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
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 limit 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.

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 $\alpha_t$ and $\beta_t$ 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 $F(\text{in}) \rightarrow \text{out}$ denote the assignment of the output of $F(\text{in})$ to out. Let MSB($z$, $\text{str}$) and LSB($z$, $\text{str}$) be the $z$ most significant bits and least significant bits of $\text{str}$, respectively.

Table I: Summary of notations.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$st_t$</td>
<td>System state at round $t$</td>
</tr>
<tr>
<td>$m$</td>
<td>Number of shards</td>
</tr>
<tr>
<td>$n^t$</td>
<td>Number of nodes in the network at round $t$</td>
</tr>
<tr>
<td>$n^k_t$</td>
<td>Number of nodes in shard $k$ at round $t$</td>
</tr>
<tr>
<td>$\alpha_t, \beta_t$</td>
<td>Average percentage of nodes joining and leaving the system at round $t$, respectively, ($\beta_t \in [0, 1]$)</td>
</tr>
<tr>
<td>$pp$</td>
<td>Public parameter</td>
</tr>
<tr>
<td>$sk_i, pk_i$</td>
<td>Secret key and public key of node $i$</td>
</tr>
<tr>
<td>$\pi_{i, st_t, k}$</td>
<td>Proof of assigning node $i$ to shard $k$ at round $t$</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Probability of a node to stay at its shard after $UpdateShard(\cdot)$ ($\gamma \in [0, 1]$)</td>
</tr>
</tbody>
</table>

Table II 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

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

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

2
When node $i$ joins the system at state $s_{ti}$, node $i$ is allocated to a shard according to the output of executing $S.A.JOINShard()$. It updates its shard allocation when the state changes from $s_{t}$ to $s'_{t}$ by executing $S.A.UPDATEShard()$. Given a shard $k$, public key $pk_i$, and a proof $\pi_{i,s_{ti},k}$, anyone can execute $S.A.VERIFYShard$ to verify whether node $i$ is allocated into shard $k$ at state $s_{ti}$. Algorithm 1 describes the typical process of shard allocation.

Algorithm 1: The shard allocation process for node $i$.

// Join the system at state $s_{ti}$
$(k_i, \pi_{i,s_{ti},k_i}) \leftarrow S.A.JOINShard(sk_i, pp, s_{ti})$

// Assign last state and shard
$s_{now}, k_{now}, \pi_{now} \leftarrow s_{ti}, k_i, \pi_{i,s_{ti},k_i}$

repeat
    Wait for a new state $s_{t}$
    // Update shard allocation
    $(k_{t}, \pi_{i,s_{t},k_{t}}) \leftarrow S.A.UPDATEShard(sk_i, pp, s_{now}, k_{now}, \pi_{i,s_{now},k_{now}}, s_{t})$
    // Reassign last state and shard
    $s_{now}, k_{now}, \pi_{now} \leftarrow s_{t}, k_{t}, \pi_{i,s_{t},k_{t}}$

until node $i$ leaves the system

B. Correctness

We consider three correctness properties for shard allocation protocols, namely liveliness, allocation-randomness, and unbiasedness. 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 liveliness of the underlying system, i.e., the underlying system should always make progress in updating the system state. We adopt 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 $S.A$ satisfies $(\tau, s)$-liveness if there are at least $[\tau \cdot s]$ 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.

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

$$Pr[k = k'] S.A.JOINShard(sk_i, pp, s_{ti}) \leftarrow (k', \pi_{i,s_{ti},k'}) = \frac{1}{m} + \epsilon$$

where $k, k' \in [1, m]$, and $\epsilon$ 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 $\gamma$. Similar to the concern discussed in the join-randomness, we define update-randomness as follows.

