transaction ledger against an adversary with $f = pn$, only if $m = \frac{n}{c' \ln n}$, where $c' \geq c\frac{5/2-p}{(1/2-p)^2}$.*

Proof. To maintain the security properties, all shards must be a-honest, where a depends on the underlying consensus protocol. To the best of our knowledge, all byzantine-fault tolerant consensus protocols require at least honest majority, hence $a \leq 1/2$. From Lemma 16, $m = \frac{n}{c' \ln n}$, where $c' \geq c\frac{5/2-p}{(1/2-p)^2}$. □

C Evaluation of Existing Protocols

In this section, we evaluate existing sharding protocols with respect to the desired properties defined in Section 2.2. A summary of our evaluation can be found in Table 1 in Section 5.

The analysis is conducted in the synchronous model and thus any details regarding performance on periods of asynchrony are discarded. The same holds for other practical refinements that do not asymptotically improve the protocols' performance.

C.1 Elastico

C.1.1 Overview. Elastico is the first distributed blockchain sharding protocol introduced by Luu at al. [29]. The protocol lies in the intersection of traditional BFT protocols and the Nakamoto consensus. The protocol is synchronous and proceeds in epochs. The setting is permissionless, and during each epoch the participants create valid identities by producing proof-of-work (PoW) solutions. The adversary is slowly-adaptive (as described in Section 2) and controls at most $f < \frac{n}{4}$ (n valid identities in total) or 25% of the computational power of the system.

At the beginning of each epoch, participants are partitioned into small shards (committees) of constant size c. The number of shards is $m = 2^s$, where s is a small constant such that $n = c \cdot 2^s$. A shard member contacts its directory committee to identify the other members of the same shard. For each party, the directory committee consists of the first c identities created in the epoch in the party's local view. Transactions are randomly partitioned in disjoint sets based on the hash of the transaction input (in the UTXO model); hence each shard only processes a fraction of the total transactions in the system. The shard members execute a BFT protocol to validate the shard's transactions and then send the validated transactions to the final committee. The final committee consists of all members with a fixed s-bit shards identity, and is in charge of two operations: (i) computing and broadcasting the final block, which is a digital signature on the union of all valid received transactions[6] (via executing a BFT protocol), and (ii) generating and broadcasting a bounded exponential biased random string which is used as a public source of randomness in the next epoch (e. g. for the PoW).

`Consensus:` Elastico does not specify the consensus protocol, but instead can employ any standard BFT protocol, like PBFT [14].

`CrossShard:` Each transaction is assigned to a shard according to the hash of the transaction's inputs. Every party maintains the entire blockchain, thus each shard can validate the assigned transaction independently, i. e., there are no cross-shard transactions. Note that Elastico assumes that transactions have a single input and output, which is not the case in cryptocurrencies as discussed in Section 3. To generalize Elastico's transaction assignment method to multiple inputs, we assume each transaction is assigned to the shard corresponding to the hash of all its inputs. Otherwise, if

[6]The final committee in Elastico broadcasts only the Merkle root for each block. However, this is asymptotically equivalent to including all transactions since the block size is constant. Furthermore, the final committee does not check if the received transactions are conflicting, but merely verifies the presence of signatures.

each input is assigned to a different shard according to its hash value, an additional protocol is required to guarantee the atomicity of transactions and hence the security (consistency) of Elastico.

`Sybil:` Participants create valid identities by producing PoW solutions using the randomness of the previous epoch.

`Divide2Shards & CompactState:` The protocol assigns each identity to a random shard in 2^s, identified by an s-bit shard identity. At the end of each epoch, the final committee broadcasts the final block that contains the Merkle hash root of every block of all shards' block. The final block is stored by all parties in the system. Hence, when the parties are re-assigned to new shards they already have the hash-chain to confirm the shard ledger and future transactions. Essentially, an epoch in Elastico is equivalent to a block generation round.

`DRG:` In each epoch, the final committee (of size c) generates a set of random strings R via a commit-and-XOR protocol. First, all committee members generate an r-bit random string r_i and send the hash $h(r_i)$ to all other committee members. Then, the committee runs an interactive consistency protocol to agree on a single set of hash values S, which they include on the final block. Later, each (honest) committee member broadcasts its random string r_i to all parties in the network. Each party chooses and XORs $c/2 + 1$ random strings for which the corresponding hash exists in S. The output string is the party's randomness for the epoch. Note that $r > 2\lambda + c - \log(c)/2$, where λ is a security parameter.

C.1.2 Analysis. Elastico's threat model allows for adversaries that can drop or modify messages, and send different messages to honest parties, which is not allowed in our model. However, we show that even under a more restrictive adversarial model, Elastico fails to meet the desired sharding properties. Specifically, we prove Elastico does not satisfy *scalability* and *consistency*. From the security analysis of [29], it follows that Elastico satisfies persistence and liveness in our system model.

Theorem 23. *Elastico does not satisfy consistency in our system model.*

Proof. Suppose a party submits two valid transactions, one spending input x and another spending input x and input y. Note that the second is a single transaction with two inputs. In this case, the probability that both hashes (transactions), $H(x, y)$ and $H(x)$, land in the same shard is $1/m$. Hence, the probability of a successful double-spending in a set of T transactions is almost $1 - (1/m)^T$, which converges to 0 as T grows, for any value $m > 1$. However, $m > 1$ is necessary to satisfy scalability (Lemma 10). Therefore, there will be almost surely a round in which two parties report two conflicting transactions. Since the final committee does not verify the validity of transactions but only checks the appropriate signatures are present, consistency is not satisfied. □

Lemma 24. *The space overhead factor of Elastico is $\omega_s = \Theta(n)$.*

Proof. At the end of each epoch, the final committee broadcasts the final block to the entire network. All parties store the final block, hence all parties maintain the entire input set of transactions. Since the block size is considered constant, maintaining the hash-chain for each shard is equivalent to maintaining all the shards' ledgers. It follows that $\omega_s = \Theta(n)$, regardless of the input set T. □

Theorem 25. *Elastico does not satisfy scalability in our system model.*

Proof. Immediately follows from Definition 4 and Lemma 24. □

C.2 Monoxide

C.2.1 Overview. Monoxide [42] is an asynchronous proof-of-work protocol, where the adversary controls at most 50% of the computational power of the system. The protocol uniformly partitions the space of user addresses into shards

(zones) according to the first k bits. Every party is permanently assigned to a shard uniformly at random. Each shard employs the GHOST [38] consensus protocol.

