Analysing and Improving Shard Allocation Protocols for Sharded Blockchains

Runchao Han*†, Jiangshan Yu*¶, Ren Zhang‡
*Monash University, {runchao.han, jiangshan.yu}@monash.edu
†CSIRO-Data61
‡Nervos Foundation, ren@nervos.org

***Abstract*— Sharding is considered one of the most promising approaches to solve the scalability issue of permissionless blockchains. In a sharding design, participants are split into groups, called shards, and each shard only executes part of the workloads. Despite its wide adoption in permissioned systems, where participants are fixed and known to everyone, transferring such success to permissionless blockchains is challenging, as it is difficult to allocate participants to different shards uniformly. Specifically, in a permissionless network, participants may join and leave the system at any time, and there can be a considerable number of Byzantine participants.**

This paper focuses on the shard allocation protocols designed for permissionless networks. We start from formally defining the shard allocation protocol, including its syntax, correctness properties, and performance metrics. Then, we apply this framework to evaluate the shard allocation subprotocols of seven state-of-the-art sharded blockchains. Our evaluation shows that none of them is fully correct or achieves satisfactory performance. We attribute these deficiencies to their redundant security assumptions and their extreme choices between two performance metrics: *self-balance* and *operability*. We further prove a fundamental trade-off between these two metrics, and prove that shard allocation should be *non-memoryless* in order to parametrise this trade-off. *Non-memorylessness* specifies that each shard allocation does not only rely on the current and the incoming system states, but also previous system states. Based on these insights, we propose WORMHOLE, a *non-memoryless* shard allocation protocol that minimises security assumptions and allows parametrisation between *self-balance* and *operability*. We formally prove WORMHOLE's correctness, and show that WORMHOLE outperforms existing shard allocation protocols.

I. INTRODUCTION

Sharding is a common approach to scale distributed systems. It partitions nodes in a network into some groups, called shards. Nodes in different shards work concurrently, so the system scales horizontally with the increasing number of shards. Sharding has been widely adopted for scaling permissioned systems, in which the set of nodes are fixed and predefined, such as databases [1], file systems [2], and permissioned blockchains [3].

Due to the success of leveraging sharding protocols to scale permissioned systems, sharding is regarded as a promising technique for scaling permissionless blockchains, where anyone can join and leave the system at any time. However, adapting sharding protocols in permissionless settings faces two challenges. First, the focus of traditional sharding protocols [4]–[8] has been on systems concerning crash faults where nodes may stop responding, whereas permissionless blockchains concern Byzantine faults where nodes may behave arbitrarily. Second, as nodes in permissionless blockchains may join or leave at any time, sharding protocols need to dynamically re-balance the number of nodes in different shards. Figure 1 gives an example of the shard allocation process. In this example, five nodes are allocated in shard 3 and four of them left the network. Without rebalancing, the only node in shard 3 will be a single point of failure.

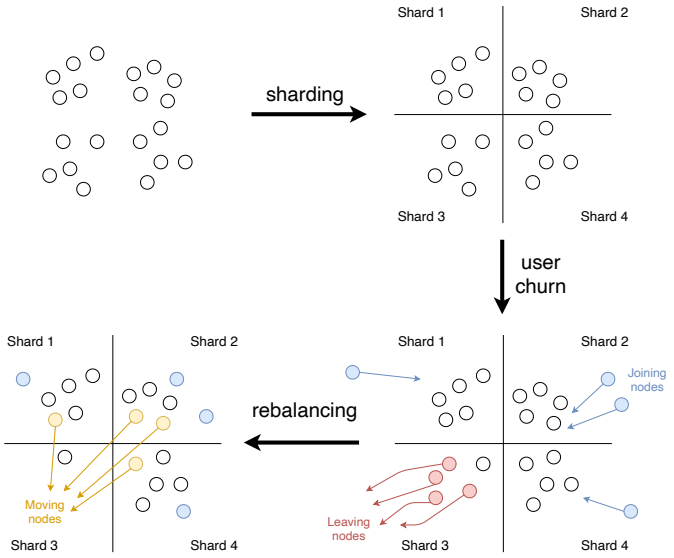

Figure 1: An example of shard allocation. New nodes (in blue) may join the network and existing nodes (in red) may leave the network. After a state update, a subset of nodes (in yellow) may be relocated.

Securely and dynamically allocating nodes into different shards is the core component of sharded blockchains to address the above challenges. While existing works on designing [9]–[15] and analysing [16]–[18] sharded permissionless blockchains studied this core component implicitly, their main focus has been on cross-shard communication and intra-shard consensus. A systematic study on this core component, which we call shard allocation, is still missing.

Contributions. This paper provides the first study on shard

¶ Corresponding author.

allocation— an overlooked core component for shared permissionless blockchains. In particular, we formalise the shard allocation protocol along with its properties and performance metrics, evaluate the shard allocation protocols in the existing leading proposals against the defined criteria, and propose WORMHOLE— a correct and efficient shard allocation protocol for permissionless blockchains. Our contributions are summarised as follows.

1) We provide **the first study on formalising the shard allocation protocol** for permissionless blockchains. We formally define the syntax, correctness properties and performance metrics for shard allocation. This can be used as a framework to assist in designing and analysing permissionless blockchain sharding protocols.
2) Based on our developed framework, we provide **an evaluation of the extracted shard allocation components** of seven state-of-the-art permissionless sharded blockchains, including five academic proposals Elastico [9] (CCS'16), Omniledger [10] (S&P'18), Chainspace [12] (CCS'19), RapidChain [11] (NDSS'19), and Monoxide [13] (NSDI'19), and two industry projects Zilliqa [14] and Ethereum 2.0 [19]. Our results show that *none of these protocols is fully correct or achieves satisfactory performance*.
3) We **observe and prove the impossibility of simultaneously achieving optimal *self-balance* and *operability*.** The former represents the ability to dynamically rebalance the number of nodes in different shards; and the latter represents the system performance w.r.t. the cost of re-allocating nodes to a different shard. While this impossibility has been conjectured by the blockchain community and informally studied [10], we formally prove it's impossible to achieve optimal values on both, and reveal the trade-off between them. Our evaluated shard allocation protocols fall into extreme values on either *self-balance* or *operability*, which may lead to security or performance issues.
4) We **formally prove that to parametrise the trade-off between *self-balance* and *operability*, the shard allocation protocol should be *non-memoryless***. *Non-memorylessness* specifies that each shard allocation does not only rely on the current and the incoming system states, but also previous system states. This opens a new in-between design space and makes the system configurable for different application scenarios.
5) We propose WORMHOLE**, a correct and efficient shard allocation protocol** for permissionless sharded blockchains. WORMHOLE relies on a randomness beacon and lightweight cryptographic primitives such as Verifiable Random Functions (VRFs), achieves all correctness properties and introduces negligible communication overhead. In addition, by being *non-memoryless*, WORMHOLE supports parametrisation of the trade-off between *self-balance* and *operability*.

Scope. This paper only focuses on the shard allocation protocol. Sharded blockchains usually combine shard allocation with other protocols such as consensus and cross-shard communication. When analysing shard allocation, we assume other protocols in sharded blockchains are secure. We consider analysis of integrating shard allocation into sharded blockchains as future work.

Table I: Summary of notations.

Symbol	Description
st_t	System state at round t
m	Number of shards
n^t	Number of nodes in the network at round t
n_k^t	Number of nodes in shard k at round t
α_t, β_t	Average percentage of nodes joining and leaving the system at round t, respectively. ($\beta_t \in [0,1]$)
pp	Public parameter
sk_i, pk_i	Secret key and public key of node i
$\pi_{i,st_t,k}$	Proof of assigning node i to shard k at round t
γ	Probability of a node to stay at its shard after $\mathsf{UpdateShard}(\cdot)$ ($\gamma \in [0,1)$)

Paper organisation. Section II defines syntax, correctness and performance metrics of shard allocation protocols. Section III outlines the evaluation results on existing shard allocation protocols, leading to our insights in Section IV. Section V describes our shard allocation protocol WORMHOLE. Section VI presents related work before we conclude in Section VII. Appendix A provides details of our evaluated shard allocation protocols.

II. FORMALISING SHARD ALLOCATION

A. Definition

System setting and notations. We consider a permissionless system with networked nodes. The system executes in rounds and st_t denotes the system state at round t. We assume a total order on the system states. At any state, each node only belongs to a single shard. Let $k \in [1, m]$ be the unique identity of a shard and n_k^t be the number of nodes in shard k at round t. The total number of nodes in the system at round t is $n^t = \sum_{k=1}^{m} n_k^t$. Each node i has a pair of secret key sk_i and public key pk_i, and is identified by its public key in the system. We consider user churn [20]: existing nodes may leave and new nodes may join the system at any time. Let α_t and β_t be the average percentage of nodes joining and leaving the network at round t, respectively, where $\beta \leq 1$. That is, for two consecutive states st_t and st_{t+1}, $n^{t+1} = (1 + \alpha_t - \beta_t)n^t$. Let $F(in) \rightarrow out$ denote the assignment of the output of $F(in)$ to out. Let $\mathsf{MSB}(z, str)$ and $\mathsf{LSB}(z, str)$ be the z most significant bits and least significant bits of str, respectively. Table I summarises notations in this paper.

Syntax. We formally define shard allocation as follows.

Definition 1 (shard allocation)**.** *A shard allocation protocol $\mathcal{SA}$ is a tuple of polynomial time algorithms*

$$\mathcal{SA} = (\mathsf{Setup}, \mathsf{JoinShard}, \mathsf{VerifyShard}, \mathsf{UpdateShard})$$

$\mathcal{SA}.\mathsf{Setup}(\lambda) \rightarrow (pp)$ *: On input the security parameter λ, outputs the public parameter pp.*

$\mathcal{SA}.\mathsf{JoinShard}(sk_i, pp, st_t) \rightarrow (k, \pi_{i,st_t,k})$ *: On input a secret key sk_i, the public parameter pp and state st_t, outputs the ID k of the shard assigned for node i, the proof $\pi_{i,st_t,k}$ of assigning i to k at st_t.*

$\mathcal{SA}.\mathsf{UpdateShard}(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1}) \rightarrow (k', \pi_{i,st_{t+1},k'}$ *: On input a secret key sk_i, the public parameter pp, state st_t, the shard index k, proof $\pi_{i,st_t,k}$ and the next state st_{t+1}, outputs the identity k' of the newly assigned shard for i, a proof of shard assignment $\pi_{i,st_{t+1},k'}$.*

$\mathcal{SA}.\mathsf{VerifyShard}(pp, pk_i, st_t, k, \pi_{i,st_t,k}) \rightarrow 0$ *or* 1 *: Deterministic. On input the public parameter pp, i's public key pk_i, a system state st_t, the shard index k and a proof of shard assignment $\pi_{i,st_t,k}$, outputs* 0 *(false) or* 1 *(true).*

When node i joins the system at state st_t, node i is allocated to a shard according to the output of executing $\mathcal{SA}.\mathsf{JoinShard}(\cdot)$. It updates its shard allocation when the state changes from st to st' by executing $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$. Given a shard k, public key pk_i and a proof $\pi_{i,st_t,k}$, anyone can execute $\mathcal{SA}.\mathsf{VerifyShard}$ to verify whether node i is allocated into shard k at state st_t. Algorithm 1 describes the typical process of shard allocation.

Algorithm 1: The shard allocation process for node i.

```
// Join the system at state st_t
(k_t, π_{i,st_t,k_t}) ← SA.JoinShard(sk_i, pp, st_t)
// Assign last state and shard
st_now, k_now, π_now ← st_t, k_t, π_{i,st_t,k_t}
repeat
    Wait for a new state st_*
    // Update shard allocation
    (k_*, π_{i,st_*,k_*}) ←
      SA.UpdateShard(sk_i, pp, st_now, k_now, π_{i,st_now,k_now} st_*)
    // Reassign last state and shard
    st_now, k_now, π_now ← st_*, k_*, π_{i,st_*,k_*}
until node i leaves the system
```

B. Correctness

We consider three correctness properties for shard allocation protocols, namely *liveness*, *allocation-randomness*, and *unbiasibility*. We additionally consider *allocation-privacy* as an optional property.

Liveness. A shard allocation protocol is live when nodes are able to make progress in updating their shard membership. This relies on the liveness of the underlying system, i.e., the underlying system should always make progress in updating the system state. We adapt the liveness definition by Garay et al. [21].

