Analysing and Improving Shard Allocation Protocols for Sharded Blockchains

Runchao Han
Monash University and CSIRO-Data61
runchao.han@monash.edu

Jiangshan Yu*
Monash University
jiangshan.yu@monash.edu

Ren Zhang
Nervos Foundation
ren@nervos.org

Abstract

Sharding is a promising approach to scale permissionless blockchains. In a sharded blockchain, participants are split into groups, called shards, and each shard only executes part of the workloads. Despite its wide adoption in permissioned systems, transferring such success to permissionless blockchains is still an open problem. In permissionless networks, participants may join and leave the system at any time, making load balancing challenging. In addition, there exists Byzantine participants, who may launch various attacks on the blockchain. To address these issues, participants should be securely and dynamically allocated into different shards uniformly. However, the protocol capturing such functionality – which we call *shard allocation* – is overlooked.

In this paper, we study shard allocation protocols for permissionless blockchains. We formally define the shard allocation protocol and propose an evaluation framework. We then apply the framework to evaluate the shard allocation subprotocols of seven state-of-the-art sharded blockchains, and show 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 identify a fundamental property *non-memorylessness* that enables parametrisation on this trade-off. Based on these insights, we propose WORMHOLE, a correct and efficient shard allocation protocol with minimal security assumptions and parameterisable *self-balance* and *operability*.

1 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 [74], file systems [59], and permissioned blockchains [43].

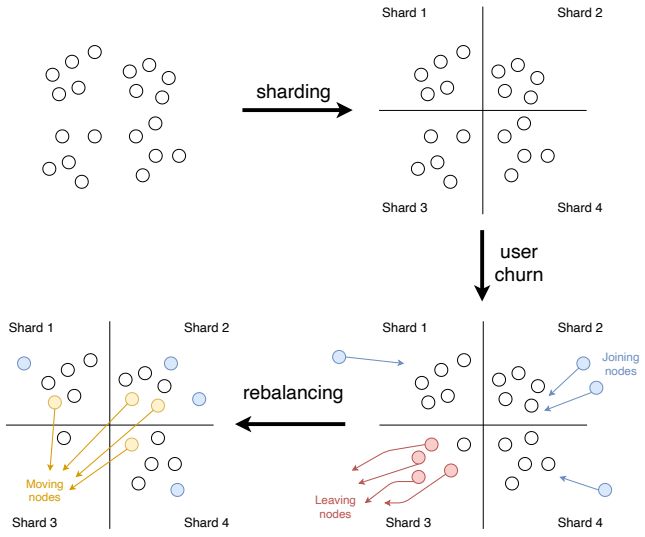


Figure 1: An example of shard allocation. New nodes (in blue) may join the network and existing nodes (in red) may leave the network. After a state update, a subset of nodes (in yellow) may be relocated.

Given the success of sharding 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, sharding permissionless blockchains faces two challenges. First, traditional sharding protocols [27, 28, 39, 45, 49] are designed for systems concerning crash faults where nodes may stop responding, whereas permissionless blockchains concern Byzantine faults where nodes may behave arbitrarily. With Byzantine faults, sharded blockchains may suffer from *single-shard takeover attacks* aka. *1% attacks* [10, 23], where the adversary gathers nodes to a single shard and compromise the shard's consensus. As

*Corresponding author.

sharding divides nodes to different shards, the adversary needs much fewer nodes for compromising a shard compared to launching *51% attacks* in non-sharded blockchains. 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.

In order to address the above challenges, nodes should be allocated into shards securely and dynamically. While existing works on designing [11, 26, 71, 78, 94, 96, 100] and analysing [29, 95, 101] 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– the overlooked core component for shared permissionless blockchains. In particular, we formalise the shard allocation protocol, evaluate the shard allocation protocols of existing blockchain sharding protocols, observe insights and propose WORMHOLE– a correct and efficient shard allocation protocol for permissionless blockchains.

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 framework, we **evaluate the shard allocation subprotocols of seven state-of-the-art permissionless sharded blockchains**, including five academic proposals Elastico [78] (CCS'16), Omniledger [71] (S&P'18), Chainspace [26] (CCS'19), RapidChain [100] (NDSS'19), and Monoxide [96] (NSDI'19), and two industry projects Zilliqa [94] and Ethereum 2.0 [13]. 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 re-balance 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 and studied informally [71], we formally prove it's impossible to achieve optimal values on both, and identify the trade-off between them. All existing shard allocation protocols fall into extreme values on either *self-balance* or *operability*, leading to serious 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**. With minimal security assumption, i.e., a randomness beacon, WORMHOLE is correct and achieves optimal performance. By being *non-memoryless*, WORMHOLE supports parametrisation of *self-balance* and *operability*. We formally prove WORMHOLE's security and performance metrics. We also implement WORMHOLE and show that WORMHOLE introduces negligible overhead.