Participants are either full-nodes that verify and maintain the transaction ledgers, or miners investing computational power to solve PoW puzzles for profit in addition to being full-nodes. Monoxide introduces a new mining algorithm, called Chu-ko-nu, that enables miners to mine in parallel for all shards. The Chu-ko-nu algorithm aims to distribute the hashing power to protect individual shards from an adversarial takeover. Successful miners include transactions in blocks. A block in Monoxide is divided into two parts: the chaining block that includes all metadata (Merkle root, nonce for PoW, etc.) creating the hash-chain, and the transaction-block that includes the list of transactions. All parties maintain the hash-chain of every shard in the system.

Furthermore, all parties maintain a distributed hash table for peer discovery and identifying parties in a specific shard. This way the parties of the same shard can identify each other and cross-shard transactions are sent directly to the destination shard. Cross-shard transactions are validated in the shard of the payer and verified from the shard of the payee via a relay transaction and the hash-chain of the payer's shard.

`Consensus:` The consensus protocol of each shard is GHOST [38]. GHOST is a DAG-based consensus protocol similar to Nakamoto consensus [32], but the consensus selection rule is the heaviest subtree instead of the longest chain.

`CrossShard:` An input shard is a shard that corresponds to the address of a sender of a transaction (payer) while an output shard one that corresponds to the address of a receiver of a transaction (payee). Each cross-shard transaction is processed in the input shard, where an additional relay transaction is created and included in a block. The relay transaction consists of all metadata needed to verify the validity of the original transaction by only maintaining the hash-chain of a shard (i. e. for light nodes). The miner of the output shard verifies that the relay transaction is stable and then includes it in a block in the output shard. Note that in case of forks in the input shard, Monoxide invalidates the relay transactions and rewrites the affected transaction ledger to maintain consistency.

`Sybil:` In a typical PoW election scheme, the adversary can create many identities and target its computational power to specific shards to gain control over more than half of the shard's participants. In such a case, the security of the protocol fails (both persistence and consistency properties do not hold). To address this issue, Monoxide introduces a new mining algorithm, Chu-ko-nu, that allows parallel mining on all shards. Specifically, a miner can batch valid transactions from all shards and use the root of the Merkle tree of the list of chaining headers in the batch as input to the hash, alongside with the nonce (and some configuration data). Thus, when a miner successfully computes a hash lower than the target, the miner adds a block to every shard.

`Divide2Shards:` Parties are permanently assigned to shards uniformly at random according the first k bits of their address.

`DRG:` The protocol uses deterministic randomness (e. g. hash function) and does not require any random source.

`CompactState:` No compaction of state is used in Monoxide.

C.2.2 Analysis. We prove that Monoxide satisfies persistence, liveness, and consistency, but does not satisfy scalability. Note that Theorem 15 also implies that Monoxide does not satisfy scalability since Monoxide demands each party to verify cross-shard transactions by acting as a light node to all shards.

Theorem 26. *Monoxide satisfies persistence and liveness in our system model for $f < n/4$.*

Proof. From the analysis of Monoxide, it holds that if all honest miners follow the Chu-ko-nu mining algorithm, then honest majority within each shard holds with high probability for any adversary with $f < n/4$ (Section 5.3 [42]).

Assuming honest majority within shards, persistence depends on two factors: the probability a stable transaction becomes invalid in a shard's ledger, and the probability a cross-shard transaction is reverted after being confirmed. Both these factors solely depend on the common prefix property of the shards' consensus mechanism. Monoxide employs GHOST as the consensus mechanism of each shard, hence the common prefix property is satisfied if we assume that invalidating the relay transaction does not affect other shards [22]. Suppose common prefix is satisfied with probability $1 - p$ (which is overwhelming on the "depth" security parameter k). Then, the probability none of the outputs of a transaction are invalidated is $(1 - p)^{(v-1)}$ (worst case where $v - 1$ outputs – relay transactions – link to one input). Thus, a transaction is valid in a shard's ledger after k blocks with probability $(1 - p)^v$, which is overwhelming in k since v is considered constant. Therefore, persistence is satisfied.

Similarly, liveness is satisfied within each shard. Furthermore, this implies liveness is satisfied for cross-shard transactions. In particular, both the initiative and relay transactions will be eventually included in the shards' transaction ledgers, as long as chain quality and chain growth are guaranteed within each shard [22]. □

Theorem 27. *Monoxide satisfies consistency in our system model for $f < n/4$.*

Proof. The common prefix property is satisfied with high probability in GHOST [23]. Thus, intra-shard transactions satisfy consistency with high probability (on the "depth" security parameter). Furthermore, if a cross-shard transaction output is invalidated after its confirmation, Monoxide allows rewriting the affected transaction ledgers. Hence, consistency is restored in case of cross-transaction failure. Thus, overall, consistency is satisfied in Monoxide. □

Note that allowing to rewrite the transaction ledgers in case a relay transaction is invalidated strengthens the consistency property but weakens the persistence and liveness properties.

Intuitively, to satisfy persistence in a sharded PoW system, the adversarial power needs to be distributed across shards. To that end, Monoxide employs a new mining algorithm, Chu-ko-nu, that incentivizes honest parties to mine in parallel on all shards. However, this implies that a miner needs to verify transactions on all shards and maintain a transaction ledger for all shards. Hence, the verification and space overhead factors are proportional to the number of (honest) participants and the protocol does not satisfy scalability.