Definition 2 (Liveness)**.** *Parametrised by a growth factor $\tau \in \mathbb{R}_{>0}$ and $s \in \mathbb{N}_{>0}$. shard allocation $\mathcal{SA}$ satisfies* (τ, s)-liveness *iff there are at least $\lfloor \tau \cdot s \rfloor$ new states for any s rounds.*

Allocation-randomness. To remain correct, each blockchain assumes a certain threshold of Byzantine nodes (or power) in the system [22]. If a large portion of Byzantine nodes is allocated into the same shard, the safety or liveness of this shard may be broken. To avoid this from happening, each new node should be allocated into a shard randomly [9], [10], [14]. We consider two parts of allocation-randomness, namely *join-randomness* and *update-randomness*. *Join-randomness* specifies that the newly joined nodes join each shard with equal probability. *Update-randomness* specifies the probability distribution of existing nodes' shard allocation upon new system states. We formally define them as follows.

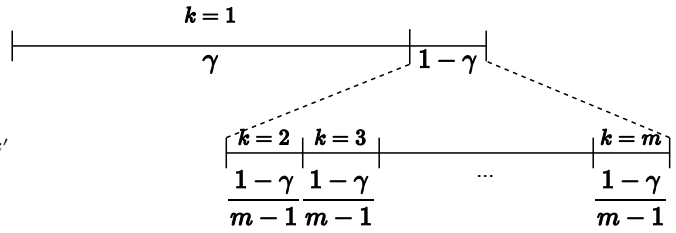

Figure 2: *Update-randomness*. After executing $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$, the probability that a node stays in its shard (say shard 1) is γ, and the probability of moving to each other shard is $\frac{1-\gamma}{m-1}$.

Definition 3 (Join-randomness)**.** *A shard allocation protocol $\mathcal{SA}$ with m shards satisfies* join-randomness *iff for any secret key sk_i, public parameter pp and state st_t, the probability of node i joining a shard k is*

$$Pr\left[k = k' \middle| \begin{array}{c}(k', \pi_{i,st_t,k'}) \leftarrow \\ \mathcal{SA}.\mathsf{JoinShard}(sk_i, pp, st_t)\end{array}\right] = \frac{1}{m} \pm \epsilon$$

where $k, k' \in [1, m]$, and ϵ is a negligible value.

For each update of the system state, the shard allocation of existing nodes may need to be updated to re-balance the number of nodes in different shards. However, when a node moves from one shard to the other, it needs to perform several operations including identifying peers in the new shard and synchronise the knowledge with the new shard. These operations introduce significant overhead and may dramatically decrease the performance of the protocol [10], [23]–[25]. Thus, only a small subset of existing nodes should be relocated for each state update. During a state update, we consider that an existing node stays in the same shard with probability γ. Similar to the concern discussed in the *join-randomness*, we define *update-randomness* as follows.

Definition 4 (Update-randomness)**.** *A shard allocation protocol $\mathcal{SA}$ with m shards satisfies* update-randomness *iff there exists $\gamma \in [0, 1)$ such that for any $k \in [1, m]$, secret key sk_i and public parameter pp, the probability of node i updates its shard from k at state st_t to k'' at state st_{t+1} is*

$$Pr\left[k'' = k' \middle| \begin{array}{c}(k', \pi_{i,st_{t+1},k'}) \leftarrow \\ \mathcal{SA}.\mathsf{UpdateShard}(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1})\end{array}\right] = \begin{cases}\gamma \pm \epsilon & \text{if } k' = k \\ \frac{1-\gamma}{m-1} \pm \epsilon & \text{otherwise}\end{cases}$$

where $k', k'' \in [1, m]$, and ϵ is a negligible value.

When $\gamma = \frac{1}{m}$, $\mathcal{SA}$ achieves optimal *update-randomness*. Figure 2 represents an intuition behind the definition.

Definition 5 (Allocation-randomness)**.** *A shard allocation protocol satisfies* allocation-randomness *if it satisfies* join-randomness *and* update-randomness*.*

Unbiasibility. The *unbiasibility* represents the ability of the system preventing a node from manipulating the probability of allocating it to a shard. While *allocation-randomness* defines the probability distribution of shard allocation, *unbiasibility* considers the possibility of Byzantine nodes manipulating the protocol.

Definition 6 (Unbiasibility)**.** *A shard allocation protocol $\mathcal{SA}$ satisfies* unbiasibility *iff given a system state, no node can manipulate the probability distribution of the resulting shard of $\mathcal{SA}.\mathsf{JoinShard}(\cdot)$ or $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$, except with negligible probability.*

Allocation-privacy. We define *allocation-privacy* as an optional property to protect the output of shard allocation for a node before the node reveals the associated proofs. *Allocation-privacy* may prevent certain attacks from a Byzantine node on a target node or shard. *Allocation-privacy* may sacrifice a shard allocation protocol's performance, as a dedicated protocol for finding peers of the same shard may be required [9], [14].

Definition 7 (Join-privacy)**.** *A shard allocation protocol $\mathcal{SA}$ with m shards provides* join-privacy *iff for any secret key sk_i, public parameter pp, and state st_t, without the knowledge of $\pi_{i,st_t,k}$ and sk_i, the probability of making a correct guess k' on k is*

$$Pr\left[k' = k \middle| \begin{array}{c}(k, \pi_{i,st_t,k}) \leftarrow \\ \mathcal{SA}.\mathsf{JoinShard}(sk_i, pp, st_t)\end{array}\right] = \frac{1}{m} \pm \epsilon$$

where $k, k' \in [1, m]$, and ϵ is a negligible value.

Definition 8 (Update-privacy)**.** *A shard allocation protocol $\mathcal{SA}$ with m shards provides* update-privacy *iff for some $\gamma \in [0, 1)$, any $k \in [1, m]$, secret key sk_i, public parameter pp, and two consecutive states st_t and st_{t+1}, without the knowledge of $\pi_{i,st_{t+1},k'}$ and sk_i, the probability of making a correct guess k'' on k' is*

$$Pr\left[k'' = k' \middle| \begin{array}{c}(k', \pi_{i,st_{t+1},k'}) \leftarrow \\ \mathcal{SA}.\mathsf{UpdateShard}(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1})\end{array}\right] = \begin{cases}\gamma \pm \epsilon & \text{if } k'' = k \\ \frac{1-\gamma}{m-1} \pm \epsilon & \text{otherwise}\end{cases}$$

where $k', k'' \in [1, m]$, and ϵ is a negligible value.

Definition 9 (Allocation-privacy)**.** *A shard allocation protocol $\mathcal{SA}$ satisfies* allocation-privacy *iff it satisfies both* join-privacy *and* update-privacy*.*

C. Performance metrics

We consider four performance metrics, namely *communication complexity*, *Sybil cost*, *self-balance*, and *operability*.

Communication complexity. It is the amount of communication required to complete a protocol [26]. For a shard allocation protocol, we consider the *communication complexity* of a node joining a shard and recomputing its shard when the system state is updated. This does not include communication of synchronising shards.

Sybil cost. shard allocation protocols should prevent Sybil attacks [27] where an adversary pretends to be numerous nodes and spams the network. To mitigate Sybil attacks, joining the system should have a non-negligible cost. The cost can be of diverse types, such as computing power introduced by the Proof-of-Work [28] and cryptocurrency deposit introduced by Proof-of-Stake [29].

Self-balance. To maximise the fault tolerance level of shards, nodes should be uniformly distributed among shards. Otherwise, the fault tolerance threshold of shards with fewer nodes and the performance of shards with more nodes may be reduced [13], [30]. Due to user churn and lack of global view, reaching global load balance is impossible for permissionless networks. Instead, the randomised self-balance approach — where a subset of nodes move to other shards randomly — provides the optimal load balance guarantee. We quantify the *self-balance* as the ability that a shard allocation protocol recovers from load imbalance.

Definition 10 (Self-balance)**.** *A shard allocation protocol with m shards is* ω-self-balanced *iff for the largest possible $\omega \in [0, 1]$, we have*

$$\omega \leq 1 - \frac{|n_i^{t+1} - n_j^{t+1}|}{n^t}, \quad \forall i, j \in [1, m]$$

When $\omega = 1$, the shard allocation protocol achieves optimal *self-balance*, where the system can balance itself within one round, regardless of the number of nodes joining or leaving the system during the last round.

Operability. To balance the number of nodes in different shards, in each round the shard allocation protocol relocates some nodes from one shard to the other. When a node moves to another shard, it should synchronise with the blockchain of that shard, which introduces significant overhead and can make the node unavailable for a long time[23]–[25]. The *operability* was introduced to measure the cost of relocating nodes [10]. We refine this notion by defining the *operability* Γ as the probability of existing nodes staying at the same shard when the system state is updated. Following our definition on *update-randomness* (Definition 4), if a shard allocation protocol satisfies update-randomness with γ, then its operability is $\Gamma = \gamma$, i.e., γ-*operable*.

When $\Gamma = 1$, the shard allocation protocol is most *operable*, i.e., nodes will never move after joining the network. However, in this case shards cannot keep balance in the presence of user churn. Later in §IV, we will formally prove a trade-off between *self-balance* and *operability* that, no correct shard allocation protocol can achieve both optimal *self-balance* of 1 and optimal *operability* of 1 simultaneously, and discuss how to parametrise this trade-off.

III. Evaluating existing shard allocation protocols

In this section, we evaluate the shard allocation protocols of seven state-of-the-art sharded blockchains. Our evaluation (summarised in Table II) shows that none of these shard allocation protocols achieves all correctness properties, and some of the protocols have high complexity and/or low operability.

A. Evaluation criteria

For our evaluation, we assume other components of these sharded blockchains are fully correct. Our evaluation considers three aspects, namely system models, correctness properties, and performance metrics. We consider the standard system model common to all sharded blockchains [9], [10], [14], [22]. Section II-B and II-C define correctness properties and performance metrics, respectively.

The system model considers three aspects, namely network models, trust assumptions, and fault tolerance capacity. Network model describes the timing guarantee of message deliveries. There are three common types of network models, namely synchronous, partially synchronous, and asynchronous [31]. A network is synchronous if messages will be delivered within a known finite time-bound; is partially synchronous if messages will be delivered within an unknown finite time-bound; or is asynchronous if messages will be delivered eventually but without a finite time-bound. Trust assumption indicates the trustworthy components that the protocol should assume to remain correct. Fault tolerance capacity indicates the percentage of voting power controlled by Byzantine nodes that the system tolerates. If each node has one vote in the blockchain protocol, the voting power is quantified by the number of nodes. If a node's power is determined by its computing power (e.g., in PoW-based consensus), the voting power is quantified by the mining power of nodes. If a node's power is determined by its balance or deposit (e.g., Proof-of-Stake-based consensus [29]), the voting power is quantified by the stake of nodes.

B. Overview of evaluated proposals

We choose seven state-of-the-art sharded blockchains, including five academic proposals Elastico [9], Omniledger [10], Chainspace [12], RapidChain [11], and Monoxide [13], and two industry projects Zilliqa [14] and Ethereum 2.0 (ETH 2.0) [19]. We briefly describe their shard allocation components and defer details to Appendix A.

Elastico, Omniledger, RapidChain, and Zilliqa rely on distributed randomness generation (DRG) protocols for shard allocation. Elstico [9] is the first protocol that introduces the idea of blockchain sharding. In Elastico, nodes in a special shard called *final committee* jointly run a commit-and-reveal DRG protocol [32] to produce a random output, then each node locally solves a PoW puzzle with this random output as input. Each node will be assigned to a shard according to the least significant bits (LSBs) of its PoW solution. Zilliqa [14] is built upon Elastico, with several optimisations. In particular, Zilliqa uses the hash of the last block as the randomness, rather than running a DRG protocol. Omniledger [10] uses *RandHound* as its DRG protocol to generate randomness. *RandHound* has a better communication complexity than the one [32] used by Elastico. Omniledger additionally relies on a trusted identity authority to allocate nodes to different shards. The identity authority knows all nodes in the network. Given the latest randomness, the identity authority samples a subset of pending nodes that are allowed to join the system and shuffles all existing nodes in the network to different shards.