Definition 4 (Update-randomness). A shard allocation protocol $S.A$ with $m$ shards satisfies update-randomness if 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 $s_{ti}$ to $k''$ at state $s_{ti+1}$ is

$$Pr[k'' = k'] S.A.UPDATEShard(sk_i, pp, s_{ti}, k, \pi_{i,s_{ti},k}, s_{ti+1}) \leftarrow (k', \pi_{i,s_{ti+1},k'}) \triangleq \begin{cases} 2 \pm \epsilon & \text{if } k' = k, \\ \gamma \frac{m-1}{m-1} \pm \epsilon & \text{otherwise} \end{cases}$$

Figure 2: Update-randomness. After executing $S.A.UPDATEShard()$, the probability that a node stays in its shard (say shard 1) is $\gamma$, and the probability of moving to each other shard is $\frac{1 - \gamma}{m - 1}$.
where \( k', k'' \in [1, m] \), and \( \epsilon \) is a negligible value.

When \( \gamma = \frac{1}{m} \), 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.

**Unbiasability.** The unbiasedness 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, unbiasedness considers the possibility of Byzantine nodes manipulating the protocol.

**Definition 6 (Unbiasedness).** A shard allocation protocol SA satisfies unbiasedness iff given a system state, no node can manipulate the probability distribution of the resulting shard of SA.JoinShard(\( \cdot \)) or SA.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 SA with \( m \) shards provides join-privacy iff for any secret key \( sk_i \), public parameter \( pp \), and state \( st_i \), without the knowledge of \( \pi_i, st_{i+1}, k \) and \( sk_i \), the probability of making a correct guess \( k' \) on \( k \) is

\[
Pr[k' = k | \langle k, \pi_i, st_i, k \rangle \leftarrow \text{SA.JoinShard}(sk_i, pp, st_i)] = \frac{1}{m} \pm \epsilon
\]

where \( k, k' \in [1, m] \), and \( \epsilon \) is a negligible value.

**Definition 8 (Update-privacy).** A shard allocation protocol 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 \( st_i \) and \( st_{i+1} \), without the knowledge of \( \pi_i, st_{i+1}, k \) and \( sk_i \), the probability of making a correct guess \( k'' \) on \( k' \) is

\[
Pr[k'' = k' | \langle k', \pi_{i+1}, st_{i+1}, k' \rangle \leftarrow \text{SA.UpdateShard}(sk_{i+1}, pp, st_{i+1}, k, \pi_i, st_i, k, st_{i+1})] = \begin{cases} 
\gamma \pm \epsilon & \text{if } k'' = k \\
\frac{1-\gamma}{m-1} \pm \epsilon & \text{otherwise}
\end{cases}
\]

where \( k', k'' \in [1, m] \), and \( \epsilon \) is a negligible value.

**Definition 9 (Allocation-privacy).** A shard allocation protocol 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 \( \omega \)-self-balanced iff for the largest possible \( \omega \in [0, 1] \), we have

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

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 \( \Gamma \) 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 \( \gamma \), then its operability is \( \Gamma = \gamma \), i.e., \( \gamma \)-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. Elastico [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 proposals 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
### Correctness

**Public verifiability.** All of these shard allocation protocols achieve public verifiability except for RapidChain. RapidChain’s shard allocation is not public verifiably, as the deployed Commensal Cuckoo protocol is not public verifiably. 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 shard blockchain designs do not specify parameters, we cannot determine the actual \( \tau \) and \( s \) for their liveness. Thus, we consider a shard allocation protocol that does not satisfy liveness when there is 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 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.