Theorem 28. *Monoxide does not satisfy scalability in our system model.*

Proof. Let m denote the number of shards (zones), m_p the fraction of mining power running the Chu-ko-nu mining algorithm and m_d the rest of the mining power ($m_p + m_d = 1$). Additionally, suppose m_s denotes the mining power of one shard. The Chu-ko-nu algorithm enforces the parties to verify transactions that belong to all shards, hence the parties store all sharded ledgers. To satisfy scalability, the space overhead factor of Monoxide can be at most $o(n)$. Thus, at most $o(n)$ parties can run the Chu-ko-nu mining algorithm, hence $nm_p = o(n)$. We note that the adversary will not participate in the Chu-ko-nu mining algorithm as distributing the hashing power is to the adversary's disadvantage.

To satisfy persistence, every shard running the GHOST protocol [38] must satisfy the common prefix property. Thus, the adversary cannot control more than $m_a < m_s/2$ hash power, where $m_s = \frac{m_d}{m} + m_p$. Thus, we have $m_a < \frac{m_s}{2(m_d+m_p)} = \frac{1}{2} - \frac{m_d(m-1)}{2m(m_d+m_p)}$. For n sufficiently large, m_p converges to 0; hence $m_a < \frac{1}{2} - \frac{(m-1)}{2m} = \frac{1}{2m}$. From Lemma 10, $m = \omega(1)$, thus $m_a < 0$ for sufficiently large n. Therefore, Monoxide does not satisfy scalability in our model. □

C.3 OmniLedger

C.3.1 Overview OmniLedger [26] proceeds in epochs, assumes a partially synchronous model within each epoch (to be responsive), synchronous communication channels between honest parties (with a large maximum delay), and a slowly-adaptive computationally-bounded adversary that can corrupt up to $f < n/4$ parties.

The protocol bootstraps using techniques from ByzCoin [24]. The core idea is that there is a global identity blockchain that is extended once per epoch with Sybil resistant proofs (proof-of-work, proof-of-stake, or proof-of-personhood [9]) coupled with public keys. At the beginning of each epoch a sliding window mechanism is employed to define the eligible validators as the ones with identities in the last W blocks, where W depends on the adaptivity of the adversary. For our definition of slowly adaptive, we set $W = 1$. The UTXO space is partitioned uniformly at random into m shards, each shard maintaining its own ledger.

At the beginning of each epoch, a new common random value is created via a distributed randomness generation (DRG) protocol. The DRG protocol employs verifiable random functions (VRF) to elect a leader who runs RandHound [40] to create the random value. The random value is used as a challenge for the next epoch's identity registration and as a seed to assigning identities of the current epoch into shards.

Once the participants for this epoch are assigned to shards and bootstrap their internal states, they start validating transactions and updating the shards' transaction ledgers by operating ByzCoinX, a modification of ByzCoin [24]. When a transaction is cross-shard, a protocol that ensures the atomic operation of transactions across shards called *Atomix* is employed. Atomix is a client-driven atomic commit protocol secure against byzantine adversaries.

`Consensus:` OmniLedger suggests the use of a strongly consistent consensus in order to support Atomix. This modular approach means that any consensus protocol [14, 18, 24, 25, 35] works with OmniLedger as long as the deployment setting of OmniLedger respects the limitations of the consensus protocol. In its experimental deployment, OmniLedger uses a variant of ByzCoin [24] called ByzCoinX [25] in order to maintain the scalability of ByzCoin and be robust as well. We omit the details of ByzCoinX as it is not relevant to our analysis.

`CrossShard (Atomix):` Atomix is a client-based adaptation of two-phase atomic commit protocol running with the assumption that the underlying shards are correct and never crash. This assumption is satisfied because of the random assignment of parties to shards, as well as the byzantine fault-tolerant consensus of each shard.

In particular, Atomix works in two steps: First, the client that wants the transaction to go through requests a proof-of-acceptance or proof-of-rejection from the shards managing the inputs, who log the transactions in their internal blockchain. Afterwards, the client either collects proof-of-acceptance from all the shards or at least one proof-of-rejection. In the first case, the client communicates the proofs to the output shards, who verify the proofs and finish the transaction by generating the necessary UTXOs. In the second case, the client communicates the proofs to the input shards who revert their state and abort the transaction. Atomix, has a subtle replay attack, hence we analyze OmniLedger with the proposed fix [39].

`Sybil:` A global identity blockchain with Sybil resistant proofs coupled with public keys is extended once per epoch.

`Divide2Shards:` Once the parties generate the epoch randomness, the parties can independently compute the shard they are assigned to for this epoch by permuting (*mod* n) the list of validators (available in the identity chain).

`DRG:` The DRG protocol consists of two steps to produce unbiasable randomness. On the first step, all parties evaluate a VRF using their private key and the randomness of the previous round to generate a "lottery ticket". Then the parties

broadcast their ticket and wait for Δ to be sure that they receive the ticket with the lowest value whose generator is elected as the leader of RandHound.

This second step is a partially-synchronous randomness generation protocol, meaning that even in the presence of asynchrony safety is not violated. If the leader is honest, then eventually the parties will output an unbiasable random value, whereas if the leader is dishonest there are no liveness guarantees. To recover from this type of fault the parties can view-change the leader and go back to the first step in order to elect a new leader.

This composition of randomness generation protocols (leader election and multiparty generation) guarantees that all parties agree on the final randomness (due to the view-change) and the protocol remains safe in asynchrony. Furthermore, if the assumed synchrony bound (which can be increasing like PBFT [14]) is correct, an honest leader will be elected in a constant number of rounds.

Note, however, that the DRG protocol is modular, thus any other scalable distributed randomness generation protocol with similar guarantees, such as Hydrand [36] or Scrape [13], can be used.

`CompactState:` A key component that enables OmniLedger to scale is the epoch transition. At the end of every epoch, the parties run consensus on the state changes and append the new state (e. g. UTXO pool) in a state-block that points directly to the previous epoch's state-block. This is a classic technique [14] during reconfiguration events of state machine replication algorithms called checkpointing. New validators do not replay the actual shard's ledger but instead, look only at the checkpoints which help them bootstrap faster.