Paper organisation. Section 2 formalises shard allocation protocols. Section 3 outlines the evaluation results on existing shard allocation protocols, leading to our insights in Section 4. Section 5 describes our shard allocation protocol WORMHOLE. Section 6 discusses extra concerns of WORMHOLE. Section 7 concludes this paper. Appendix A provides full security proofs of WORMHOLE. Appendix B presents related work. Appendix C provides details of our evaluated shard allocation protocols.

2 Formalising shard allocation

In this section, we formalise the shard allocation protocol, including its syntax, threat model, correctness properties and performance metrics.

2.1 Definition

Table 1: 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^t_k	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 that node i is in shard k at round t
γ	Probability of a node to stay at its shard after $\mathsf{UpdateShard}(\cdot)$ ($\gamma \in [0,1)$)

System setting. We consider a permissionless sharded blockchain with m shards. The blockchain executes in rounds. Let st_t be the blockchain's system state at round t. Let n_k^t be the number of nodes in shard $k \in [1,m]$, and $n^t = \sum_{k=1}^{m} n_k^t$ be the total number of nodes at round t. We consider user churn [91]: 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 $\beta_t \leq 1$. That is, for two consecutive rounds t and $t+1$, $n^{t+1} = (1+\alpha_t-\beta_t)n^t$. Each node i has a pair of secret key sk_i and public key pk_i, and is identified by pk_i in the blockchain. In each round, nodes execute shard allocation so that each node is allocated to a unique shard. After shard allocation, nodes in each shard execute a consensus protocol independently, and shards communicate with each other if necessary. Table 1 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*

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

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

$\mathcal{SA}.\mathsf{JoinShard}(sk_i, pp, st_t) \to (k, \pi_{i,st_t,k})$ *: 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.*

$\mathcal{SA}.\mathsf{UpdateShard}(sk_i, pp, st_t, k, \pi_{i,st_t,k}, st_{t+1}) \to (k', \pi_{i,st_{t+1},k'})$ *: 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 $\pi_{i,st_{t+1},k'}$.*

$\mathcal{SA}.\mathsf{VerifyShard}(pp, pk_i, st_t, k, \pi_{i,st_t,k}) \to 0$ *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 round t, node i executes $\mathcal{SA}.\mathsf{JoinShard}(\cdot)$ to be allocated to a shard. Upon a new round $t+1$, it updates its allocated shard 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 round t. Algorithm 1 describes the typical process of shard allocation.

Threat model. We assume all nodes control the same amount of voting power on the consensus. The consensus protocol assumes that the portion of Byzantine nodes does not exceed a certain threshold Ψ. We consider an adversary controlling ϕn^t nodes, where $\phi < \Psi$.

Shard allocation's safety is mainly threatened by the *single-shard takeover attack*. To launch the *single-shard takeover attack* in shard k, the adversary should control more than Ψn_k^t nodes in it. The adversary has several strategies for this. First, the adversary may bias the shard allocation and gather its nodes to a single shard. In addition, if the adversary can predict the shard allocation results, then it may choose key pairs that make nodes to be allocated to the same shard. Moreover, for probabilistic shard allocation protocols, the adversary can launch the *join-leave attack* [50], i.e., enforce each node to keep joining and leaving the blockchain until allocated to the targeted shard.

The adversary may also compromise shard allocation's liveness. If the shard allocation protocol requires collaboration between nodes, then the adversary may refuse to collaborate and prevent shard allocation from terminating. In this way, nodes can never decide their allocated shards.

Algorithm 1: The shard allocation process for node i.

```
// Join at round t
(k_t, π_{i,st_t,k_t}) ← SA.JoinShard(sk_i, pp, st_t)
// Current state, shard and proof
st_*, k_*, π_* ← 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_*, k_*, π_{i,st_*,k_*}, st_+)
    st_* ← st_+
until node i leaves the system
```

2.2 Correctness properties

Based on the threat model, 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, which should always make progress in updating the system state. We adapt the liveness definition by Garay et al. [57].