In RapidChain [11], each node should solve an offline PoW puzzle before joining the system. To prevent pre-computing PoW puzzles, RapidChain generates randomness periodically using a DRG protocol based on Feldman VSS [33], and uses the random output as a part of the input of PoW puzzles. RapidChain uses a special shard called *reference committee* to allocate nodes into different shards. The reference committee assigns nodes into different shards using the Commensal Cuckoo rule [34], which assures the load balance of shards under churn. Commensal Cuckoo maps each node to a real-number ID on the interval $[0, 1)$. When a new node joins the system with generated real number x, the reference committee moves each of the nodes with ID close to x to a random shard. Chainspace [12] is a sharded smart contract system. Nodes can apply to move to other shards at any time, and whether to approve such requests depends on the voting results of nodes. The voting works over a special smart contract called `ManageShards`, which is assumed secure by Chainspace. Monoxide [13] and Ethereum 2.0 [15] simply allocate nodes into different shards according to prefixes of their addresses.

C. System model

Network model. Elastico and RapidChain assume synchronous network models, as they rely on DRG protocols [32], [33]. Omniledger assumes partially synchronous networks. In Omniledger, nodes start from running *RandHound*, which assumes partially synchronous networks. If *RandHound* fails for five times, nodes will instead run the asynchronous DRG [35]. All other proposals assume asynchronous networks. Chainspace assigns nodes into different shards using transactions on smart contracts, and transactions are committed to the ledger asynchronously. Monoxide and ETH 2.0 assign nodes into different shards simply by most significant bits (MSBs) of addresses. Zilliqa replaces the DRG by using block hashes, so no longer requires synchronous networks.

Trust assumption. Omniledger and Chainspace rely on a trusted identity blockchain and smart contracts for shard allocation, respectively. Other protocols assume no trustworthy components.

Fault tolerance capacity. Elastico and Omniledger achieve the fault tolerance level of $\frac{1}{3}$, which is inherited from their DRG protocols. RapidChain cannot tolerate any faults, as one faulty node can make the Feldman VSS-based DRG lose liveness. Chainspace, Monoxide, Zilliqa, and ETH 2.0 can tolerate any fraction of adversaries. For Monoxide and ETH 2.0, computing shards is offline. Chainspace assumes

Table II: Evaluation of seven permissionless shard allocation protocols. Red indicates strong assumptions, unsatisfied correctness properties, and relatively poor performance. Yellow indicates unspecified assumptions, partly satisfied correctness properties, and unspecified performance metrics. Green indicates weak assumptions, satisfied correctness properties, and better performance.

		System model			Correctness					Performance metrics				
	State update	Network model	Trust assumption	Fault tolerance	Public verifiability	Liveness	Allocation-rand.	Unbiasibility	Privacy^o	Join comm. compl.	Update comm. compl.	Sybil cost	Self-balance	Operability
Elastico	New block	Sync.	-	$\frac{1}{3}$	✓	✓	✓	✗*	✓	$O(n^f)$	$O(n^f)$	Comp.	1	$\frac{1}{m}$
Omniledger	Identity authority	Part. sync.	Identity authority	$\frac{1}{3}$	✓	✓	✓	✓	✗	$O(1)$	$O(n) \sim O(n^3)$	-	1	$\frac{1}{m}$
RapidChain	Nodes joining	Sync.	-	0	✗	✓	✓	✓	✗	$O(n^2)$	$O(n^2)$	Comp.	$1-\beta_t$	$max(1-\kappa\alpha_t n, 0)$
Chainspace	-	Async.	Smart contracts	1	✓	✓	✗	✗	✗	-	$O(1)$	-	$1-\beta_t$	-
Monoxide	-	Async.	-	1	✓	✓	✗	✓	✗	0	0	No**	$1-\beta_t$	1
Zilliqa	New block	Async.	-	1	✓	✓	✓	✗*	✓	0	0	Comp.	1	$\frac{1}{m}$
ETH 2.0	-	Async.	-	1	✓	✓	✗	✓	✗	0	0	No**	$1-\beta_t$	1
WORMHOLE (Our proposal in §V)	New rand.	Async.	Rand. Beacon	1	✓	✓	✓	✓	✓	0	0	Comp.	$1-\beta_t+\frac{\beta_t}{2^{op}}$	$1-\frac{m-1}{m\cdot 2^{op}}$

^{o} Optional. * Protected by PoW puzzles. ** Protected by Sybil-resistant consensus protocols.

trustworthy smart contracts. For Zilliqa, we assume blocks are produced correctly and shard computation is offline.

D. Correctness

Public verifiability. All of these shard allocation protocols achieve public verifiability except for RapidChain. RapidChain's shard allocation is not publicly verifiable, as the deployed Commensal Cuckoo protocol is not publicly verifiable. Elastico and Zilliqa achieve public verifiability by using PoW puzzles; Omniledger achieves public verifiability as the identity blockchain is operated publicly; Chainspace achieves public verifiability using smart contracts; Monoxide and ETH 2.0 achieve public verifiability as they simply use MSBs of addresses to shard nodes.

Liveness. As papers describing these sharded blockchains do not specify parameters, we cannot determine the actual τ and s for their liveness. Thus, we consider a shard allocation protocol does not satisfy liveness when there exists an attack that can stop the protocol from producing new system states. All shard allocation protocols satisfy liveness except for Omniledger. The leader election of *RandHound* may lose liveness in an asynchronous network. In asynchronous networks, two subsets of nodes A and B may agree on two different smallest tickets and nominate two different leaders a and b. As Omniledger chooses safety over liveness, nodes in A will discard all messages from b, and vice versa. This leads to a scenario that neither *RandHound* of A nor B will terminate. In addition, the Collective Signing (CoSi) protocol [36] used in *RandHound* may lose liveness under a single Byzantine node [37].

Allocation-randomness. Elastico, Omniledger, RapidChain, and Zilliqa satisfy allocation-randomness, as for each new round all nodes move to other shards randomly. Chainspace satisfies neither join-randomness nor update-randomness, as nodes can choose which shard to join. Monoxide and ETH 2.0 also satisfy neither of them, as nodes determine their shards using prefixes of addresses and never move among shards.

Unbiasibility. Elastico and Zilliqa do not fully achieve *unbiasibility*. Compared to the PoW puzzles in Bitcoin-like systems, the PoW puzzles in Elastico and Zilliqa are relatively easy to solve and each node in Elastico and Zilliqa solves the PoW puzzle at least once within each round. This allows nodes to solve multiple puzzles within a round to be allocated to a preferred shard. Chainspace does not achieve *unbiasibility*, as it does not satisfy *allocation-randomness* and nodes are free to choose shards. Omniledger, RapidChain, Monoxide and ETH 2.0 satisfy *unbiasibility*.

Allocation-privacy. Only Elastico and Zilliqa satisfy *allocation-privacy*. Given a new random number output, each node should find a valid PoW solution, which is probabilistic. Thus, without revealing the PoW solution, other nodes cannot know which shard a node belongs to. Therefore, Elastico and Zilliqa require an extra peer finding step. Elastico and Zilliqa call this step "overlay setup", and a special shard called "directory committee" is responsible for helping nodes to find their peers. Omniledger, RapidChain, and Chainspace do not satisfy *allocation-privacy* as memberships can be queried at the identity blockchain, the reference committee and the `ManageShards` smart contract, respectively. Monoxide and ETH 2.0 do not satisfy *allocation-privacy*, as nodes determine their shards using their addresses, which are publicly known.

E. Performance

Communication complexity. The communication complexity of Elastico's $\mathsf{JoinShard}(\cdot)$ and $\mathsf{UpdateShard}(\cdot)$ is $O(n^f)$, where n and f are the number of nodes and faulty nodes in the network, respectively. In $\mathsf{JoinShard}(\cdot)$ and $\mathsf{UpdateShard}(\cdot)$, the final committee runs the DRG protocol to produce a random output, then each node locally finds a valid PoW solution with the random output as input. The DRG protocol is bottlenecked by the vector consensus [38], whose communication complexity is $O(n^f)$. Ideally, the final committee consists of $\frac{n}{m}$ nodes, so the communication complexity of $\mathsf{JoinShard}(\cdot)$

and $\mathsf{UpdateShard}(\cdot)$ is $O(\frac{n}{m}\frac{f}{m}) = O(\frac{n}{m}\frac{f}{m}) = O(n^f)$ (m is constant). The communication complexity of Omniledger's $\mathsf{JoinShard}(\cdot)$ and $\mathsf{UpdateShard}(\cdot)$ is $O(1)$ and $O(n) \sim O(n^3)$, respectively. In Omniledger, a node communicates with the identity authority when joining shards, and runs a DRG protocol when updating shards. The best case of Omniledger's $\mathsf{UpdateShard}(\cdot)$ is that a leader is elected with a VRF sortition, then follows a successful run of *RandHound*, whose communication complexity is $O(n)$. The worst case is that, after failing leader elections five times, nodes have to run the asynchronous DRG [35], whose communication complexity is $O(n^3)$. The communication complexity of RapidChain's $\mathsf{JoinShard}(\cdot)$ and $\mathsf{UpdateShard}(\cdot)$ is $O(n^2)$. This is from Feldman VSS, which has the communication complexity of $O(\frac{n}{m}^2) = O(n^2)$ [33]. With Monoxide and Zilliqa, to join the system or update its allocated shard, a node only needs to solve a PoW puzzle offline, therefore there is no communication cost.

Sybil cost. The Sybil cost of Elastico, RapidChain, and Zilliqa is performing computational work to solve a PoW puzzle for joining a shard. Monoxide and ETH 2.0 do not aim at addressing Sybil attacks within the shard allocation component. Instead, they address Sybil attacks using Sybil-resistant consensus protocols. Omniledger relies on the identity authority to issue Sybil-resistant memberships but does not provide details on how to issue them. Similarly, Chainspace also does not provide details on how to issue memberships.

Self-balance. The *self-balance* of Elastico, Omniledger, Monoxide, Zilliqa and ETH 2.0 is 1. For Elastico, Omniledger, and Zilliqa, all nodes are shuffled after each round. The *self-balance* of RapidChain, Chainspace, Monoxide and ETH 2.0 is $1 - \beta_t$. When $\Gamma = 1$, no nodes will newly join the system, and no node will move to other shards. Thus, *self-balance* is $\frac{n-\beta_t n}{n} = 1 - \beta_t$.

Operability. The *operability* of Elastico, Omniledger and Zilliqa are $\frac{1}{m}$, as all nodes are forced to change their shards for each new round. The *operability* of RapidChain is $\max(1 - \kappa\alpha_t n, 0)$, where $\kappa \in [0, 1]$ is the size of the interval in which nodes should move to other shards, and α_t is the percentage of nodes joining the network at round t. Consider there are $\alpha_t n$ nodes joining the network. Each newly joined node causes the reallocation of κn other nodes. The *operability* then becomes $1 - \frac{\alpha_t n \cdot \kappa n}{n} = 1 - \kappa\alpha_t n$. However, in reality the *operability* cannot be smaller than 0. Thus, the *operability* is $\max(1 - \kappa\alpha_t n, 0)$. We cannot determine the *operability* of Chainspace, as Chainspace does not specify how many nodes can propose to change their shards. Monoxide and ETH 2.0 have the *operability* of 1, as nodes in Monoxide and ETH 2.0 never move to other shards.

IV. Observation and insights

We observe from Table II that the evaluated shard allocation protocols cannot simultaneously achieve optimal *self-balance* and *operability*. In fact, the *operability* is either 1 or $\frac{1}{m}$ (except for RapidChain), and *self-balance* is either $1 - \beta_t$ or 1, where value "1" is the optimal value. We formally analyse this observation, and prove that it is impossible to achieve optimal *operability* and *self-balance* simultaneously. We then observe a property, called *non-memoryless*, that is necessary for parametrising the trade-off between *operability* and *self-balance*.

Impossibility. Intuitively, optimal *self-balance* requires all nodes to be relocated at each new state, so that the number of nodes can be distributed evenly into different shards. Meanwhile, optimal *operability* requires all nodes to stay within the same shard across upon a new state, as this saves the cost of relocating nodes. We study their relation and prove that a correct shard allocation protocol cannot achieve both optimal *self-balance* and *operability*. We first analyse the relationship between *self-balance* and γ (per Definition 4). Then we show that unless $\beta_t = 0$, i.e., no node leaves the system, optimal *self-balance* and *operability* cannot be achieved simultaneously.

Lemma 1. *If a correct shard allocation protocol $\mathcal{SA}$ with m shards satisfies update-randomness with γ, the self-balance ω of $\mathcal{SA}$ is*

$$\omega = 1 - \left| \frac{(\gamma m - 1)\beta_t}{m - 1} \right|$$

where β_t is the percentage of nodes leaving the network at round t.