In order to guarantee the continuous operation of the system, after the parties finish the state commitment process, the shards are reconfigured in small batches (at most 1/3 of the parties in each shard at a time). If there are any blocks committed after the state-block, the validators replay the state-transitions directly.

C.3.2 Analysis. In this section, we prove OmniLedger satisfies persistence, consistency, and scalability (on expectation) but fails to satisfy liveness. Nevertheless, we estimate the efficiency of OmniLedger by providing an upper bound on its throughput factor.

Lemma 29. *At the beginning of each epoch, OmniLedger provides an unbiased, unpredictable, common to all parties random value (with overwhelming probability in t within t rounds).*

Proof. If the elected leader that orchestrates the distributed randomness generation protocol (RandHound or equivalent) is honest the statement holds. On the other hand, if the leader is byzantine, the leader cannot affect the security of the protocol, meaning the leader cannot bias the random value. However, a byzantine leader can delay the process by being unresponsive. We show that there will be an honest leader, hence the protocol will output a random value, with overwhelming probability in the number of rounds t.

The adversary cannot pre-mine PoW puzzles, because the randomness of each epoch is used in the PoW calculation of the next epoch. Hence, the expected number of identities the adversary will control (number of byzantine parties) in the next epoch is $f < n/4$. Hence, the adversary will have the smallest ticket – output of the VRF – and thus will be the leader that orchestrates the distributed randomness generation protocol (RandHound) with probability 1/2. Then, the probability there will be an honest leader in t rounds is $1 - \frac{1}{2^t}$, which is overwhelming in t.

The unpredictability is inherited by the properties of the employed distributed randomness generation protocol. □

Lemma 30. *The distributed randomness generation protocol has $O(\frac{n \log^2 n}{R})$ amortized communication complexity, where R is the number of rounds in an epoch.*

Proof. The DRG protocol inherits the communication complexity of RandHound, which is $O(c^2n)$ [36]. In [40], the authors claim that c is constant. However, the protocol requires a constant fraction of honest parties (e. g. $n/3$) in each of the n/c partitions of size c against an adversary that can corrupt a constant fraction of the total number of parties (e. g. $n/4$). Hence, from Lemma 16, we have $c = \Omega(\log n)$, which leads to communication complexity $O(n \log^2 n)$ for each epoch. Assuming each epoch consist of R rounds, the amortized per round communication complexity is $O(\frac{n \log^2 n}{R})$. □

Corollary 31. *In each epoch, the expected size of each shard is n/m.*

Proof. Due to Lemma 29, the n parties are assigned independently and uniformly at random to m shards. Hence, the expected number of parties in a shard is n/m. □

Lemma 32. *In each epoch, all shards are $\frac{1}{3}$-honest for $m \leq \frac{n}{300c \ln n}$, where c is a security parameter.*

Proof. Due to Lemma 29, the n parties are assigned independently and uniformly at random to m shards. Since $a = 1/3 > p = 1/4$, both a, p constant, the statement holds from Lemma 16 for $m = \frac{n}{c' \ln n}$, where $c' > 300c$. □

Note that the bound is theoretical and holds for a large number of parties since the probability tends to 1 as the number of parties grows. For practical bounds, we refer to OmniLedger's analysis [26].

Theorem 33. *OmniLedger satisfies persistence in our system model for $f < n/4$.*

Proof. From Lemma 32, each shard has an honest supermajority $\frac{2}{3}\frac{n}{m}$ of participants. Hence, persistence holds by the common prefix property of the consensus protocol of each shard. Specifically, for ByzCoinX, persistence holds for depth parameter $k = 1$ because ByzCoinX guarantees finality. □

Theorem 34. *OmniLedger does not satisfy liveness in our system model for $f < n/4$.*

Proof. To estimate the liveness of the protocol, we need to examine all the subprotocols: (i) `Consensus`, (ii) `CrossShard` or Atomix, (iii) `DRG`, (iv) `CompactState`, and (v) `Divide2Shards`.

`Consensus`: From Lemma 32, each shard has an honest supermajority $\frac{2}{3}\frac{n}{m}$ of participants. Hence, in this stage liveness holds by chain growth and chain quality properties of the underlying blockchain protocol (an elaborate proof can be found in [17]). The same holds for `CompactState` as it is executed similarly to `Consensus`.

`CrossShard`: Atomix guarantees liveness since the protocol's efficiency depends on the consensus of each shard involved in the cross-shard transaction. Note that liveness does not depend on the client's behavior; if the appropriate information or some part of the transaction is not provided in multiple rounds to the parties of the protocol then the liveness property does not guarantee the inclusion of the transaction in the ledger. Furthermore, if some other party wants to continue the process it can collect all necessary information from the ledgers of the shards.

`DRG`: During the epoch transition, the DRG protocol provides a common random value with overwhelming probability within t rounds (Lemma 29). Hence, liveness is satisfied in this subprotocol as well.

`Divide2Shrds`: Liveness is not satisfied in this protocol. The reason is that a slowly-adaptive adversary can select who to corrupt during epoch transition, and thus can corrupt a shard from the previous epoch. Since the compact state has not been disseminated in the network, the adversary can simply delete the shard's state. Thereafter, the data unavailability prevents the progress of the system. □

Theorem 35. *OmniLedger satisfies consistency in our system model for $f < n/4$.*

Proof. Each shard is $\frac{1}{3}$-honest (Lemma 32). Hence, consistency holds within each shard, and the adversary cannot successfully double-spend. Nevertheless, we need to guarantee consistency even when transactions are cross-shard. OmniLedger employs Atomix, a protocol which guarantees cross-shard transactions are atomic. Thus, the adversary cannot validate two conflicting transactions across different shards.