Definition 2 (Liveness)**.** *Parametrised by a growth factor $\tau \in \mathbb{R}^+$ and $s \in \mathbb{N}^+$. 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. If the adversary can predict shard allocation results, then it can launch the *single-shard takeover attack* by choosing key pairs that make nodes to be allocated to the same shard. To prevent such predicting behaviours, Shard allocation should allocate each node to a random shard deterministically [71, 78, 94]. 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. Formally,

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*

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

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

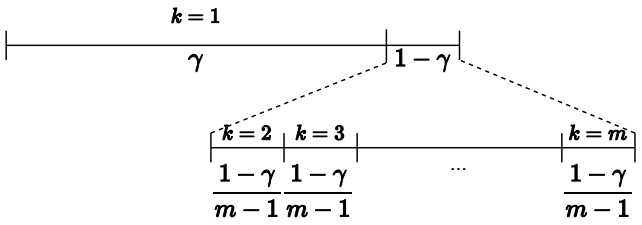


Figure 2: *Update-randomness*. After executing $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$, the probability that a node stays in its shard (say shard 1) is γ, and the probability of moving to each other shard is $\frac{1-\gamma}{m-1}$.

Update-randomness specifies the probability distribution of existing nodes' shard allocation. To remain balanced under churn, existing nodes may need to move to other shards upon state update. Moving to a new shard is computation and communication-intensive, as a node needs to synchronise and verify the new shard's ledger, which can take hundreds of Gigabytes [21, 46, 71, 85]. If a large portion of nodes move to other shards upon each state update, then this introduces non-negligible overhead and may make the system unavailable for a long time. To avoid this, only a small subset of nodes should move each state update. During a state update, we consider that an existing node stays in the same shard with probability γ. 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*

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

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* specifies the blockchain should prevent a node from manipulating the shard allocation results. 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. *Allocation-privacy* specifies that one cannot know the allocated shard of a node before the node reveals itself. We consider *allocation-privacy* is optional, as it has both advantages and disadvantages. *Allocation-privacy* can prevent the adversary from running $\mathsf{JoinShard}(\cdot)$ or $\mathsf{UpdateShard}(\cdot)$ for other nodes and predicting the distribution of nodes among shards. On the other side, *allocation-privacy* makes nodes difficult to find peers in the same shard, and the sharded blockchain should additionally employ a peer finding protocol [78, 94], therefore compromise the sharded blockchain's performance.

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*

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

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*

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

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.

2.3 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 [98]. 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 [52] where the adversary runs numerous nodes to spam the network. To mitigate Sybil attacks, joining the system should have a non-negligible cost. The cost can be diverse, e.g., computing power in Proof-of-Work [66] and money deposit in Proof-of-Stake [70].

Self-balance. To maximise the fault tolerance capacity 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 [96, 102]. 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 the optimal *self-balance*: regardless how many nodes join or leave the system during the last round, the system can balance itself within a round.

Operability. To balance shards, the shard allocation protocol should move some nodes to other shards upon each state update. As mentioned, moving nodes to other shards introduces non-negligible overhead and may make the system unavailable for a long time. *Operability* was introduced to measure the cost of moving nodes [71]. We formally define *operability* Γ as the probability that a node stays at its shard upon a state update. Following *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 §4, 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.

3 Evaluating shard allocation protocols

In this section, we model shard allocation protocols of seven state-of-the-art sharded blockchains and evaluate them based on our framework. Our evaluation (summarised in Table 2) shows that none of these shard allocation protocols is fully correct or achieves satisfactory performance.

3.1 Evaluation criteria

Our evaluation considers three aspects, namely system models, correctness properties, and performance metrics. We consider the standard system model common to all sharded blockchains [71, 78, 94]. As our evaluation only focuses on shard allocation, we assume other subprotocols in sharded blockchains – e.g., consensus and cross-shard communication – are secure. Section 2.2 and 2.3 define correctness properties and performance metrics, respectively.

System model. The system model includes three aspects, namely network models, trust assumptions, and fault tolerance capacity. Network model describes the timing guarantee of message deliveries. We consider three common network models, namely synchrony, partial synchrony [53], and asynchrony. A network is synchronous if messages are delivered within a known finite time-bound; is asynchronous if messages are delivered without a known time-bound; or is partially synchronous if messages are delivered within a known finite time-bound with some clock drift. Trust assumption indicates the trustworthy components that the protocol should assume to remain correct. Fault tolerance capacity indicates the threshold of Byzantine nodes that shard allocation can tolerate while remaining correct. With nodes more than the fault tolerance capacity, then the adversary can compromise some correctness properties of shard allocation. We define the fault tolerance capacity only for shard allocation rather than the entire sharded blockchain, where consensus and cross-shard communication protocols may tolerate fewer faulty nodes. For example, if a shard allocation protocol totally relies on a trusted third party, then we consider its fault tolerance capacity as 1, as it works correctly even when all nodes are Byzantine.