Proof. Let n^t be the number of nodes in the system at round t. According to Definition 10, for all k, the number n_k^t of nodes in shard k at round t is $\frac{n^t}{m}$. Assuming there are $\beta_t n^t$ nodes leaving the network at round t. Let Δn_k^t be the number of leaving nodes in shard $k \in [1, m]$ at round t, we have $\sum_{k=1}^{m} \Delta n_k^t = \beta_t n^t$. Upon the next system state st_{t+1}, each node executes $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$, and its resulting shard complies with the probability distribution in Definition 4. After executing $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$, there are some nodes in shard k moving to other shards, and there are some nodes from other shards moving to shard k as well.

By the definition of *operability*, there are $\gamma(n_k^t - \Delta n_k^t)$ nodes in shard k that do not move to other shards. There are

$$(1 - \beta_t)n^t - (n_k^t - \Delta n_k^t)$$

nodes that do not belong to shard k. By Definition 4, there are

$$\frac{1 - \gamma}{m - 1}[(1 - \beta_t)n^t - (n_k^t - \Delta n_k^t)]$$

nodes moving to shard k. Thus, the number n_k^{t+1} of nodes in shard k at round $t + 1$ is

$$n_k^{t+1} = \gamma(n_k^t - \Delta n_k^t) + \frac{1 - \gamma}{m - 1}[(1 - \beta_t)n^t - (n_k^t - \Delta n_k^t)] \tag{1}$$

$$= \frac{\gamma m - 1}{m - 1}(n_k^t - \Delta n_k^t) + \frac{(1 - \gamma)(1 - \beta_t)}{m - 1}n^t \tag{2}$$

By Definition 10, to find the largest ω, we should find the largest $\frac{|n_i^{t+1}-n_j^{t+1}|}{n}$, which can be calculated as

$$\frac{|n_i^{t+1}-n_j^{t+1}|}{n^t} = \frac{|\frac{\gamma m-1}{m-1}(n_i^t-\Delta n_i^t)-\frac{\gamma m-1}{m-1}(n_j^t-\Delta n_j^t)|}{n^t} \tag{3}$$

$$= \frac{|\frac{\gamma m-1}{m-1}(\Delta n_i^t-\Delta n_j^t)|}{n^t} \tag{4}$$

When $(\Delta n_i^t - \Delta n_j^t)$ is maximal, $\frac{|n_i^{t+1}-n_j^{t+1}|}{n^t}$ is maximal, and ω is also maximal. As there are $\beta_t n^t$ nodes leaving the network in total, the maximal value of $(\Delta n_i^t - \Delta n_j^t)$ is $\beta_t n^t$. When $\Delta n_i^t - \Delta n_j^t = \beta_t n^t$,

$$\omega = 1 - \frac{|n_i^{t+1}-n_j^{t+1}|}{n^t} \tag{5}$$

$$= 1 - \frac{|\frac{\gamma m-1}{m-1}(\Delta n_i^t-\Delta n_j^t)|}{n^t} \tag{6}$$

$$= 1 - \frac{|\frac{\gamma m-1}{m-1}\beta_t n^t|}{n^t} = 1 - \left|\frac{(\gamma m-1)\beta_t}{m-1}\right| \tag{7}$$

□

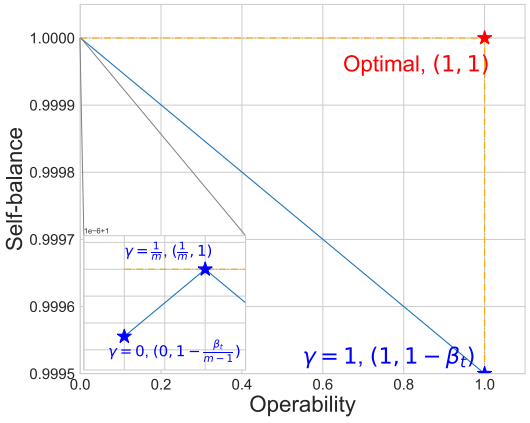

Figure 3: Relationship between *operability* and *self-balance*. We pick $m = 1000$ and $\beta_t = 0.0005$ as an example.

Figure 3 visualises the relationship between *self-balance* and *operability*, according to Lemma 1. The line never reaches the point $(1, 1)$, indicating that $\mathcal{SA}$ can never achieve both optimal *self-balance* and optimal *operability*. In addition, the *self-balance* decreases to zero then increases with *operability* increasing. When $\gamma = \frac{1}{m}$, *self-balance* becomes 1, i.e., optimal. With γ smaller than $\frac{1}{m}$, $\frac{(\gamma m-1)\beta_t}{m-1}$ becomes less than zero and keeps decreasing, so its absolute value keeps increasing. When $\gamma = 0$, *self-balance* becomes $1 - \frac{\beta_t}{m-1}$. This is because when $\gamma = 0$, all nodes are mandatory to change their shards. As shard k has fewer shards, during $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ it loses fewer nodes but receives more nodes from other shards.

Theorem 1. *Let β_t be the percentage of nodes leaving the network at round t. It is impossible for a correct shard allocation protocol $\mathcal{SA}$ with m shards to achieve optimal self-balance and operability simultaneously for any $\beta_t \neq 0$ and $m > 1$.*

Proof. We prove this by contradiction. Assuming *self-balance* $\omega = 1$ and *operability* $\gamma = 1$. According to Lemma 1, $\omega = 1$ only when either $\beta_t = 0$ or $\gamma m = 1$. As $\gamma = 1$ and $m > 1$, $\gamma m > 1$. Thus, $\mathcal{SA}$ can achieve $\omega = 1$ and $\gamma = 1$ simultaneously only when $\beta_t = 0$. However, $\beta_t > 0$, which leads to a contradiction. □

Parameterising self-balance and operability. As shown in Figure 3, $(1, 1-\beta_t)$ and $(\frac{1}{m}, 1)$ are two extreme points on the line of relationship between *self-balance* and *operability*. shard allocation protocols lying at these two points are impractical. However, none of the evaluated protocols allows the flexibility of parametrising the trade-off between *self-balance* and *operability*. We observe that, to parametrise *self-balance* and *operability*, sharding protocols should be *non-memoryless*. In signal processing, a system is said to be *memoryless* if the output signal at each time depends only on the input at that time [39]. On the contrary, *non-memoryless* indicates the output does not only depend on the current input, but also some previous inputs.

Definition 11 (Non-memoryless)**.** *We say a shard allocation protocol $\mathcal{SA}$ is non-memoryless iff for any secret key sk_i, public parameter pp, and shard k, the output of*

$$\mathcal{SA}.\mathsf{UpdateShard}(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1})$$

depends on system states earlier than st_t.

As both *self-balance* and *operability* are related to the probability γ (see Definition 4) of nodes staying at the same shard, to be able to parametrise *self-balance* and *operability*, a shard allocation protocol should have its memory on the shard allocation in the previous states.

Theorem 2. *If a correct shard allocation protocol $\mathcal{SA}$ is ω-self-balanced and γ-operable where $\omega \in (0, 1-\beta_t)$ and $\gamma \in (\frac{1}{m}, 1)$, then $\mathcal{SA}$ is non-memoryless.*

Proof. We prove this by contradiction. Assuming $\mathcal{SA}$ is memoryless, i.e., the output of $\mathcal{SA}.\mathsf{UpdateShard}(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1})$ only depends on st_t and st_{t+1}. This means there exists no $\delta \geq 1$ such that $\pi_{i,st_t,k}$ involves any information of $st_{t-\delta}$.

When $\gamma \in (\frac{1}{m}, 1)$, the distribution of the resulting shard of $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ is *non-uniform*, given the *update-randomness* property. In this case, executing $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ requires the knowledge of k — index of the shard that i locates at state st_t. Thus, $\pi_{i,st_{t+1},k'}$ — one of the output of $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ — should enable verifiers to verify node i is at shard k at state st_t.

"Verify node i is at shard k at state st_t" is achieved by verifying $\pi_{i,st_t,k}$. This means $\pi_{i,st_{t+1},k'}$ depends on st_t and $\pi_{i,st_t,k}$. Similarly, $\pi_{i,st_t,k}$ depends on st_{t-1} and $\pi_{i,st_{t-1},k}$, and $\pi_{i,st_{t-1},k}$ depends on st_{t-2} and $\pi_{i,st_{t-2},k}$. Recursively, $\pi_{i,st_t,k}$

depends on all historical system states. Thus, if the assumption holds, then this contradicts *update-randomness*. □

Remark 1. *Note that when* $\gamma = \frac{1}{m}$ *or* 1, $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ *does not need to depend on any prior system state. When* $\gamma = \frac{1}{m}$, *the distribution of the resulting shard of* $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ *is uniform, so* $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ *can just assign nodes randomly according to the incoming system state. When* $\gamma = 1$, *the resulting shard of* $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ *is certain, and* $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ *does not need to depend on any system state. All of our evaluated shard allocation protocols choose* $\gamma = \frac{1}{m}$ *or* 1, *except for RapidChain using Commensal Cuckoo and Chainspace allowing nodes to choose shards upon requests.*

V. WORMHOLE— SHARD ALLOCATION FROM RANDOMNESS BEACON

Based on the gained insights, we propose WORMHOLE, a correct and efficient permissionless shard allocation protocol that supports parametrisation of *self-balance* and *operability*. WORMHOLE only relies on a randomness beacon (e.g. [40]–[43]) and a verifiable random function (e.g. [44]–[46].

For a fairer comparison, we also evaluate Distributed Randomness Generation (DRG)-based shard allocation protocols while replacing DRG protocols with an external randomness beacon. The evaluation shows that DRG is the major source of strong network and fault tolerance assumptions and introduces significant communication overhead. However, even with a randomness beacon, these DRG-based shard allocation protocols still suffer from some problems they originally have, especially the poor *operability*.

A. Primitives

WORMHOLE relies on verifiable random functions (VRFs) and randomness beacon, defined as follows.

Verifiable random functions. VRF [44] is a public-key version of the cryptographic hash function. In addition to the input string, VRF also requires a pair of secret and public keys. Given an input string and a secret key, one can compute a hash and a proof. Anyone knowing the associated public key and the proof can verify whether the hash is from the input and whether the hash is generated by the owner of the secret key. Formally, a VRF is a tuple of four algorithms ($\mathsf{VRFKeyGen}$, $\mathsf{VRFHash}$, $\mathsf{VRFProve}$ and $\mathsf{VRFVerify}$).

- $\mathsf{VRFKeyGen}(\lambda) \rightarrow (sk, pk)$: On input a security parameter λ, outputs the secret/public key pair (sk, pk).
- $\mathsf{VRFHash}(sk, str) \rightarrow h$: On input sk and an arbitrary-length string str, outputs a fixed-length hash h.
- $\mathsf{VRFProve}(sk, str) \rightarrow \pi$: On input sk and str, outputs the proof π for h.
- $\mathsf{VRFVerify}(pk, str, h, \pi) \rightarrow \{0, 1\}$: On input pk, str, h, π, outputs the verification result 0 or 1.

A VRF should satisfy three security properties [47] as follows.

- *VRF-Uniqueness*: Given a secret key sk and an input str, $\mathsf{VRFHash}(sk, str)$ produces a unique valid output.
- *VRF-Collision-Resistance*: It is computationally hard to find two inputs str and str' that $\mathsf{VRFHash}(sk, str) = \mathsf{VRFHash}(sk, str')$.
- *VRF-Pseudorandomness*: It is computationally hard to distinguish the output of $\mathsf{VRFHash}(\cdot)$ from a random string if not knowing the corresponding public key and proof.

Randomness beacon and distributed randomness generation. Randomness beacon protocols are designed to generate randomness periodically, which are usually constructed from distributed randomness generation (DRG) protocols [35], [42], [48]–[54] — a family of protocols where nodes jointly produce a random output. A randomness beacon should satisfy the following properties [43]:

- *RB-Availability*: No node can prevent the protocol from making progress.
- *RB-Unpredictability*: No node can know the value of the random output before it is produced.
- *RB-Unbiasibility*: No node can influence the value of the random output.
- *RB-Public-Verifiability*: Everyone can verify the correctness of the random output.

B. Basic idea

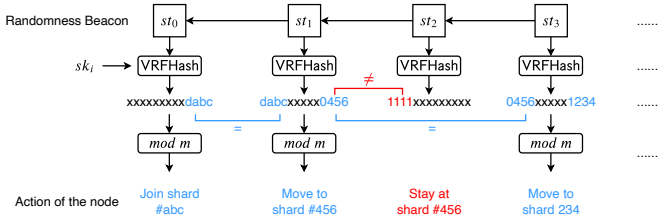

Figure 4: An intuition of WORMHOLE. All numbers are in hexadecimal, $op = 4$ and $m = 16^3$.