Moreover, the adversary cannot revert the chain of a shard and double-spend an input of a cross-shard transaction after the transaction is accepted in all relevant shards because persistence holds (Theorem 33). Suppose persistence holds with probability p. Then, the probability the adversary breaks consistency in a cross-shard transaction is the probability of successfully double-spending in one of the relevant to the transaction shards, $1 - p^v$, where v is the average size of transactions. Since v is constant, consistency holds with high probability, given that persistence holds with high probability. □

To prove OmniLedger satisfies scalability (on expectation) we need to evaluate the scaling factors in the following subprotocols of the system: (i) `Consensus`, (ii) `CrossShard`, (iii) `DRG`, and (iv) `Divide2Shards`. Note that `CompactState` is merely an executon of `Consensus`.

Lemma 36. *The maximum overhead factor of Consensus is $O(n/m)$.*

Proof. From Corollary 31, the expected number of parties in a shard is n/m. ByzCoin has quadratic to the number of parties worst-case communication complexity, hence the communication overhead factor of the protocol is $O(n/m)$. The verification complexity collapses to the communication complexity. The space overhead factor is $O(n/m)$, as each party maintains the ledger of the assigned shard for the epoch. □

Lemma 37. *The maximum overhead factor of Atomix (CrossShard) is $O(v\frac{n}{m})$, where v is the average size of transactions.*

Proof. In a cross-shard transaction, Atomix allows the participants of the output shards to verify the validity of the transaction's inputs without maintaining any information on the input shards' ledgers. This holds due to persistence (Theorem 33).

Furthermore, the verification process requires each input shard to verify the validity of the transaction's inputs and produce a proof-of-acceptance or proof-of-rejection. This corresponds to one query to the verification oracle for each input. In addition, each party of an output shard must verify that all proofs-of-acceptance are present and no shard rejected an input of the cross-shard transaction. The proof-of-acceptance (or rejection) consists of the signature of the shard which is linear to the number of parties in the shard. The relevant parties have to receive all the information related to the transaction from the client (or leader), hence the communication overhead factor is $O(v\frac{n}{m})$.

So far, we considered the communication complexity of Atomix. However, each input must be verified within the corresponding input shard. From Lemma 36, we get that the communication overhead factor at this step is $O(v\frac{n}{m})$. □

Lemma 38. *The maximum overhead factor of Divide2Shards is $O(\frac{n}{mR})$, where R is the number of rounds in an epoch.*

Proof. During the epoch transition each party is assigned to a shard uniformly at random and thus most probably needs to bootstrap to a new shard, meaning the party must store the new shard's ledger. At this point, within each shard OmniLedger introduces checkpoints, the state blocks that summarize the state of the ledger (CompactState). Therefore, when a party syncs with a shard's ledger, it does not download and store the entire ledger but only the active UTXO pool corresponding to the previous epoch's state block.

For security reasons, each party that is reassigned in a new shard must receive the state block of the new shard by $O(n/m)$ parties. Thus, the communication complexity of the protocol is $O(\frac{n}{mR})$ amortized per round, where R is the number of rounds in an epoch.

The space complexity is constant as the state block has constant size, and there is no verification process at this stage. □

Theorem 39. *OmniLedger satisfies scalability in our system model for $f < n/4$ with scaling factor $\Sigma = O(\frac{n}{v \log n})$, where v constant.*

Proof. To evaluate the scalability of OmniLedger, we need to estimate the maximum overhead factor of all the subprotocols of the system: (i) `Consensus`, (ii) `CrossShard`, (iii) `DRG`, and (iv) `Divide2Shards`.

`Consensus` has maximum overhead factor $O(n/m)$ (Lemma 36) while Atomix (`CrossShard`) has expected maximum overhead factor $O(v\frac{n}{m})$ (Lemma 37). As discussed, in Section 3, we assume the average size of transactions to be constant. In this case, the Atomix protocol has expected overhead factor $O(n/m)$.

The epoch transition consists of the `DRG`, `CompactState`, and `Divide2Shards` protocols. `CompactState` has the same overhead with Consensus hence it is not critical. For $R = \Omega(\log n)$ number of rounds per epoch, `DRG` has an expected amortized overhead factor $O(\log n)$ (Lemma 30), while `Divide2Shards` has an expected amortized overhead factor of $O(1)$ (Lemma 38).

Overall, $\Sigma = \Theta(m/v) = O(\frac{n}{v \log n})$, where the last equation holds from Lemma 10 and Lemma 32. □

Theorem 40. *In OmniLedger, the throughput factor is $\sigma = \mu \cdot \tau \cdot \frac{m}{v} < \mu \cdot \tau \cdot \frac{n}{\ln n} \cdot \frac{(a-p)^2}{2+a-p} \cdot \frac{1}{v}$.*

Proof. In Atomix, at most v shards are affected per transaction, thus $m' < m/v$ [7]. From Lemma 16, $m < \frac{n}{\ln n} \cdot \frac{(a-p)^2}{2+a-p}$. Therefore, $\sigma < \mu \cdot \tau \cdot \frac{n}{\ln n} \cdot \frac{(a-p)^2}{2+a-p} \cdot \frac{1}{v}$ □

The parameter v depends on the input transaction set. The parameters μ, τ, a, p depend on the choice of the consensus protocol. Specifically, μ represents the ratio of honest blocks in the chain of a shard. On the other hand, τ depends on the latency of the consensus protocol, i. e., what is the ratio between the propagation time and the block generation time. Last, a expresses the resilience of the consensus protocol (e. g., 1/3 for PBFT), while p the fraction of corrupted parties in the system ($f = pn$).

In OmniLedger, the consensus protocol is modular, so we chose to maintain the parameters for a fairer comparison to other protocols.

C.4 RapidChain

C.4.1 Overview. RapidChain [44] is a synchronous protocol and proceeds in epochs. The adversary is slowly-adaptive, computationally-bounded and corrupts less than 1/3 of the participants ($f < n/3$).