3.2 Overview of evaluated proposals

We choose seven state-of-the-art sharded blockchains, including five academic proposals Elastico [78], Omniledger [71], Chainspace [26], RapidChain [100], and Monoxide [96], and two industry projects Zilliqa [94] and Ethereum 2.0 (ETH 2.0) [13]. We briefly describe their shard allocation protocols and defer their details to Appendix C.

Table 2: 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	✓	✓	✓	✗*	✓	$O(1)$	$O(1)$	Comp.	1	$\frac{1}{m}$
ETH 2.0	-	Async.	-	1	✓	✓	✗	✓	✗	0	0	No**	$1-\beta_t$	1
WORMHOLE (Our proposal in §5)	New rand.	Async.	Rand. Beacon	1	✓	✓	✓	✓	✓	$O(1)$	$O(1)$	Comp.	$1-\beta_t+\frac{\beta_t}{2^{\rho p}}$	$1-\frac{m-1}{m\cdot 2^{\rho p}}$

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

Elastico, Omniledger, RapidChain, and Zilliqa rely on distributed randomness generation (DRG) protocols for shard allocation. In Elstico [78], nodes in a special shard called *final committee* run a commit-and-reveal DRG protocol [30] to produce a random output. Then, with this random output and its identity, each node solves a PoW puzzle, and will be assigned to a shard according to its PoW solution. Zilliqa [94] is built upon Elastico with several optimisations. Most notably, Zilliqa uses the last block's hash value as the current round's random output. Omniledger [71] uses the *RandHound* [92] DRG protocol, and relies on a trusted identity authority for shard allocation. As RandHound is leader-based, nodes need to run a leader election before RandHound. If the leader election fails for five times, then nodes fallback to run an asynchronous DRG protocol [37]. Given the latest random output, the identity authority samples a subset of pending nodes and approve them to join the system, and shuffles all existing nodes in the network to different shards. In RapidChain [100], each node solves an offline PoW puzzle before joining the system. To prevent pre-computing PoW puzzles, RapidChain employs a Feldman VSS [55]-based DRG protocol and uses its random outputs as PoW puzzles' inputs. A special shard called *reference committee* allocates joined nodes into different shards following the *Commensal Cuckoo* rule [90]. In Commensal Cuckoo, each node is mapped to an ID $x \in [0,1)$. Upon a new node with ID x, the reference committee moves nodes with IDs close to x to other shards randomly.

The rest of sharded blockchains do not employ DRG protocols. In Chainspace [26], nodes can apply to move shards at any time, and other nodes decide on the application by voting. The voting works over a special smart contract `ManageShards`, which is assumed secure. Monoxide [96] and Ethereum 2.0 [11] simply allocate nodes into different shards according to prefixes of their addresses.

3.3 System model

Network model. Elastico and RapidChain assume synchronous network models, as they rely on synchronous DRG protocols [30,55]. 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 [37]. 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 the synchronous DRG protocol [30].

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

Fault tolerance capacity. Elastico and Omniledger achieve the fault tolerance capacity 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 trustworthy smart contracts. For Zilliqa, we assume blocks are produced correctly and shard computation is offline.

3.4 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 Collective Signing protocol [93] in *RandHound* may lose liveness under a single Byzantine node [99].

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.