### Performance metrics

<table>
<thead>
<tr>
<th>System model</th>
<th>Correctness</th>
<th>Performance metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Network model</strong></td>
<td><strong>Public verifiability</strong></td>
<td><strong>Join comm. compl.</strong></td>
</tr>
<tr>
<td><strong>Trust assumption</strong></td>
<td><strong>Liveness</strong></td>
<td><strong>Update comm. compl.</strong></td>
</tr>
<tr>
<td><strong>Fault-tolerance</strong></td>
<td><strong>Unbiasibility</strong></td>
<td><strong>Sybil cost</strong></td>
</tr>
<tr>
<td><strong>Allocation-randomness</strong></td>
<td><strong>Privacy</strong></td>
<td><strong>Safe-transaction</strong></td>
</tr>
<tr>
<td><strong>Operability</strong></td>
<td></td>
<td><strong>Operability</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Elastico</td>
<td>New block</td>
<td>Sync.</td>
<td>-</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
</tr>
<tr>
<td>Omniledger</td>
<td>Identity authority</td>
<td>Part. sync.</td>
<td>Identity authority</td>
<td>-</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
</tr>
<tr>
<td>RapidChain</td>
<td>Nodes joining</td>
<td>Sync.</td>
<td>-</td>
<td>0</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
</tr>
<tr>
<td>Chainspace</td>
<td>-</td>
<td>Async.</td>
<td>Smart contracts</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
</tr>
<tr>
<td>Monoxide</td>
<td>-</td>
<td>Async.</td>
<td>-</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
</tr>
<tr>
<td>Zilliqa</td>
<td>New block</td>
<td>Async.</td>
<td>-</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
</tr>
<tr>
<td>ETH 2.0</td>
<td>-</td>
<td>Async.</td>
<td>-</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
<td>1</td>
<td>( \frac{m}{n} )</td>
<td>( \frac{n}{m} )</td>
</tr>
</tbody>
</table>

**WORMHOLE (Or our proposal)** | New rand. | Async. | Rand. Beacon | 1 | \( \frac{m}{n} \) | \( \frac{n}{m} \) | \( \frac{m}{n} \) | \( \frac{n}{m} \) | \( \frac{m}{n} \) | \( \frac{n}{m} \) | 1 | \( \frac{m}{n} \) | \( \frac{n}{m} \) | \( \frac{m}{n} \) | \( \frac{n}{m} \) |

- Optional. * Protected by PoW puzzles. ** Protected by Sybil-resistant consensus protocols.

**Unbiasability.** Elastico and Zilliqa do not fully achieve unbiasedness. 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 unbiasedness, as it does not satisfy allocation-randomness and nodes are free to choose shards. Omniledger, RapidChain, Monoxide and ETH 2.0 satisfy unbiasedness.

**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 JoinShard(\( \cdot \)) and UpdateShard(\( \cdot \)) is \( O(n^2) \), where \( n \) and \( f \) are the number of nodes and faulty nodes in the network, respectively. In JoinShard(\( \cdot \)) and 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 \( O(n^2) \). Ideally, the final committee consists of \( \frac{n}{m} \) nodes, so the communication complexity of JoinShard(\( \cdot \))
and $\text{UpdateShard}(\cdot)$ is $O\left(\frac{n^2}{m}\right) = O\left(\frac{n^2}{m}\right) = O(n^f)$ ($m$ is constant). The communication complexity of Omniledger’s $\text{JoinShard}(\cdot)$ and $\text{UpdateShard}(\cdot)$ 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 $\text{UpdateShard}(\cdot)$ 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 $\text{JoinShard}(\cdot)$ and $\text{UpdateShard}(\cdot)$ is $O(n^2)$. This is from Feldman VSS, which has the communication complexity of $O\left(\frac{n^2}{m}\right) = O(n^2)$ [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 join the system, and no node will move to other shards. Thus, self-balance is $\frac{n - \Delta t}{n} = 1 - \beta_t$.

**Operability.** The operability of Elastico, Omniledger and Zilliqa is $\frac{1}{m}$, as all nodes are forced to change their shards for each new round. The operability of RapidChain is $\max(1 - \kappa n, 0)$, where $\kappa \in [0, 1]$ is the size of the interval in which nodes should move to other shards, and $\alpha_t$ 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 $\kappa n$ other nodes. The operability then becomes $1 - \frac{\kappa n}{m} = 1 - \kappa n$. However, in reality the operability cannot be smaller than 0. Thus, the operability is $\max(1 - \kappa n, 0)$. 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 III 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 $\gamma$ (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 $\gamma$, the self-balance $\omega$ of $\mathcal{SA}$ is

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

where $\beta_t$ 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 $n^t_k$ 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 $\Delta n^t_k$ be the number of leaving nodes in shard $k \in [1, m]$ at round $t$, we have $\sum_{k=1}^m \Delta n^t_k = \beta_t n^t$. Upon the next system state $s_{t+1}$, each node executes $\mathcal{SA}.\text{UpdateShard}(\cdot)$, and its resulting shard complies with the probability distribution in Definition 4. After executing $\mathcal{SA}.\text{UpdateShard}(\cdot)$, 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 $\gamma (n^t_k - \Delta n^t_k)$ nodes in shard $k$ that do not move to other shards. There are