The protocol bootstraps via a committee election protocol that selects $O(\sqrt{n})$ parties – the root group. The root group generates and distributes a sequence of random bits used to establish the reference committee. The reference committee consists of $O(\log n)$ parties, is re-elected at the end of each epoch, and is responsible for: (i) generating the

[7] Note that if v is constant, a more elaborate analysis could yield a lower upper bound on m' better than m/v (depending on D_T). However, if v is not constant but approximates the number of shards m, then m' is also bounded by the scalability of the Atomix protocol (Lemma 37), and thus the throughout factor can be significantly lower.

randomness of the next epoch, (ii) validating the identities of participants for the next epoch from the PoW puzzle, and (iii) reconfiguring the shards from one epoch to the next (to protect against single shard takeover attacks).

The parties are divided into shards of size $O(\log n)$ (committees). Each shard handles a fraction of the transactions, assigned based on the prefix of the transaction ID. Transactions are sent by external users to an arbitrary number of active (for this epoch) parties. The parties then use an inter-shard routing scheme (based on Kademlia [30]) to send the transactions to the input and output shards, i. e., the shards handling the inputs and outputs of a transaction, resp.

To process cross-shard transactions, the leader of the output shard creates an additional transaction for every different input shard. Then the leader sends (via the inter-shard routing scheme) these transactions to the corresponding input shards for validation. To validate transactions (i. e., a block), each shard runs a variant of the synchronous consensus of Ren et al. [35] and thus tolerates $1/2$ byzantine parties.

At the end of each epoch, the shards are reconfigured according to the participants registered in the new reference block. Specifically, RapidChain uses a bounded version of Cuckoo rule [37]; the reconfiguration protocol adds a new party to a shard uniformly at random, and also moves a constant number of parties from each shard and assigns them to other shards uniformly at random.

`Consensus:` In each round, each shard randomly picks a leader. The leader creates a block, gossips the block header H (containing the round and the Merkle root) to the members of the shard, and initiates the consensus protocol on H. The consensus protocol consists of four rounds: (1) The leader gossips $(H, propose)$, (2) All parties gossip the received header $(H, echo)$, (3) The honest parties that received at least two echoes containing a different header gossip $(H', pending)$, where H' contains the null Merkle root and the round, (4) Upon receiving $\frac{nf}{m} + 1$ echos of the same and only header, an honest party gossips $(H, accept)$ along with the received echoes. To increase the transaction throughput, RapidChain allows new leaders to propose new blocks even if the previous block is not yet accepted by all honest parties.

`CrossShard:` For each cross-shard transaction, the leader of the output shard creates one "dummy" transaction for each input UTXO in order to move the transactions' inputs to the output shard, and execute the transaction within the shard. To be specific, assume we have a transaction with two inputs I_1, I_2 and one output O. The leader of the output shard creates three new transactions: tx_1 with input I_1 and output I_1', where I_1' holds the same amount of money with I_1 and belongs to the output shard. tx_2 is created similarly. tx_3 with inputs I_1' and I_2' and output O. Then the leader sends tx_1, tx_2 to the input shards respectively. In principle, the output shard is claiming to be a trusted channel [4] (which is guaranteed from the assignment), hence the input shards should transfer their assets there and then execute the transaction atomically inside the output shard (or abort by returning their assets back to the input shards).

`Sybil:` A party can only participate in an epoch if it solves a PoW puzzle with the previous epoch's randomness, submit the solution to the reference committee, and consequently be included in the next reference block. The reference block contains the active parties' identities for the next epoch, their shard assignment, and the next epoch's randomness, and is broadcast by the reference committee at the end of each epoch.

`Divide2Shards:` During bootstrapping, the parties are partitioned independently and uniformly at random in groups of size $O(\sqrt{n})$ with a deterministic random process. Then, each group runs the DRG protocol and creates a (local) random seed. Every node in the group computes the hash of the random seed and its public key. The e (small constant) smallest tickets are elected from each group and gossiped to the other groups, along with at least half the signatures of the group. These elected parties are the root group. The root group then selects the reference committee

of size $O(\log n)$, which in turn partitions the parties randomly into shards as follows: each party is mapped to a random position in $[0, 1)$ using a hash function. Then, the range $[0, 1)$ is partitioned into k regions, where k is constant. A shard is the group of parties assigned to $O(\log n)$ regions.

During epoch transition, a constant number of parties can join (or leave) the system. This process is handled by the reference committee which determines the next epoch's shard assignment, given the set of active parties for the epoch. The reference committee divides the shards into two groups based on each shard's number of active parties in the previous epoch: group A contains the $m/2$ larger in size shards, while the rest comprise group I. Every new node is assigned uniformly at random to a shard in A. Then, a constant number of parties is evicted from each shard and assigned uniformly at random in a shard in I.

`DRG:` RapidChain uses Feldman's verifiable secret sharing [16] to distributively generate unbiased randomness. At the end of each epoch, the reference committee executes a distributed randomness generation (DRG) protocol to provide the random seed of the next epoch. The same DRG protocol is also executed during bootstrapping to create the root group.

`CompactState:` No protocol for compaction of the state is used.

C.4.2 Analysis. RapidChain does not maintain a robust sharded transaction ledger under our security model since it assumes a weaker adversary. To fairly evaluate the protocol, we weaken our security model. First, assume the adversary cannot change more than a constant number of byzantine parties during an epoch transition, which we term *constant-adaptive adversary*. In general, we assume *bounded epoch transitions*, i. e., at most a constant number of leave/join requests during each transition. Furthermore, the number of epochs is asymptotically less than polynomial to the number of parties. In this weaker security model, we prove RapidChain maintains a robust sharded transaction ledger, and provide an upper bound on the throughput factor of the protocol.