WORMHOLE relies on a secure randomness beacon $\mathcal{RB}$. Each new random output from $\mathcal{RB}$ updates the system state. We consider each shard as a "universe". Each time a new state is generated, every node in the system travels through a series of "wormholes" to arrive at a destination universe, which may or may not be the same universe before travelling.

We introduce an operability parameter op to tune the trade-off between *self-balance* and *operability*. The operability parameter op defines the hardness of a node allocating to a different shard. As shown in Figure 4, if the op least significant bits (LSBs) of the previous state's VRF output equal to the op most significant bits (MSBs) of a incoming state's VRF output, then the node moves to a new shard according to the incoming state's VRF output, otherwise the node stays. If op is very large, then the probability that nodes move to a different shard is very small.

As proved in Theorem 2 and Remark 1, if it's possible that operability $\gamma \in (\frac{1}{m}, 1)$, each membership proof should carry information of all states rather than just the current and/or incoming states. However, this keeps proof size growing as

the system operates. To limit the number of previous states required to prove shard membership, we introduce a Sybil resistance parameter sr. WORMHOLE runs in epochs, each epoch consists of sr system states. Joining the system and relocating a shard require the memory of ℓ random outputs, where $\ell \in [sr, 2sr)$. In other words, for each shard allocation, a node needs to travel through ℓ wormholes to reach the final destination. The Sybil resistance parameter sr also controls the sybil resistance degree. With larger sr, joining the system will require nodes to run VRF for more times.

An adversary cannot move a node to its preferred shards by repetitively joining the system. As VRF is deterministic, the node is always allocated to the same shard if joining the system at the same state. If the adversary assigns a new key pair to this node, then it needs to compute ℓ VRFs again, leading to non-negligible overhead. In addition, the adversary cannot gather its nodes to the same shard to reach the shard's security threshold. Upon each new system state, some nodes may be allocated to other shards.

Algorithm 2: Calculating the shard using VRF hashes.

Algorithm $\mathsf{calculateShard}\,(m, op, h_{t-\ell+1}, \ldots, h_t)$ **:**
- $shard\_id \leftarrow h_{t-\ell+1} \mod m$
- $idx \leftarrow t - \ell + 1$
- **for** $j \in [t-\ell+2, t]$ **do**
  - **if** $\mathsf{MSB}(op, h_j) = \mathsf{LSB}(op, h_{idx})$ **then**
    - $shard\_id \leftarrow h_j \mod m$
    - $idx \leftarrow j$
- **return** $shard\_id$

C. Detailed construction

System setup (Algorithm 3) The setup algorithm takes a security parameter λ as input, outputs the number of shards m, the Sybil resistance parameter sr, and the operability parameter op.

Algorithm 3: System setup.

Algorithm $\mathsf{Setup}\,(\lambda)$ **:**
- $m, op, sr \leftarrow \lambda$
- **return** (m, op, sr)

Joining a shard (Algorithm 4) A node i executes $\mathsf{JoinShard}(sk_i, pp, st_t)$ in order to obtain a membership of a shard at round t. First, node i calculates VRF hashes and proofs of the latest ℓ system states, where $\ell = t \mod sr + sr$. Node i calculates the ID k of the shard it belongs to by $\mathsf{calculateShard}(\cdot)$ (Algorithm 2). The proof $\pi_{i,st_t,k}$ that node i has a valid membership in shard k at state st includes node i's public key pk_i, a sequence of VRF hashes $h_{t-\ell+1}, h_{t-\ell+2}, \ldots, h_t$, together with their proofs $\pi_{t-\ell+1}, \pi_{t-\ell+2}, \ldots, \pi_t$.

Algorithm 4: Joining a shard.

Algorithm $\mathsf{JoinShard}\,(sk_i, pp, st_t)$ **:**
- $m, op, sr \leftarrow pp$
- $\ell \leftarrow t \mod sr + sr$
- **for** $j \in [t-\ell+1, t]$ **do**
  - $h_j \leftarrow \mathsf{VRFHash}(sk_i, st_j)$
  - $\pi_j \leftarrow \mathsf{VRFProve}(sk_i, st_j)$
- $k \leftarrow \mathsf{calculateShard}(m, op, h_{t-\ell+1}, \ldots, h_t)$
- $\pi_{i,st_t,k} \leftarrow (pk_i, h_{t-\ell+1}, \ldots, h_t, \pi_{t-\ell+1}, \ldots, \pi_t)$
- Store $\pi_{i,st_t,k}$ in memory
- **return** $k, \pi_{i,st_t,k}$

Updating shard membership (Algorithm 5) Updating shard membership follows a process similar to joining a shard. After joining the system at round t, in the next state $t+1$ node i only needs to calculate one more VRF hash of st_{t+1} and repeats the process in Algorithm 4 over the sequence of states from $st_{t-\ell+1}$ to st_{t+1}.

Algorithm 5: Updating shard membership.

Algorithm
$\mathsf{UpdateShard}\,(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1})$ **:**
- $m, op, sr \leftarrow pp$
- $\ell \leftarrow t \mod sr + sr$
- $(pk_i, h_{t-\ell+1}, \ldots, h_t, \pi_{t-\ell+1}, \ldots, \pi_t) \leftarrow \pi_{i,st_t,k}$
- $\ell^+ \leftarrow (t+1) \mod sr + sr$
- **for** $j \in [t-\ell+1, t-\ell^++1]$ **do**
  - Remove h_j and π_j from memory
- $h_{t+1} \leftarrow \mathsf{VRFHash}(sk_i, st_{t+1})$
- $\pi_{t+1} \leftarrow \mathsf{VRFProve}(sk_i, st_{t+1})$
- $k' \leftarrow \mathsf{calculateShard}(m, op, h_{t-\ell^++2}, \ldots, h_{t+1})$
- $\pi_{i,st_{t+1},k'} \leftarrow$ $(pk_i, h_{t-\ell^++2}, \ldots, h_{t+1}, \pi_{t-\ell^++2}, \ldots, \pi_{t+1})$
- Store $\pi_{i,st_{t+1},k'}$ in memory
- **return** $k', \pi_{i,st_{t+1},k'}$

Verifying shard membership (Algorithm 6) Verifying a proof $\pi_{i,st_t,k}$ of shard membership consists of two steps. The verifier executes $\mathsf{VRFVerify}(\cdot)$ to checks the validity of all ℓ VRF hashes and VRF proofs, and executes $\mathsf{calculateShard}(\cdot)$ over these VRF hashes to verify its output against k.

D. Security and performance analysis

We formally prove the correctness and evaluate the performance of WORMHOLE.

Lemma 2. *If the deployed $\mathcal{RB}$ generates $\lfloor \tau \cdot s \rfloor$ random outputs in any s round (where $\tau \in \mathbb{R}_{>0}$ and $s \in \mathbb{N}_{>0}$), then* WORMHOLE *satisfies (τ, s)-liveness.*

Proof. We prove this by contradiction. Assuming that WORMHOLE does not satisfy (τ, s)-liveness, i.e., there exists a sequence of s rounds when the system generates less than

Algorithm 6: Verifying shard membership.

Algorithm VerifyShard $(pp, pk_i, st_t, k, \pi_{i,st_t,k})$ **:**
- $m, op, sr \leftarrow pp$
- $\ell \leftarrow t \mod sr + sr$
- $(pk_i, h_{last}, \ldots, h_t, \pi_{last}, \ldots, \pi_t) \leftarrow \pi_{i,st_t,k}$
- **if** $last \neq t - \ell + 1$ **then**
  - **return** 0
- **for** $j \in [t - \ell + 1, t]$ **do**
  - **if** $\mathsf{VRFVerify}(pk_i, st_j, h_j, \pi_j) = 0$ **then**
    - **return** 0
- **if** $k \neq \mathsf{calculateShard}(m, op, h_{t-\ell+1}, \ldots, h_t)$ **then**
  - **return** 0
- **return** 1

$\lfloor \tau \cdot s \rfloor$ states. By RB-Availability, no node can prevent $\mathcal{RB}$ from producing fresh randomness. Each new random output updates the system state. If $\mathcal{RB}$ generates $\lfloor \tau \cdot s \rfloor$ random outputs in any s rounds, then $\mathcal{RB}$ updates the system state for $\lfloor \tau \cdot s \rfloor$ times in any s rounds. Thus, if WORMHOLE does not satisfy (τ, s)-liveness, then this contradicts RB-Availability. □

Lemma 3. WORMHOLE *satisfies unbiasibility.*

Proof. We prove this by contradiction. Assuming that WORMHOLE does not satisfy *unbiasibility*: given a system state, an adversary can manipulate the probability distribution of the output shard of $\mathsf{JoinShard}(\cdot)$ or $\mathsf{UpdateShard}(\cdot)$ with non-negligible probability. This consists of three attack vectors: 1) the adversary can manipulate the system state; 2) when $\mathcal{SA}.\mathsf{JoinShard}(\cdot)$ or $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ are probabilistic, the adversary can keep generating memberships until outputting a membership of its preferred shard; and 3) the adversary can forge proofs of memberships of arbitrary shards.

By RB-Unbiasibility, the randomness produced by $\mathcal{RB}$ is unbiasible, so the system state of WORMHOLE is unbiasible. By VRF-Uniqueness, given a secret key, the VRF hash of the system state is unique, which eliminates the last two attack vectors. In addition, the uniqueness of the VRF hash of the unbiasible system state indicates that the VRF hash is unbiasible. The output shard of $\mathsf{JoinShard}(\cdot)$ or $\mathsf{UpdateShard}(\cdot)$ is a modulus of the VRF hash, which is also unbiasible. This eliminates the first attack vector. Thus, if WORMHOLE does not resist against the first attack vector, then this contradicts RB-Unbiasibility; and if WORMHOLE does not resist against the second and/or the last attack vectors, then this contradicts VRF-Uniqueness. □

Lemma 4. WORMHOLE *satisfies join-randomness.*

Proof. We prove this by contradiction. Assuming that WORMHOLE does not satisfy *join-randomness*, i.e., the probabilistic of a node joining a shard $k \in [1, m]$ is $\frac{1}{m} + \epsilon$ for some k and non-negligible ϵ. Running $\mathsf{JoinShard}(\cdot)$ requires the execution of $\mathsf{VRFHash}(\cdot)$ over a series of system states. By VRF-Pseudorandomness, VRF hashes of system states are pseudorandom. As a modulo of a VRF hash, the output shard of $\mathsf{JoinShard}(\cdot)$ is also pseudorandom. Thus, if WORMHOLE does not satisfy *join-randomness*, then this contradicts VRF-Pseudorandomness. □

Lemma 5. WORMHOLE *satisfies update-randomness.*

Proof. We prove this by contradiction. Assuming that WORMHOLE does not satisfy *update-randomness*, i.e., with non-negligible probability, there is no γ such that the probability of a node joining a shard k complies with the distribution in Definition 4. When $t = k \cdot sr$, the last VRF hash will change. When this happens, all nodes will be shuffled. Note that this is not frequent when sr is large. By VRF-Pseudorandomness, the probability of moving to each shard is same. Thus, there is a $\gamma = \frac{1}{m}$ that makes the output shard of $\mathsf{UpdateShard}(\cdot)$ to comply with the distribution in Definition 4.

When $t \neq k \cdot sr$, the last VRF hash remains unchanged. In $\mathsf{UpdateShard}(\cdot)$, given the last VRF hash, the probability that the op MSBs of the new VRF hash equal to op LSBs of the last VRF hash is $\frac{1}{2^{op}}$. By VRF-Pseudorandomness, the probability of moving to each other shard is same. Thus, there is a $\gamma = 1 - \frac{1}{2^{op}} \cdot \frac{m-1}{m} = 1 - \frac{m-1}{m \cdot 2^{op}}$ that makes the output shard of $\mathsf{UpdateShard}(\cdot)$ to comply with the distribution in Definition 4.