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

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

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

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

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

(1)

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

(2)
By Definition 10 to find the largest $\omega$, we should find the largest $\left|\frac{n_{i+1} - n_{j+1}}{n}\right|$, which can be calculated as

$$\frac{|n_{i+1} - n_{j+1}|}{n} = \frac{\frac{2m-1}{m-1}(n_i - \Delta n_i) - \frac{2m-1}{m-1}(n_j - \Delta n_j)}{n}$$

(3)

$$= \frac{\frac{2m-1}{m-1}(\Delta n_i - \Delta n_j)}{n}$$

(4)

When $(\Delta n_i - \Delta n_j)$ is maximal, $|\frac{n_{i+1} - n_{j+1}}{n}|$ is maximal, and $\omega$ is also maximal. As there are $\beta n^2$ nodes leaving the network in total, the maximal value of $(\Delta n_i - \Delta n_j)$ is $\beta n^2$. When $\Delta n_i - \Delta n_j = \beta n^2$,

$$\omega = 1 - \frac{|n_{i+1} - n_{j+1}|}{n} = 1 - \frac{\frac{2m-1}{m-1}(\Delta n_i - \Delta n_j)}{n}$$

(5)

$$= 1 - \frac{\frac{2m-1}{m-1}\beta n^2}{n} = 1 - \left(\frac{\gamma m - 1}{m - 1}\right)\beta_i$$

(7)

Figure 3 visualises the relationship between operability and self-balance. We pick $m = 1000$ and $\beta_i = 0.0005$ as an example.

**Theorem 1.** Let $\beta_i$ 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 $\beta_i \neq 0$ 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 $\beta_i = 0$ 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 $\beta_i = 0$. However, $\beta_i > 0$, which leads to a contradiction. □

**Parameterising self-balance and operability.** As shown in Figure 3 $\gamma = 1$ and $\theta_i = \frac{1}{\beta_i}$ are two extreme points on the line of relationship between self-balance and operability. Shard allocation protocols relying 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

$$\mathcal{SA}.\text{UpdateShard}(sk_i, pp, st_i, k, \pi_{i,st_i,k}, st_{i+1})$$

depends on system states earlier than $st_i$. As both self-balance and operability are related to the probability $\gamma$ (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 self-balanced and operable where $\omega \in (0, 1 - \beta_i)$ and $\gamma \in (\frac{1}{m - 1}, 1)$, then $\mathcal{SA}$ is non-memoryless.

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

When $\gamma \in (\frac{1}{m - 1}, 1)$, the distribution of the resulting shard of $\mathcal{SA}.\text{UpdateShard}(\cdot)$ is non-uniform, given the update-randomness property. In this case, executing $\mathcal{SA}.\text{UpdateShard}(\cdot)$ requires the knowledge of $k$ — index of the shard that $i$ locates at state $st_i$. Thus, $\pi_{i,st_{i+1},k'}$ — one of the output of $\mathcal{SA}.\text{UpdateShard}(\cdot)$ — should enable verifiers to verify node $i$ is at shard $k$ at state $st_i$.

“Verify node $i$ is at shard $k$ at state $st_i”$ is achieved by verifying $\pi_{i,st_i,k}$. This means $\pi_{i,st_{i+1},k'}$ depends on $st_i$ and $\pi_{i,st_i,k}$. Similarly, $\pi_{i,st_{i+2},k}$ depends on $st_i$ and $\pi_{i,st_{i+1},k}$. Let $\pi_{i,st_{i-2},k}$ and $\pi_{i,st_{i-1},k}$ and $\pi_{i,st_{i-1},k}$ and $\pi_{i,st_{i-2},k}$. Recursively, $\pi_{i,st_{i+2},k}$
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, \( \text{S.A.} \cdot \text{UpdateShard}(\cdot) \) does not need to depend on any prior system state. When \( \gamma = \frac{1}{m} \), the distribution of the resulting shard of \( \text{S.A.} \cdot \text{UpdateShard}(\cdot) \) is uniform, so \( \text{S.A.} \cdot \text{UpdateShard}(\cdot) \) can just assign nodes randomly according to the incoming system state. When \( \gamma = 1 \), the resulting shard of \( \text{S.A.} \cdot \text{UpdateShard}(\cdot) \) is certain, and \( \text{S.A.} \cdot \text{UpdateShard}(\cdot) \) 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]–[42]) 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 (VRFKeyGen, VRFHash, VRFProve and VRFVerify).