Note that in cross-shard transactions, the "dummy" transactions that are committed in the shards' ledgers as valid, spend UTXOs that are not signed by the corresponding users. Instead, the original transaction, signed by the users, is provided to the shards to verify the validity of the "dummy" transactions. Hence, the transaction validation rules change. Furthermore, the protocol that handles cross-shard transactions has no proof of security against byzantine leaders. For analysis purposes, we assume the following holds:

Assumption 41. *CrossShard satisfies safety even under a byzantine leader (of the output shard).*

Lemma 42. *The maximum overhead factor of DRG is $O(n/m)$.*

Proof. The DRG protocol is executed by the final committee once each epoch. The size of the final committee is $O(n/m) = O(\log n)$. The communication complexity of the DRG protocol is quadratic to the number of parties [16]. Thus, the communication overhead factor is $O(n/m)$. □

Lemma 43. *In each epoch, all shards are $\frac{1}{2}$-honest for $m \leq \frac{n}{78c \ln n}$, where c is a security parameter.*

Proof. During the bootstrapping process of RapidChain (first epoch), the n parties are partitioned independently and uniformly at random into m shards [16]. For $p = 1/3$, the shards are $\frac{1}{2}$-honest only if $m \leq \frac{n}{78c \ln n}$, where c is a security parameter (Lemma 16). At any time during the protocol, all shards remain $\frac{1}{2}$-honest ([44], Theorem 5). Hence, the statement holds after each epoch transition, as long as the number of epochs is $o(n)$. □

Lemma 44. *In each epoch, the expected size of each shard is $O(n/m)$.*

Proof. During the bootstrapping process of RapidChain (first epoch), the n parties are partitioned independently and uniformly at random into m shards [16]. The expected shard size in the first epoch is n/m. Furthermore, during epoch transition the shards remain "balanced" (Theorem 5 [44]), i. e., the size of each shard is $O(n/m)$. □

Theorem 45. *RapidChain satisfies persistence in our system model for constant-adaptive adversaries with $f < n/3$ and bounded epoch transitions.*

Proof. The consensus protocol in RapidChain achieves safety if the shard has no more than $t < 1/2$ fraction of byzantine parties ([44], Theorem 2). Hence, the statement follows from Lemma 43. □

Theorem 46. *RapidChain satisfies liveness in our system model for constant-adaptive adversaries with $f < n/3$ and bounded epoch transitions.*

Proof. To estimate the liveness of RapidChain, we need to examine the following subprotocols: (i) `Consensus`, (ii) `CrossShard`, (iii) `DRG`, and (iv) `Divide2Shards`.

The consensus protocol in RapidChain achieves liveness if the shard has less than $\frac{n}{2m}$ byzantine parties (Theorem 3 [44]). Thus, liveness is guaranteed during `Consensus` (Lemma 43).

Furthermore, the final committee is $\frac{1}{2}$-honest with high probability. Hence, the final committee will route each transaction to the corresponding output shard. We assume transactions will reach all relevant honest parties via a gossip protocol. RapidChain employs IDA-gossip protocol, which guarantees message delivery to all honest parties (Lemma 1 and Lemma 2 [44]). From Assumption 41, the protocol that handles cross-shard transactions satisfies safety even under a byzantine leader. Hence, all "dummy" transactions will be created and eventually delivered. Since the consensus protocol within each shard satisfies liveness, the "dummy" transactions of the input shards will become stable. Consequently, the "dummy" transaction of the output shard will become valid and eventually stable (consensus liveness). Thus, `CrossShard` satisfies liveness.

During epoch transition, `DRG` satisfies liveness [16]. Moreover, `Divide2Shards` allows only for a constant number of leave/join/move operations and thus terminates in a constant number of rounds. □

Theorem 47. *RapidChain satisfies consistency in our system model for constant-adaptive adversaries with $f < n/3$ and bounded epoch transitions.*

Proof. In every epoch, each shard is $\frac{1}{2}$-honest; hence, the adversary cannot double-spend and consistency is satisfied.

Nevertheless, to prove consistency is satisfied across shards, we need to prove that cross-shard transactions are atomic. `CrossShard` in RapidChain ensures that the "dummy" transaction of the output shard becomes valid only if all "dummy" transactions are stable in the input shards. If a "dummy" transaction of an input shard is rejected, the "dummy" transaction of the output shard will not be executed, and all the accepted "dummy" transactions will just transfer the value of the input UTXOs to other UTXOs that belong to the output shard. This holds because the protocol satisfies safety even under a byzantine leader (Assumption 41).

Lastly, the adversary cannot revert the chain of a shard and double-spend an input of the cross-shard transaction after the transaction is accepted in all relevant shards because consistency with each shard and persistence (Theorem 33) hold. Suppose persistence holds with probability p. Then, the probability the adversary breaks consistency in a cross-shard transaction is the probability of successfully double-spending in one of the relevant to the transaction shards, hence

$1 - p^v$ where v is the average size of transactions. Since v is constant, consistency holds with high probability, given persistence holds with high probability. □

Similarly to OmniLedger, to calculate the scaling factor of RapidChain, we need to evaluate the following protocols of the system: (i) `Consensus`, (ii) `CrossShard`, (iii) `DRG`, and (iv) `Divide2Shards`.

Lemma 48. *The maximum overhead factor of Consensus is* $O(\frac{n}{m})$.

Proof. From Lemma 44, the expected number of parties in a shard is $O(n/m)$. The consensus protocol of RapidChain has quadratic to the number of parties communication complexity. Hence, the communication overhead factor `Consensus` is $O(\frac{n}{m})$. The verification complexity collapses to the communication complexity. The space overhead factor is $O(\frac{n}{m})$, as each party maintains the ledger of the assigned shard for the epoch. □

Lemma 49. *The maximum overhead factor of CrossShard is* $O(v\frac{n}{m})$*, where* v *is the average size of transactions.*

Proof. During the execution of the protocol, the interaction between the input and output shards is limited to the leader, who creates and routes the "dummy" transactions. Hence, the communication complexity of the protocol is dominated by the consensus within the shards. For an average size of transactions v, the communication overhead factor is $O(vn/m + v) = O(vn/m)$ (Lemma 44). Note that this bound holds for the worst case, where transactions have $v - 1$ inputs and a single output while all UTXOs belong to different shards.