Thus, if WORMHOLE does not satisfy *update-randomness*, then this contradicts VRF-Pseudorandomness. □

Lemma 6. WORMHOLE *satisfies allocation-privacy.*

Proof. This follows proofs of Lemma 4 and 5. □

Theorem 3. WORMHOLE *is a correct shard allocation protocol.*

Proof. By Lemma 2-6, WORMHOLE is a correct shard allocation protocol. □

Performance metrics The *communication complexity* of $\mathsf{JoinShard}(\cdot)$ and $\mathsf{UpdateShard}(\cdot)$ of WORMHOLE are $O(1)$. To execute $\mathsf{JoinShard}(\cdot)$, the only communication that a node should make is to receive ℓ random outputs, where $\ell = t \mod sr + sr$. As sr is a constant, the *communication complexity* of $\mathsf{JoinShard}(\cdot)$ is $O(1)$. To execute $\mathsf{UpdateShard}(\cdot)$, a node only need to receive the latest randomness, and the *communication complexity* of $\mathsf{UpdateShard}(\cdot)$ is also $O(1)$. WORMHOLE resists Sybil attacks using computation. To execute $\mathsf{JoinShard}(\cdot)$, a node should executes $\mathsf{VRFHash}$ for ℓ times, where $\ell = t \mod sr + sr$. One can make WORMHOLE more Sybil-resistant by making $\mathsf{VRFHash}(\cdot)$ more computation-intensive. WORMHOLE's *operability* γ is $1 - \frac{m-1}{m \cdot 2^{op}}$. By Proof V-D,

$$\gamma = 1 - \frac{m-1}{m} \cdot \frac{1}{2^{op}} = 1 - \frac{m-1}{m \cdot 2^{op}}$$

The *self-balance* ω of WORMHOLE is $1 - \beta_t + \frac{\beta_t}{2^{op}}$, where β_t is the percentage of nodes that leave from the system at round t. By Definition 1,

$$\omega = 1 - \left| \frac{(\gamma m - 1)\beta_t}{m - 1} \right| \tag{8}$$

$$= 1 - \frac{1}{m-1} \cdot [(1 - \frac{m-1}{m \cdot 2^{op}})m - 1]\beta_t \tag{9}$$

$$= 1 - \frac{1}{m-1} \cdot [(m-1) - \frac{m-1}{2^{op}}]\beta_t \tag{10}$$

$$= 1 - (1 - \frac{1}{2^{op}})\beta_t \tag{11}$$

$$= 1 - \beta_t + \frac{\beta_t}{2^{op}} \tag{12}$$

E. Comparison with shard allocation protocols with randomness beacon

Elastico, Omniledger, RapidChain, and Zilliqa employ DRG protocols, which can be replaced by a randomness beacon. As DRG protocols usually rely on strong assumptions and suffer from high communication complexity, replacing them by a randomness beacon can significantly simplify assumptions, improve security, and reduce communication overhead of shard allocation protocols. For Elastico, the network model can be asynchronous, the fault tolerance will be 1, and the communication complexity will be constant. For Omniledger, the network does not need to be partially synchronous, the fault tolerance will be 1, and the communication complexity of $\mathsf{UpdateShard}(\cdot)$ will be $O(n)$. RapidChain will be able to assume synchronous networks and tolerate any percentage of Byzantine faults, and the communication complexity of $\mathsf{JoinShard}(\cdot)$ and $\mathsf{UpdateShard}(\cdot)$ will be $O(n)$.

We evaluate these DRG-based shard allocation protocols while replacing DRG protocols with a randomness beacon. The evaluation in Table III shows that DRG is the major source of strong network and fault tolerance assumptions and introduces significant communication overhead. For example, with a randomness beacon, all of these shard allocation protocols can work in asynchronous networks, and three of them raise their fault tolerance capacity to 1 and their communication complexity to $O(n)$ or even 0.

However, all of them still suffer from some problems they originally have. Most results on their correctness remain unchanged: RapidChain still lacks *public verifiability*; Elastico and Zilliqa are still partially biasible; Omniledger and RapidChain still do not satisfy *privacy*. In addition, Omniledger should still assume an identity authority for approving nodes to join the system. Moreover, all of them still suffer from poor *operability*.

F. Practical considerations

The randomness beacon assumption. WORMHOLE relies on a randomness beacon as trusted third party. We consider randomness beacon as an acceptable assumption. Randomness beacon is a weak trusted third party — weaker than the identity authority and the smart contract used in Omniledger and Chainspace, respectively. A compromised randomness beacon only breaks WORMHOLE's *liveness*, but not the other four properties. As long as the produced randomness is uniformly distributed, then WORMHOLE remains correct. If the compromised randomness beacon produces biased randomness, nodes can detect it by checking the randomness' distribution. If the compromised randomness beacon stops working, then shard allocation only loses *liveness*. In addition, there have been emerging decentralised randomness beacon protocols, from both academia and industry. Researchers construct decentralised randomness beacon based on blockchains [41], Publicly Verifiable Secret Sharing (PVSS) [42], [43], [55] and/or Verifiable Delay Functions [56], [57]. Some countries [58]–[60] and organisations [61], [62] also deploy centralised randomness beacons as public services.

Construction without allocation-privacy. As mentioned in §II-B, *allocation-privacy* is not always a desired property. To remove *allocation-privacy* from WORMHOLE, one can replace $\mathsf{VRFHash}(sk_i, st_t)$ with $H(pk_i||st_t)$, where sk_i and pk_i are key pairs of node i, st_t is a system state, and $H(\cdot)$ is a cryptographic hash function.

Overhead. WORMHOLE introduces little overhead for both communication and computation. The majority of overhead is propagating and verifying proofs of memberships. Each proof of membership consists of a public key and $\ell = t \mod sr + sr \in [sr, 2sr)$ VRF outputs and VRF proofs. For each membership proof, one should run $\mathsf{VRFVerify}(\cdot)$ for $[sr, 2sr)$ times. A public key, a VRF output or a VRF proof usually takes 32 bytes. Thus, the proof size is $32(\ell+1) = 32(t \mod sr + sr + 1)$ bytes.

Consider $sr = 10,000$, which is relatively large. Upon a new peer, a node should receive a membership proof of approximately $320 \sim 640$ KB, and verify the proof by running $\mathsf{VRFVerify}(\cdot)$ for $[10,000, 20,000)$ times. Upon a new randomness, this node only needs to receive one more VRF output the associated VRF proof, which can be small. If a node connects with 125 inbound peers and 8 outbound peers (which are maximum values of Bitcoin-core [63]), it should receive ~ 85 MB for membership proofs, then keep synchronising and verifying new VRF outputs and proofs upon new system states. Compared to synchronising and verifying blockchains, propagating and verifying membership proofs introduces negligible overhead. In addition, we can compress membership proofs to succinct ones using advanced cryptographic primitives, e.g., Proof-Carrying Data [64], [65].

VI. RELATED WORK

We now briefly review existing research on sharding distributed systems and compare our contributions with two studies that systematising permissionless sharding designs.

Sharding for Crash Fault Tolerant distributed systems. Sharding has been widely deployed in crash fault tolerance (CFT) systems to raise their throughput. Allocating nodes to shards in a CFT system is straightforward, as there is no Byzantine adversaries in the system, and the total number of nodes is fixed and known to everyone [1], [4], [66]. The main

Table III: Evaluation of shard allocation protocols that replace DRG with a randomness beacon. Meanings of colours are same as Table II. ★ means the metric is improved by replacing DRG with a randomness beacon.

		System model			Correctness					Performance metrics				
	State update	Network model	Trust assumption	Fault tolerance	Public verifiability	Liveness	Allocation-rand.	Unbiasibility	Privacy$^{\circ}$	Join comm. compl.	Update comm. compl.	Sybil cost	Self-balance	Operability
Elastico	New block	Async.★	Rand. Beacon	1★	✓	✓	✓	✗*	✓	0★	0★	Comp.	1	$\frac{1}{m}$
Omniledger	Identity authority	Async.★	Identity auth. Rand. Beacon	1★	✓	✓	✓	✓	✗	$O(1)$	$O(n)$★	-	1	$\frac{1}{m}$
RapidChain	Nodes joining	Async.★	Rand. Beacon	1★	✗	✓	✓	✓	✗	$O(n)$★	$O(n)$★	Comp.	$1-\beta_t$	$max(1-\kappa\alpha_t n, 0)$
Zilliqa	New block	Async.★	Rand. Beacon	1	✓	✓	✓	✗*	✓	0	0	Comp.	1	$\frac{1}{m}$
WORMHOLE (Our proposal in §V)	New rand.	Async.	Rand. Beacon	1	✓	✓	✓	✓	✓	0	0	Comp.	$1-\beta_t+\frac{\beta_t}{2^{\sigma p}}$	$1-\frac{m-1}{m\cdot 2^{\sigma p}}$

$^{\circ}$ Optional. * Protected by PoW puzzles.

challenge is to balance the computation, communication, and storage workload. Despite a large number of load-balancing algorithms [5]–[8], [67], none of them is applicable in the permissionless setting as they do not tolerate Byzantine faults.

Distributed Hash Tables. Many peer-to-peer (P2P) storage services [68], [69] employ Distributed Hash Tables (DHT) [70] to assign file metadata, i.e., a list of keys, to their responsible nodes. In a DHT, nodes share the same ID space with the keys; a file's metadata is stored at the nodes whose IDs are closest to the keys. Although designed to function in a permissionless environment, DHTs are vulnerable to several attacks [71]–[73], therefore are not suitable for blockchains, which demands strong consistency on financial data.

Distributed Slicing. Distributed Slicing [74] aims at grouping nodes with heterogeneous computing and storage capacities in a P2P network to optimize resource utilisation. In line with CFT systems, these algorithms [75]–[78] require each node to honestly report their computing and storage capacities, therefore are not suitable in a Byzantine environment.

Evaluation of sharded blockchains. Wang et al. [16] propose an evaluation framework based on Elastico's architecture; Avarikioti et al. [17] formalise sharded blockchains by extending the model of Garay et al. [21]. Both of them aim at evaluating the entire sharded designs, and put most efforts on DRG or cross-shard communication, neglecting the security and performance challenges of shard allocation.

VII. CONCLUSION

In this paper, we formally define the permissionless shard allocation protocol, including its syntax, correctness, and performance metrics. Based on the formalisation, we evaluate existing shard allocation protocols. Our evaluation shows that none of them is correct, and some of them suffer from poor performance or rely on unrealistic assumptions. From our evaluation, we observe and formally prove a trade-off between *self-balance* and *operability*, and discuss how to parametrise it. Based on this insight, we propose WORMHOLE, a correct and efficient shard allocation protocol from a randomness beacon. WORMHOLE is correct and more efficient than our evaluated shard allocation protocols, and supports parametrising *Sybil cost* and the trade-off between *self-balance* and *operability*.

REFERENCES

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

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

[3] G. Danezis and S. Meiklejohn, "Centrally banked cryptocurrencies", *arXiv preprint arXiv:1505.06895*, 2015.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data", *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store", *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[6] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores", in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 79–94.

[7] B. Augustin, T. Friedman, and R. Teixeira, "Measuring load-balanced paths in the Internet", in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, 2007, pp. 149–160.

[8] M. Annamalai, K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, and M. Stumm, "Sharding the shards: managing datastore locality at scale with Akkio", in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 445–460.

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

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

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

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

[13] J. Wang and H. Wang, "Monoxide: Scale out Blockchains with Asynchronous Consensus Zones", in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 95–112.

[14] Z. Team *et al.*, "The ZILLIQA Technical Whitepaper", *Retrieved September*, vol. 16, p. 2019, 2017.

[15] (). Ethereum/eth2.0-specs, [Online]. Available: https://github.com/ethereum/eth2.0-specs.

[16] G. Wang, Z. J. Shi, M. Nixon, and S. Han, *Sok: Sharding on blockchain*, Cryptology ePrint Archive, Report 2019/1178, 2019.

[17] G. Avarikioti, E. Kokoris-Kogias, and R. Wattenhofer, "Divide and Scale: Formalization of Distributed Ledger Sharding Protocols", *arXiv preprint arXiv:1910.10434*, 2019.

[18] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt, "Sok: Communication across distributed ledgers", IACR Cryptology ePrint Archive, 2019: 1128, Tech. Rep., 2019.

[19] (). Ethereum/wiki, [Online]. Available: https://github.com/ethereum/wiki.

[20] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks", in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, ACM, 2006, pp. 189–202.

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

[22] C. Natoli, J. Yu, V. Gramoli, and P. Esteves-Verissimo, "Deconstructing Blockchains: A Comprehensive Survey on Consensus, Membership and Structure", *arXiv preprint arXiv:1908.08316*, 2019.