- VRFKeyGen(\( \lambda \)) \( \rightarrow (sk,pk) \): On input a security parameter \( \lambda \), outputs the secret/public key pair \((sk,pk)\).
- VRFHash\((sk,str)\) \( \rightarrow h \): On input \( sk \) and an arbitrary-length string \( str \), outputs a fixed-length hash \( h \).
- VRFProve\((sk,str)\) \( \rightarrow \pi \): On input \( sk \) and \( str \), outputs the proof \( \pi \) for \( h \).
- VRFVerify\((pk,str,h,\pi)\) \( \rightarrow \{0,1\} \): On input \( pk \), \( str \), \( h \), \( \pi \), 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 \( str \), VRFHash\((sk,str)\) produces a unique valid output.
- VRF-Collision-Resistance: It is computationally hard to find two inputs \( str \) and \( str' \) that VRFHash\((sk,str)\) = VRFHash\((sk,str')\).
- VRF-Pseudorandomness: It is computationally hard to distinguish the output of 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 \( m = 16^3 \).](image)

WORMHOLE relies on a secure randomness beacon \( RB \). Each new random output from \( RB \) 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 $\ell$ random outputs, where $\ell \in [sr, 2sr)$. In other words, for each shard allocation, a node needs to travel through $\ell$ 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 $\ell$ 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 calculateShard** ($m, op, h_{t-\ell+1}, \ldots, h_t$):

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

C. Detailed construction

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

**Algorithm Setup** ($\lambda$):

- return ($m, op, sr$)

Joining a shard (Algorithm 4) A node $i$ executes $\text{JoinShard}(sk_i, pp, st_t)$ in order to obtain a membership of a shard at round $t$. First, node $i$ calculates VRF hashes and proofs of the latest $\ell$ system states, where $\ell = t \mod sr + sr$. Node $i$ calculates the ID $k$ of the shard it belongs to by $\text{calculateShard}(\cdot)$ (Algorithm 2). The proof $\pi_{i,st_t,k}$ that node $i$ has a valid membership in shard $k$ at state $st_t$ 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 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
  - $h_j \leftarrow \text{VRFHash}(sk_i, st_j)$
  - $\pi_j \leftarrow \text{VRFProve}(sk_i, st_j)$
- $k \leftarrow \text{calculateShard}(m, op, h_{t-\ell+1}, \ldots, h_t)$
- $\pi_{i,st_t,k} \leftarrow (pk_i, h_{t-\ell+1}, \ldots, h_t, \pi_{t-\ell+1}, \ldots, \pi_t)$
- 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 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 $\pi_j$ from memory
- $h_{t+1} \leftarrow \text{VRFHash}(sk_i, st_{t+1})$
- $\pi_{t+1} \leftarrow \text{VRFProve}(sk_i, st_{t+1})$
- $k' \leftarrow \text{calculateShard}(m, op, h_{t-\ell'+2}, \ldots, h_t)$
- $\pi_{i,st_{t+1},k'} \leftarrow (pk_i, h_{t-\ell'+2}, \ldots, h_{t+1}, \pi_{t-\ell'+2}, \ldots, \pi_{t+1})$
- 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 VRFVerif$(\cdot)$ to check the validity of all $\ell$ VRF hashes and VRF proofs, and executes $\text{calculateShard}(\cdot)$ 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 RB generates $\lceil \tau \cdot s \rceil$ random outputs in any $s$ round (where $\tau \in \mathbb{R}_{>0}$ and $s \in \mathbb{N}_{>0}$), then WORMHOLE satisfies $(\tau, s)$-liveness.