For each cross-shard transaction, each party of the input and output shards queries the verification oracle once. Hence, the verification overhead factor is $O(vn/m)$. Last, the protocol does not require any verification across shards and thus the only storage requirement for each party is to maintain the ledger of the shard it is assigned to. □

Lemma 50. *The maximum overhead factor of Divide2Shards is* $O(\frac{R \cdot n}{m^2})$.

Proof. The number of join/leave and move operations is constant per epoch, denoted by k. Further, each shard is $\frac{1}{2}$-honest (Lemma 43) and has size $O(\frac{n}{m})$ (Lemma 44); these guarantees hold as long as the number of epochs is $o(n)$.

Each party changing shards receives the new shard's ledger of size T/m by $O(n/m)$ parties in the new shard. Thus the total communication complexity at this stage is $O(\frac{T}{m} \cdot \frac{n}{m})$, hence the communication overhead factor is $O(\frac{T}{m^2}) = O(\frac{R \cdot e}{m^2})$, where R is the number of rounds in each epoch and e the number of epochs since genesis. Since $e = o(n)$, the communication overhead factor is $O(\frac{R \cdot n}{m^2})$.

□

Theorem 51. *RapidChain satisfies scalability in our system model for constant-adaptive adversaries with* $f < n/3$ *and bounded epoch transitions, with expected scaling factor* $\Sigma = \Theta(m) = O(\frac{n}{\log n})$*, for* v *constant and epoch size* $R = O(m)$.

Proof. `Consensus` has expected maximum overhead factor $O(\frac{n}{\log n})$ (Lemma 48), while `CrossShard` has expected expected maximum overhead factor $O(v\frac{n}{\log n})$. (Lemma 49). As discussed, in Section 3, we assume constant average size of transactions, thus the expected maximum overhead factor $O(\frac{n}{\log n})$.

During epoch transitions, `DRG` has expected maximum overhead factor $O(\frac{n}{m})$ (Lemma 42) while `Divide2Shards` has expected overhead factor $O(\frac{n \cdot R}{m^2})$ (Lemma 50). Thus for $R = O(m)$, the maximum overhead factor during epoch transitions is $O(n/m)$.

Overall, RapidChain's expected scaling factor is $\Theta(m) = O(\frac{n}{\log n})$, where the equation holds for $m = \frac{n}{78c \ln n}$ (Lemma 43). □

Theorem 52. *In RapidChain, the throughput factor is* $\sigma = \mu \cdot \tau \cdot \frac{m}{v} < \mu \cdot \tau \cdot \frac{n}{\ln n} \cdot \frac{(a-p)^2}{2+a-p} \cdot \frac{1}{v}$.

Proof. In RapidChain, at most v shards are affected per transaction – when each transaction has $v-1$ inputs and one output, and all belong to different shards. Therefore, $m' < m/v$. From Lemma 16, $m < \frac{n}{\ln n} \cdot \frac{(a-p)^2}{2+a-p}$. Therefore, $\sigma < \mu \cdot \tau \cdot \frac{n}{\ln n} \cdot \frac{(a-p)^2}{2+a-p} \cdot \frac{1}{v}$. □

In RapidChain, the consensus protocol is synchronous and thus not practical. We estimate the throughput factor irrespective of the chosen consensus, to provide a fair comparison to other protocols. We notice that both RapidChain and OmniLedger have the same throughout factor when v is constant.

We provide an example of the throughput factor in case the employed consensus is the one suggested in RapidChain. In this case, we have $a = 1/2$, $p = 1/3$, $\mu = 1/2$ (Theorem 1 [44]), and $\tau = 1/8$ (4 rounds are needed to reach consensus for an honest leader, and the leader will be honest every two rounds on expectation [43].). Note that τ can be improved by allowing the next leader to propose a block even if the previous block is not yet accepted by all honest parties; however, we do not consider this improvement. Hence, for $v = 5$, we have throughput factor

$$\sigma < \frac{1}{2} \cdot \frac{1}{8} \cdot \frac{n}{\ln n} \cdot \frac{(\frac{1}{2} - \frac{1}{3})^2}{2 + \frac{1}{2} - \frac{1}{3}} \cdot \frac{1}{5} = \frac{n}{6240 \ln n}$$

C.5 Chainspace

Chainspace is a sharding protocol introduced by Al-Bassam et al. [2] that operates in the permissioned setting. The main innovation of Chainspace is on the application layer. Specifically, Chainspace presents a sharded, UTXO-based distributed ledger that supports smart contracts. Furthermore, limited privacy is enabled by offloading computation to the clients, who need to only publicly provide zero-knowledge proofs that their computation is correct. Chainspace focuses on specific aspects of sharding; epoch transition or reconfiguration of the protocol is not addressed. Nevertheless, the cross-shard communication protocol, namely S-BAC, is of interest as a building block to secure sharding.

S-BAC protocol. S-BAC is a shard-led cross-shard atomic commit protocol used in Chainspace. In S-BAC, the client submits a transaction to the input shards. Each shard internally runs a BFT protocol to tentatively decide whether to accept or abort the transaction locally and broadcasts its local decision to other shards that take part in the transaction. If the transaction fails locally (e. g., is a double-spend), then the shard generates pre-abort(T), whereas if the transaction succeeds locally the shard generates pre-accept(T) and changes the state of the input to 'locked'. After a shard decides to pre-commit(T), it waits to collect responses from other participating shards, and commits the transaction if all shards respond with pre-accept(T), or aborts the transaction if at least one shard announces pre-abort(T). Once the shards decide, they send their decision (accept(T) or abort(T)) to the client and the output shards. If the decision is accept(T), the output shards generate new 'active' objects and the input shards change the input objects to 'inactive'. If an input shard's decision is abort(T), all input shards unlock the input objects by changing their state to 'active'.

S-BAC, just like Atomix, is susceptible to replay attacks [39]. To address this problem, sequence numbers are added to the transactions, and output shards generate dummy objects during the first phase (pre-commit, pre-abort). More details and security proofs can be found on [39], as well as a hybrid of Atomix and S-BAC called Byzcuit.