[23] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network", in *IEEE P2P 2013 Proceedings*, IEEE, 2013, pp. 1–10.

[24] X. Qian, "Improved authenticated data structures for blockchain synchronization", PhD thesis, 2018.

[25] (). Warp sync - wiki parity tech documentation, [Online]. Available: https://wiki.parity.io/Warp-Sync.

[26] A. C.-C. Yao, "Some complexity questions related to distributive computing (preliminary report)", in *Proceedings of the eleventh annual ACM symposium on Theory of computing*, 1979, pp. 209–213.

[27] J. R. Douceur, "The sybil attack", in *International workshop on peer-to-peer systems*, Springer, 2002, pp. 251–260.

[28] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols", in *Secure Information Networks*, Springer, 1999, pp. 258–272.

[29] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake", *self-published paper, August*, vol. 19, 2012.

[30] J. Zhao, J. Yu, and J. K. Liu, "Consolidating Hash Power in Blockchain Shards with a Forest", in *International Conference on Information Security and Cryptology*, Springer, 2019, pp. 309–322.

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

[32] B. Awerbuch and C. Scheideler, "Robust random number generation for peer-to-peer systems", in *International Conference On Principles Of Distributed Systems*, Springer, 2006, pp. 275–289.

[33] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing", in *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, IEEE, 1987, pp. 427–438.

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

[35] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography", *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

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

[37] J. Yu, D. Kozhaya, J. Decouchant, and P. Verissimo, "RepuCoin: Your reputation is your power", *IEEE Transactions on Computers*, 2019.

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

[39] A. V. Oppenheim, *Discrete-time signal processing*. Pearson Education, 1999.

[40] B. Bünz, S. Goldfeder, and J. Bonneau, "Proofs-of-delay and randomness beacons in ethereum", *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

[41] J. Bonneau, J. Clark, and S. Goldfeder, "On Bitcoin as a public randomness source.", *IACR Cryptology ePrint Archive*, vol. 2015, p. 1015, 2015.

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

[43] P. Schindler, A. Judmayer, N. Stifter, and E. Weippl, "HydRand: Efficient Continuous Distributed Randomness", in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 32–48.

[44] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions", in *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, IEEE, 1999, pp. 120–130.

[45] Y. Dodis, "Efficient construction of (distributed) verifiable random functions", in *International Workshop on Public Key Cryptography*, Springer, 2003, pp. 1–17.

[46] S. Goldberg, J. Vcelak, D. Papadopoulos, and L. Reyzin, "Verifiable random functions (VRFs)", 2018.

[47] S Goldberg, D Papadopoulos, and J Vcelak, *draft-goldbe-vrf: Verifiable Random Functions.(2017)*, 2017.

[48] M. Blum, "Coin flipping by telephone a protocol for solving impossible problems", *ACM SIGACT News*, vol. 15, no. 1, pp. 23–27, 1983.

[49] M. O. Rabin, "Transaction protection by beacons", *Journal of Computer and System Sciences*, vol. 27, no. 2, pp. 256–267, 1983.

[50] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems", in *International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 1999, pp. 295–310.

[51] J. A. Halderman and B. Waters, "Harvesting verifiable challenges from oblivious online sources", in *Proceedings of the 14th ACM conference on Computer and communications security*, ACM, 2007, pp. 330–341.

[52] O. Oluwasanmi and J. Saia, "Scalable byzantine agreement with a random beacon", in *Symposium on Self-Stabilizing Systems*, Springer, 2012, pp. 253–265.

[53] A. Kate and I. Goldberg, "Distributed key generation for the internet", in *2009 29th IEEE International Conference on Distributed Computing Systems*, IEEE, 2009, pp. 119–128.

[54] E. Kokoris-Kogias, A. Spiegelman, D. Malkhi, and I. Abraham, "Bootstrapping Consensus Without Trusted Setup: Fully Asynchronous Distributed Key Generation",

[55] I. Cascudo and B. David, "Albatross: Publicly attestable batched randomness based on secret sharing",

[56] A. K. Lenstra and B. Wesolowski, "A random zoo: Sloth, unicorn, and trx.", *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 366, 2015.

[57] N. Ephraim, C. Freitag, I. Komargodski, and R. Pass, "Continuous verifiable delay functions", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2020, pp. 125–154.

[58] J. Kelsey, L. T. Brandão, R. Peralta, and H. Booth, "A reference for randomness beacons: Format and protocol version 2", National Institute of Standards and Technology, Tech. Rep., 2019.

[59] (). Random uchile - random uchile, [Online]. Available: https://beacon.clcert.cl/en/.

[60] (). Brazilian beacon, [Online]. Available: https://beacon.inmetro.gov.br/.

[61] (). Distributed randomness beacon — cloudflare, [Online]. Available: https : / / www . cloudflare . com / leagueofentropy/.

[62] (). Unicorn beacon by lacal, [Online]. Available: http://trx.epfl.ch/beacon/index.php.

[63] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies*. " O'Reilly Media, Inc.", 2014.

[64] S. Bowe, J. Grigg, and D. Hopwood, "Halo: Recursive Proof Composition without a Trusted Setup", Cryptology ePrint Archive, Report 2019/1021, Tech. Rep., 2019.

[65] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner, "Proof-carrying data from accumulation schemes.", *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 499, 2020.

[66] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform", *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[67] S. Che, G. Rodgers, B. Beckmann, and S. Reinhardt, "Graph coloring on the GPU and some techniques to improve load imbalance", in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IEEE, 2015, pp. 610–617.

[68] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The bittorrent p2p file-sharing system: Measurements and analysis", in *International Workshop on Peer-to-Peer Systems*, Springer, 2005, pp. 205–216.

[69] Y. Kulbak, D. Bickson, *et al.*, "The eMule protocol specification", *eMule project, http://sourceforge. net*, 2005.

[70] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network", in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001, pp. 161–172.

[71] C. Lin, Y. Jiang, X. Chu, H. Yang, *et al.*, "An effective early warning scheme against pollution dissemination for BitTorrent", in *GLOBECOM 2009-2009 IEEE Global Telecommunications Conference*, IEEE, 2009, pp. 1–7.

[72] X. Lou and K. Hwang, "Collusive piracy prevention in P2P content delivery networks", *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 970–983, 2009.

[73] P. Dhungel, D. W. 0001, B. Schonhorst, and K. W. Ross, "A measurement study of attacks on BitTorrent leechers.", in *IPTPS*, vol. 8, 2008, pp. 7–7.

[74] M. Jelasity and A.-M. Kermarrec, "Ordered slicing of very large-scale overlay networks", in *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*, IEEE, 2006, pp. 117–124.
[75] A. Fernández, V. Gramoli, E. Jiménez, A.-M. Kermarrec, and M. Raynal, "Distributed slicing in dynamic systems", in *27th International Conference on Distributed Computing Systems (ICDCS'07)*, IEEE, 2007, pp. 66–66.
[76] V. Gramoli, Y. Vigfusson, K. Birman, A.-M. Kermarrec, and R. van Renesse, "A fast distributed slicing algorithm", in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, 2008, pp. 427–427.
[77] F. Maia, M. Matos, R. Oliveira, and E. Riviere, "Slicing as a distributed systems primitive", in *2013 Sixth Latin-American Symposium on Dependable Computing*, IEEE, 2013, pp. 124–133.
[78] D. J. DeWitt, J. F. Naughton, and D. F. Schneider, "Parallel sorting on a shared-nothing architecture using probabilistic splitting", University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1991.
[79] B. Schoenmakers, "A simple publicly verifiable secret sharing scheme and its application to electronic voting", in *Annual International Cryptology Conference*, Springer, 1999, pp. 148–164.
[80] (). Zilliqa - the next generation, high throughput blockchain platform, [Online]. Available: https://zilliqa.com/.
[81] (). Zilliqa developer portal · technical and api documentation for participating in the zilliqa network., [Online]. Available: https://zilliqa.github.io/dev-portal/en/.
[82] (). Zilliqa/zilliqa at v5.0.1, [Online]. Available: https://github.com/Zilliqa/Zilliqa/tree/v5.0.1.
[83] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger", *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[84] V. Buterin. (). Casper the friendly finality gadget, [Online]. Available: https://vitalik.ca/files/casper_note.html.
[85] ——, (). Serenity design rationale, [Online]. Available: https://notes.ethereum.org/@vbuterin/rkhCgQteN?type=view.
[86] (). Phase 0 for humans, [Online]. Available: https://notes.ethereum.org/@djrtwo/Bkn3zpwxB?type=view.
[87] (). Beacon chain research synopsis, [Online]. Available: https://github.com/prysmaticlabs/prysm/blob/master/docs/PRIOR_RESEARCH.md.
[88] (). Ethereum/wiki at 6aee544ccc427490e443639ed29a1e4597cb898e, [Online]. Available: https://github.com/ethereum/wiki/tree/6aee544ccc427490e443639ed29a1e4597cb898e.
[89] (). Ethereum/eth2.0-specs at v0.9.0, [Online]. Available: https://github.com/ethereum/eth2.0-specs/tree/v0.9.0.
[90] (). Randao: A dao working as rng of ethereum, [Online]. Available: https://github.com/randao/randao.

APPENDIX

We attach the pseudocode of our evaluated proposals. Algorithm 7-10 are pseudocode of Elastico, Omniledger, RapidChain and Zilliqa, respectively. We do not present pseudocode for Monoxide or ETH 2.0, as their shard allocation protocols simply distribute nodes to shards according to their IDs' prefixes. We do not present pseudocode for Chainspace, as it does not specify details on how nodes join the system and nodes choose to move to other shards by themselves.

A. Modelling PoW.

PoW is frequently used in shard allocation protocols. We model PoW as follows. PoW consists of two algorithms $(\mathsf{PoWWork}, \mathsf{PoWVerify})$:

$\mathsf{PoWWork}(T, in) \rightarrow (nonce, out)$: A probabilistic algorithm. On input a difficulty parameter T and an input in, outputs a string $nonce$ and an output out.

$\mathsf{PoWVerify}(nonce, T, in) \rightarrow \{0/1\}$: A deterministic algorithm. On input $nonce$, T and in, outputs 0 (false) or 1 (true).

B. Elastico

In Elastico, a new block will trigger the state update, which triggers the shard allocation protocol. Elastico's shard allocation protocol employs a commit-then-reveal DRG protocol to produce randomness, and PoW to assign nodes to different shards. Elastico has a special shard called *final committee*, which is responsible for executing the DRG protocol. With a valid randomness as input, a node needs to run a PoW satisfying a specified difficulty parameter. The prefix of a valid PoW solution is ID of the shard that the node should join.

The DRG protocol works as follows. Let $n_s = \frac{n}{m}$ and $f_s = \frac{f}{m}$ be the number of nodes and faulty nodes in the final committee, respectively. First, each node in the final committee chooses a random string, then broadcasts its hash to others. The protocol assumes each node will receive $\geq \frac{2}{3}n_s$ hashes after broadcasting. Second, nodes execute a vector consensus [38] to agree on a set of hashes. The vector consensus works under synchronous networks, and has the *communication complexity* of $O({n_s}^{f_s})$. Third, each node broadcasts its original random string to other nodes, and the protocol assumes each node will have $\geq \frac{2}{3}n_s$ signed string/hash pairs. Last, each node can arbitrarily choose $\frac{1}{2}n_s + 1$ strings, then XOR them to get a valid randomness.

C. Omniledger

Similar to Elastico, Omniledger's shard allocation protocol is also constructed from DRG. In Omniledger, all nodes in the network jointly run *RandHound* [42] - a leader-based DRG protocol tolerating $\frac{1}{3}$ faulty nodes - to generate the randomness. *RandHound* adapts Publicly Verifiable Secret Sharing (PVSS) [79] to make the randomness publicly verifiable, and CoSi [36] to improve the communication and space complexity of generating multi-signatures. It has the *communication complexity* of $O(n)$, works under asynchronous network, and tolerates $\frac{1}{3}$ faulty nodes [42].

Algorithm 7: Elastico's shard allocation protocol.

Preliminaries:

- State is the blockchain, and state update is triggered by a new block.
- DRG is the commit-then-reveal DRG protocol.
- $st_t.T$ is the mining difficulty at state st_t.
- t is the block template for the miner to mine.