**Proof.** We prove this by contradiction. Assuming that WORMHOLE does not satisfy $(\tau, s)$-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_j, k, π_i, st_i, k):
  m, op, sr ← pp
  τ ← t mod sr + sr
  (pk_i, h_{last}, ..., h_t, π_{last}, ..., π_t) ← π_i, st_i, k
  if last ≠ τ + 1 then
    return 0
  for j ∈ [t − ℓ + 1, t] do
    if VRFVerify (pk_i, st_j, h_j, π_j) = 0 then
      return 0
  if k ≠ calculateShard (m, op, h_{t−1}, ..., h_t) then
    return 0
  return 1

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

Lemma 3. WORMHOLE satisfies unbiasability.

Proof. We prove this by contradiction. Assuming that WORMHOLE does not satisfy unbiasability, given a system state, an adversary can manipulate the probability distribution of the output shard of JoinShard(·) or UpdateShard(·) with non-negligible probability. This consists of three attack vectors: 1) the adversary can manipulate the system state; 2) when $SA$ JoinShard(·) or $SA$ UpdateShard(·) 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-Unbiasability, the randomness produced by RB is unbiased, so the system state of WORMHOLE is unbiased. 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 unbiased system state indicates that the VRF hash is unbiased. The output shard of JoinShard(·) or UpdateShard(·) is a modulus of the VRF hash, which is also unbiased. This eliminates the first attack vector. Thus, if WORMHOLE does not resist against the first attack vector, then this contradicts RB-Unbiasability; 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 $\epsilon$. Running JoinShard(·) requires the execution of VRFHash(·) 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 JoinShard(·) 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 $\gamma$ 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 UpdateShard(·) 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 WORMHOLE is a correct shard allocation protocol.

Performance metrics The communication complexity of JoinShard(·) and UpdateShard(·) of WORMHOLE are $O(1)$. To execute JoinShard(·), the only communication that a node should make is to receive $\ell$ random outputs, where $\ell = t \mod sr + sr$. As $sr$ is a constant, the communication complexity of JoinShard(·) is $O(1)$. To execute UpdateShard(·), a node only need to receive the latest randomness, and the communication complexity of UpdateShard(·) is also $O(1)$. WORMHOLE resists Sybil attacks using computation. To execute JoinShard(·), a node should executes VRFHash for $\ell$ times, where $\ell = t \mod sr + sr$. One can make WORMHOLE more Sybil-resistant by making VRFHash(·) more computation-intensive. WORMHOLE’s operability $\gamma$ 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 \( \omega \) of WORMHOLE is \( 1 - \beta_t + \frac{\beta_t}{2^{op}} \), where \( \beta_t \) is the percentage of nodes that leave from the system at round \( t \). By Definition 1,

\[
\omega = 1 - \frac{(\gamma m - 1)\beta_t}{m} 
\]

(8)

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

(9)

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

(10)

\[
\omega = 1 - (1 - \frac{1}{2^{op}})\beta_t 
\]

(11)

\[
\omega = 1 - \beta_t + \frac{\beta_t}{2^{op}} 
\]

(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 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 JoinShard(\( \cdot \)) and 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 repairable; 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 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 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 ~ 640 KB, and verify the proof by running 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 \( \sim 85 \) 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], [60]. The main
challenge is to balance the computation, communication, and storage workload. Despite a large number of load-balancing algorithms [5, 6, 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.

## 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.

<table>
<thead>
<tr>
<th></th>
<th>System model</th>
<th>Correctness</th>
<th>Performance metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elastico</td>
<td>New block</td>
<td>Async.</td>
<td>1</td>
</tr>
<tr>
<td>Omniledger</td>
<td>Identity auth</td>
<td>Async.</td>
<td>1</td>
</tr>
<tr>
<td>RapidChain</td>
<td>Nodes joining</td>
<td>Async.</td>
<td>1</td>
</tr>
<tr>
<td>Zilliqa</td>
<td>New block</td>
<td>Async.</td>
<td>1</td>
</tr>
</tbody>
</table>

|                      | Meanings of colours are same as Table II. ★ means the metric is improved by replacing DRG with a randomness beacon. |

|                      | New rand | Async. | Rand. Beacon | 1 | ★ | ★ | ★ | ★ | 0★ | 0★ | Comp | 1 | 1 |

**REFERENCES**


We attach the pseudocode of our evaluated proposals. Algorithm 10 are pseudocode of Elastico, Omniledger, Rapid-Chain 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 (PoWWork, PoWVerify):

\[
\text{PoWWork}(T, in) \rightarrow (\text{nonce, out}) \quad \text{A probabilistic algorithm. On input a difficulty parameter } T \text{ and an input } \text{in, outputs a string } \text{nonce and an output } \text{out.}
\]

\[
\text{PoWVerify}(\text{nonce, T, in}) \rightarrow \{0/1\} \quad \text{A deterministic algorithm. On input } \text{nonce, T and in, outputs 0 (false) or 1 (true).}
\]

**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 \( O(n_s^2) \). 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 strings/has 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 \( \frac{2}{3} \) faulty nodes - to generate the randomness. **RandHound** adapts Publicly Verifiable Secret Sharing (PVSS) [79] to make the randomness publicly verifiable, and CoSi [50] 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 (\( \lambda \)):

\[
m \leftarrow \lambda \\
\text{return } m, st_0
\]

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

\[
m \leftarrow pp \\
\text{if } i \text{ is in final committee then} \\
\quad \text{Run DRG with peers in final committee to get a randomness as } st_t.R \\
\quad \text{Broadcast } st_t.R \\
\text{else} \\
\quad \text{Wait for a valid } st_t.R \\
nonce, out \leftarrow \text{PoWWork}(st_t.T, st_t.R || t) \\
k \leftarrow out \mod m \\
\pi \leftarrow (nonce, st_t, t) \\
\text{return } k, \pi
\]

Algorithm VerifyShard (\( pk_i, st_t, k, \pi_i, st_{t+1}, k \)):

\[
m \leftarrow pp \\
(nonce, st_{t+1}, t) \leftarrow \pi_i.st_{t+1} \\
\text{if } st_{t+1} \neq st_t \text{ then} \\
\quad \text{return } 0 \\
\text{else if } \text{PoWVerify}(nonce, st_t.T, st_t.R || t) == 0 \text{ then} \\
\quad \text{return } 0 \\
\text{else if } k \neq out \mod m \text{ then} \\
\quad \text{return } 0 \\
\text{return } 1
\]

As RandHound is leader-based, nodes should elect a leader before running RandHound. In Omniledger, nodes run a VRF-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 \( \text{VRFHash}(sk_i, "leader" || peers || v) \), 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 \( \Delta \). After \( \Delta \), 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 \( \Delta \), 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 \cite{35}.

Algorithm Setup (\( \lambda \)):

\[
m \leftarrow \lambda \\
\text{return } m, st_0
\]

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

- Register \( pk_i \) on \( I \)
- Get \( k, \pi \) from \( I \)
- \text{return } k, \pi

Algorithm UpdateShard (\( sk_i, pp, st_t, k, \pi_i, st_{t+1}, k \)):