Algorithm Setup (λ) **:**

$m \leftarrow \lambda$

return m, st_0

Algorithm JoinShard (sk_i, pp, st_t) **:**

$m \leftarrow pp$

if i *is in final committee* **then**

Run DRG with peers in final committee to get a randomness as $st_t.R$

Broadcast $st_t.R$

else

Wait for a valid $st_t.R$

$nonce, out \leftarrow \mathsf{PoWWork}(st_t.T, st_t.R||t)$

$k \leftarrow out \mod m$

$\pi \leftarrow (nonce, st_t, t)$

return k, π

Algorithm UpdateShard $(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1})$ **:**

$(k', \pi, st_{t+1}) \leftarrow \mathsf{JoinShard}(sk_i, pp, st_{t+1})$

return k', π

Algorithm VerifyShard $(pk_i, st_t, k, \pi_{i,st_t,k})$ **:**

$m \leftarrow pp$

$(nonce, st_{t+1}, t) \leftarrow \pi_{i,st_t,k}$

if $st_{t+1} \neq st_t$ **then**

return 0

else if $\mathsf{PoWVerify}(nonce, st_t.T, st_t.R||t) == 0$ **then**

return 0

else if $k \neq out \mod m$ **then**

return 0

return 1

As *RandHound* is leader-based, nodes should elect a leader before running *RandHound*. In Omniledger, nodes run a VRF [44]-based cryptographic sortition to elect a leader. Each node first obtains the whole list of peers from the identity authority. Then, each node computes a ticket by running $\mathsf{VRFHash}(sk_i, "leader"||peers||v)$, where sk_i is its secret key, $peers$ is the list of peers, and v is a view counter starting from zero. Each node then broadcasts its ticket, and waits for a timeout Δ. After Δ, each node takes the one with smallest ticket as the leader, and the leader should start *RandHound*. If the leader does not start *RandHound* after another Δ, nodes will increase the view counter by 1, compute another ticket and broadcast it again.

Omniledger assumes the leader election is highly possible

Algorithm 8: Omniledger's shard allocation protocol.

Preliminaries:

- State is maintained and periodically updated by the identity authority I.
- RandHound is the RandHound DRG protocol.
- CoinToss is the asynchronous Coin Toss protocol [35].

Algorithm Setup (λ) **:**

$m \leftarrow \lambda$

return m, st_0

Algorithm JoinShard (sk_i, pp, st_t) **:**

Register pk_i on I

Get k, π from I

return k, π

Algorithm UpdateShard $(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1})$ **:**

$peers \leftarrow$ all peers stored in I at st_t

for $v \in [0, 5)$ **do**

$ticket_i \leftarrow \mathsf{VRFHash}(sk_i, "leader"||peers||v)$

Broadcast $ticket_i$

Collect tickets until $timer$ that timeouts in Δ

Find the smallest ticket as $ticket_j$

if $ticket_i == ticket_j$ **then**

$\pi_{ticket,i} \leftarrow \mathsf{VRFProve}(sk_i, "leader"||peers||v)$

Broadcast $\pi_{ticket,i}$

Start RandHound as a leader

generate the randomness from RandHound as $st_{t+1}.R$

break

else

Reset $timer$ (still timeouts in Δ)

Wait for $\pi_{ticket,j}$ within $timer$

Wait for pk_j to start

$st_{t+1}.R \leftarrow$ RandHound within $timer$

if *Receive* RandHound *message from* pk_j *before* Δ **then**

Execute $st_{t+1}.R \leftarrow \mathsf{RandHound}(\cdot)$

break

$v += 1$

if $st_t.R == None$ **then**

Run CoinToss with all peers to generate the randomness as $st_t.R$

Calculate a permutation p using $st_t.R$

$k' \leftarrow p[pk_i]$

return $k', \perp$

Algorithm VerifyShard $(pk_i, st_t, k, \pi_{i,st_t,k})$ **:**

Calculate a permutation p using $st_t.R$

return $p[pk_i] == k$

to succeed. However, if the sortition fails for five times, nodes quit the leader election as well as *RandHound*. Instead, nodes produce the randomness using an asynchronous coin-tossing protocol [35]. The coin-tossing protocol [35] does not scale due to high *communication complexity* of $O(n^3)$, but guarantees safety under asynchronous networks.

After generating a randomness, some new nodes can join the system, and some existing nodes will change their shards. Omniledger gradually swaps in newly joined nodes: for each state update, each shard can only swap in $\leq \frac{n}{m}$ nodes from pending nodes. Omniledger assumes a centralised identity authority to manage pending nodes. Upon a new randomness, the identity authority randomly selects some pending nodes to join the system.

Similar with Elastico, Omniledger shuffles existing nodes upon each randomness. From the identity authority, each node knows all other nodes in the network. Upon a new randomness, each node can permute an order on the list of peers. Each node then divides the permuted list of peers to equally sized intervals, and nodes in an interval belong to a shard.

D. RapidChain

In RapidChain, a node is required to find a valid PoW solution before joining a shard. RapidChain employs a DRG protocol only for preventing long range attacks, where one pre-computes PoW solutions in order to take advantage of consensus in the future. Nodes in a special shard called *reference committee* executes DRG periodically. The DRG protocol works as follows. First, each node in the *reference committee* chooses a random string and shares it to others using Feldman Verifiable Secret Sharing (VSS) [33]. Second, each node adds received shares together to a single string, then broadcasts it. Last, each node calculates the final randomness using Lagrange interpolation on received strings. Feldman VSS assumes a synchronous and complete network, as the VSS-share step requires each node to receive all shares from other nodes. Feldman VSS cannot tolerate any faults. If there is a node who fails to send shares to all other nodes, the protocol will restart. That is, a single faulty node can make the protocol to lose *liveness*.

RapidChain's shard allocation protocol employs the Commensal Cuckoo rule [34] to partition nodes into different shards. Each node is pseudorandomly mapped to a number in $[0, 1)$. The interval is then divided into smaller segments, and nodes within the same segment belong to the same shard. When a node a joins the network, it will "push forward" nodes in a constant-size interval surrounding a to other shards. This guarantees the load is balanced adaptively with new nodes joining.

Commensal Cuckoo works under asynchronous model. However, as Feldman VSS assumes synchrony, RapidChain's shard allocation protocol should assume synchrony to remain correct. Like Feldman VSS, Commensal Cuckoo assumes crash faults. As Commensal Cuckoo does not provide publicly verifiable membership, a Byzantine node can choose not to obey the protocol and stay in any shard freely.

Algorithm 9: RapidChain's shard allocation protocol.

Preliminaries:

- State is the blockchain, and state update is triggered by a new block.
- $st_t.T$ is the mining difficulty at state st_t.
- DRG is the Feldman VSS-based DRG protocol.
- C_r is the reference committee.

Algorithm Setup (λ)**:**
  $m \leftarrow \lambda$
  return m, st_0

Algorithm JoinShard (sk_i, pp, st_t)**:**
  $nonce, out \leftarrow$ PoWWork$(st_t.T, timestamp||pk_i||st_t.R)$
  Send $nonce, out$ to C_r
  C_r creates a list of all active nodes l at last state $st\_$
  C_r uses $st_t.R$ to randomly assign node i to a shard k
  C_r enforces nodes near node i to move to other shards
  return $k, \perp$

Algorithm UpdateShard $(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1})$**:**
  if *Node i is in C_r* **then**
    Run DRG with peers in C_r to get a randomness as $st_{t+1}.R$
  if *C_r asks node i to move to another shard k'* **then**
    Move to shard k'
  else
    $k' = k$
  return $k', \perp$

Algorithm VerifyShard $(pk_i, st_t, k, \pi_{i,st_t,k})$**:**
  return 1

E. Chainspace

Chainspace uses a smart contract called `ManageShards` to manage nodes' membership. Nodes can request to move to other shards by invoking transactions of `ManageShards`.

Note that `ManageShards` runs upon Chainspace itself. While the security of `ManageShards` relies on the whole system's security, the system's security relies on nodes. Meanwhile, nodes' membership rely on `ManageShards`, which leads to a chicken-and-egg problem. To avoid this chicken-and-egg problem, Chainspace assumes `ManageShards` executes correctly.

F. Monoxide

Monoxide's identity system is similar to Bitcoin. Nodes are free to create identities, and nodes are assigned to different shards according to their addresses' most significant bits (MSBs).

Unlike other protocols, Monoxide's shard allocation protocol does not seek to solve all problems in our formalisation.

Instead, it solves these problems by employing PoW-based consensus upon the shard allocation protocol. In PoW-based consensus, the voting power is decided by computing power (a.k.a. mining power), and Sybil attacks can no longer be profitable.

G. Zilliqa

Algorithm 10: Zilliqa's shard allocation protocol.

Preliminaries:
- State is the blockchain, and state update is triggered by a new block.
- $st_t.R$ and $st_t.T$ are the hash of the last block and the mining difficulty at state st_t, respectively.
- t is the block template for the miner to mine.

Algorithm $\mathsf{Setup}\,(\lambda)$**:**
- $m \leftarrow \lambda$
- **return** m, st_0

Algorithm $\mathsf{JoinShard}\,(sk_i, pp, st_t)$**:**
- $m \leftarrow pp$
- $nonce, out \leftarrow \mathsf{PoWWork}(st_t.T, st_t.R||t)$
- $k \leftarrow out \mod m$
- $\pi \leftarrow (nonce, st_t, t)$
- **return** k, π

Algorithm $\mathsf{UpdateShard}\,(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1})$**:**
- $(k', \pi, st_{t+1}) \leftarrow \mathsf{JoinShard}(sk_i, pp, st_{t+1})$
- **return** k', π

Algorithm $\mathsf{VerifyShard}\,(pk_i, st_t, k, \pi_{i,st_t,k})$**:**
- $m \leftarrow pp$
- $(nonce, st_{t+1}, t) \leftarrow \pi_{i,st_t,k}$
- **if** $st_{t+1} \neq st_t$ **then**
  - **return** 0
- **else if** $\mathsf{PoWVerify}(nonce, st_t.T, st_t.R||t) = 0$ **then**
  - **return** 0
- **else if** $k \neq out \mod m$ **then**
  - **return** 0
- **return** 1

Zilliqa [80] is a permissionless sharded blockchain that claims to achieve the throughput of over 2,828 transactions per second. It follows the design of Elastico [9], but with several optimisations. Our evaluation is based on Zilliqa's whitepaper [14], Zilliqa's developer page [81], and Zilliqa's source code (the latest stable release v5.0.1) [82].

Different from Elastico which runs a DRG to generate randomness, Zilliqa simply uses the SHA2 hash of the latest block as randomness. Taking the randomness as input, each node generates two valid PoW solutions. Each node should solve two PoW puzzles within a time window of 60 seconds, otherwise he cannot join any shard for this epoch. This means propagating PoW solutions should finish within a time bound, which implicitly assumes synchronous network. The first PoW is used for selecting nodes to form the final committee, and the second PoW is used for distributing the rest nodes to other committees. The final committee is responsible for collecting nodes in the network and helping nodes find their peers in the same shards.

H. Ethereum 2.0

Ethereum 2.0 (ETH 2.0) is the next generation of Ethereum [83]. ETH 2.0 aims at achieving better performance by sharding, better privacy by using zkSNARKs, and better energy efficiency by switching from PoW to PoS. There is neither official whitepaper nor implementation for ETH 2.0. Instead, ETH 2.0 is still under active development, and its design is disaggregated in their wiki [19] and blogs [84]–[87], and the Ethereum community provides the specification [15] without having a technical whitepaper first. Our analysis is based on the latest (29/10/2019) public official materials, including the sharding FAQ of Ethereum Wiki branch 6aee544ccc427490e443639ed29a1e4597cb898e [88] and the ETH 2.0 specification v0.9.0 Tonkatsu [89].

Following ETH1.0, ETH 2.0 has two types of accounts: externally-owned account for wallets and smart contract account for smart contracts. Each account has an unique ID, and accounts are assigned to different shards according to their IDs. A node should use an externally-owned account to mine Ether, the native cryptocurrency of Ethereum.

Like Monoxide, ETH 2.0 addresses Sybil attacks and load balance in the consensus layer. More specifically, ETH 2.0 plans to employ Proof-of-Stake (PoS)-based consensus. In PoS-based consensus, the voting power is decided by cryptocurrency deposits a.k.a. staking power. As holding cryptocurrency deposits can be expensive, Sybil attacks can no longer be profitable.

Note that ETH 2.0 also employs a DRG protocol (called RANDAO [90]), but it is used for sampling "validators" (a subset of nodes that can produce blocks) rather than shard allocation.