\[
\text{peers} \leftarrow \text{all peers stored in } I \text{ at } st_t \\
\text{for } v \in [0, 5] \text{ do} \\
\quad ticket_i \leftarrow \text{VRFHash}(sk_i, "leader" || peers || v) \\
\quad \text{Broadcast } ticket_i \\
\quad \text{Collect tickets until } timer \text{ that timeouts in } \Delta \\
\quad \text{Find the smallest ticket as } \text{ticket}_j \\
\quad \text{if } \text{ticket}_i == \text{ticket}_j \text{ then} \\
\quad \quad \pi_{\text{ticket}_i} \leftarrow \text{VRFProve}(sk_i, "leader" || peers || v) \\
\quad \quad \text{Broadcast } \pi_{\text{ticket}_i} \\
\quad \quad \text{Start RandHound as a leader} \\
\quad \quad \text{generate the randomness from RandHound as } st_{t+1}.R \\
\quad \quad \text{break} \\
\quad \text{else} \\
\quad \quad \text{Reset } \text{timer} \text{ (still timeouts in } \Delta) \\
\quad \quad \text{Wait for } \pi_{\text{ticket}_i} \text{ within } timer \\
\quad \quad \text{Wait for } pk_i \text{ to start} \\
\quad \quad st_{t+1}.R \leftarrow \text{RandHound within } timer \\
\quad \quad \text{if } \text{Receive RandHound message from } pk_i \text{ before } \Delta \text{ then} \\
\quad \quad \quad \text{Execute } st_{t+1}.R \leftarrow \text{RandHound(·)} \\
\quad \quad \quad \text{break} \\
\quad \text{v}+1 = 1 \\
\text{if } st_t.R == \text{None} \text{ then} \\
\quad \text{Run CoinToss with all peers to generate the randomness as } st_t.R \\
\quad \text{Calculate a permutation } p \text{ using } st_t.R \\
\quad k' \leftarrow p[pk_i] \\
\quad \text{return } k', \bot
\]

Algorithm VerifyShard (\( pk_i, st_t, k, \pi_i, st_{t+1}, k \)):

- Calculate a permutation \( p \) using \( st_t.R \)
- \text{return } p[pk_i] == k
committees choose a random string and share it to others executes DRG periodically. The DRG reference committee consensus in the future. Nodes in a special shard called pre-computes PoW solutions in order to take advantage of protocol only for preventing long range attacks, where one solution before joining a shard. RapidChain employs a DRG D. RapidChain node then divides the permuted list of peers to equally sized each node can permute an order on the list of peers. Each knows all other nodes in the network. Upon a new randomness, upon each randomness. From the identity authority, each node knows 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). 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 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 an 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.
- \( st_t \) is the mining difficulty at state \( st_t \).
- DRG is the Feldman VSS-based DRG protocol.
- \( C_r \) is the reference committee.

Algorithm Setup \((\lambda)\):
\[
m \leftarrow \lambda
\]
return \((m, s_0)\)

Algorithm JoinShard \((s_{k_i}, pp, st_t)\):
\[
nonce, out \leftarrow PoWWork(st_t, T, timestamp||pk_{i}||st_t.R)
\]
Send \((nonce, out)\) to \( C_r \)
\( C_r \) creates a list of all active nodes \( l \) at last state \( st_l \)
\( C_r \) 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 \((k, \bot)\)

Algorithm UpdateShard \((s_{k_i}, pp, st_t, k, \pi_{i, st_t, k}, st_{t+1})\):
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 \((k', \bot)\)

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 Setup (\( \lambda \)) :**

\[
\text{return } m, st_0
\]

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

\[
\begin{align*}
m & \leftarrow \text{pp} \\
nonce, out & \leftarrow \text{PoWWork}(st_t.T, st_t.R | t) \\
k & \leftarrow out \mod m \\
\pi & \leftarrow (nonce, st_t, t) \\
\text{return } k, \pi
\end{align*}
\]

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

\[
\begin{align*}(k', \pi, st_{t+1}) & \leftarrow \text{JoinShard}(sk_i, pp, st_{t+1}) \\
\text{return } k', \pi
\end{align*}
\]

**Algorithm VerifyShard \( (pk_i, st_t, k, \pi_i, st_t, k) \) :**

\[
\begin{align*}m & \leftarrow \text{pp} \\
(nonce, st_{t+1}, t) & \leftarrow \pi_i, st_t, k \\
\text{if } st_{t+1} & \neq st_t \\
\text{then } & \text{return 0} \\
\text{else if } \text{PoWVerify}(nonce, st_t.T, st_t.R || t) & = 0 \\
\text{then } & \text{return 0} \\
\text{else if } k & \neq out \mod m \\
\text{then } & \text{return 0} \\
\text{return 1}\end{align*}
\]

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 6aee544ccc427490e44369ed29a1e4597cb898e [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.