3.5 Performance

Communication complexity. For Elastico, the communication complexity of $\mathsf{JoinShard}(\cdot)$ and $\mathsf{UpdateShard}(\cdot)$ are $O(n^f)$, 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 needs to run the DRG protocol, which consists of a vector consensus [83] with communication complexity of $O(n^f)$. Ideally, the final committee has $\frac{n}{m}$ nodes, and the communication complexity is $O(\frac{n}{m}^{\frac{f}{m}}) = O(\frac{n}{m}^{\frac{f}{m}}) = O(n^f)$ (m is constant). For Omniledger, the communication complexity of $\mathsf{JoinShard}(\cdot)$ is $O(1)$, as a node only communicates with the identity authority for joining the system. The communication complexity of $\mathsf{UpdateShard}(\cdot)$ is $O(n)$ or $O(n^3)$. The best case of $\mathsf{UpdateShard}(\cdot)$ is that the VRF-based leader election and RandHound are both successful. This leads to the communication complexity of $O(n)$. The worst case is that nodes fallback to run the asynchronous DRG [37], whose communication complexity is $O(n^3)$. For RapidChain, the communication complexity of $\mathsf{JoinShard}(\cdot)$ and $\mathsf{UpdateShard}(\cdot)$ are $O(n^2)$, as the Feldman VSS has the communication complexity of $O(n^2)$ [55]. Monoxide and ETH2.0 requires no communication for shard allocation, as a node decides its shard by its address. Zilliqa requires $O(1)$ communication complexity as a node as a node needs the latest block to calculate its shard.

Sybil cost. The Sybil cost of Elastico, RapidChain, and Zilliqa comes from the computational work of solving PoW puzzles for joining shards. 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. Chainspace does not provide details on issuing memberships either.

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 $1-\frac{\alpha_t n \cdot \kappa n}{n} = 1-\kappa\alpha_t n$. As *operability* cannot be smaller than 0 in reality, *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.

4 Observation and insights

We observe from Table 2 that the evaluated shard allocation protocols cannot simultaneously achieve optimal *self-balance* and *operability*. In particular, *operability* γ is either 1 or $\frac{1}{m}$ (except for RapidChain), and *self-balance* ω is either $1-\beta_t$ or 1. We formally analyse this observation, and prove that it is impossible to achieve optimal *operability* and *self-balance* simultaneously. We then observe a property *non-memorylessness* that enables parametrisation of the trade-off between *operability* and *self-balance*. Given that shard allocation with $\gamma = \frac{1}{m}$ or $\omega = 1-\beta_t$ is impractical, making this trade-off parametrisable opens a new in-between design space and makes shard allocation practical.

4.1 Impossibility and trade-off

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. By 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

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

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

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

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

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

Thus, 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 $\Delta n_i^t - \Delta n_j^t = \beta_t n^t$,

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

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

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

□

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.

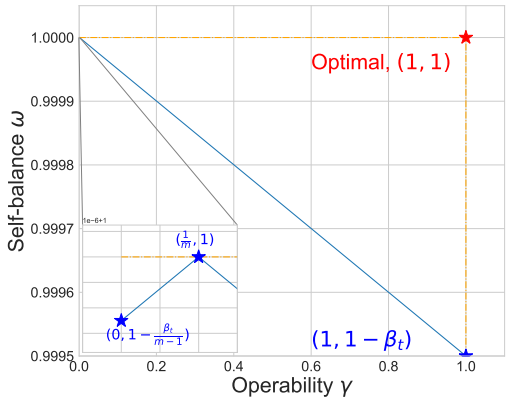


Figure 3: Relationship between *operability* and *self-balance*. We pick $m = 1000$ and $\beta_t = 0.0005$ as an example.

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. □

4.2 Parameterising the trade-off

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. In addition, none of our evaluated protocols allows parametrising the trade-off between *self-balance* and *operability*. We observe that, to parametrise this trade-off, sharding protocols should be *non-memoryless*. In signal processing, a system is *memoryless* if the output signal at each time depends only on the input at that time [82]. On the contrary, *non-memoryless* means 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. *When $\gamma = \frac{1}{m}$ or 1, $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ does not rely on any prior system state. When $\gamma = \frac{1}{m}$, the resulting shard of $\mathcal{SA}.\mathsf{UpdateShard}(\cdot)$ is uniformly distributed, 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. 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.*

5 WORMHOLE– Shard allocation from randomness beacon and VRFs

Based on the gained insights, we propose WORMHOLE, a correct and efficient permissionless shard allocation protocol. WORMHOLE is constructed from a randomness beacon (e.g. [34, 35, 88, 92]) and a verifiable random function (VRF) (e.g., [51, 61, 80]). WORMHOLE works correctly regardless of the network environment or the portion of Byzantine nodes, and supports parametrisation on the trade-off between *self-balance* and *operability* in Figure 3. We formally analyse WORMHOLE's security and experimentally evaluates its efficiency. The evaluation results show that WORMHOLE achieves practical efficiency.

5.1 Primitives

Verifiable random function [80] is a public-key version of hash functions. 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. Some VRFs support *batch verification* [44,65], i.e., verifying multiple VRF hashes at the same time, which is faster than verifying VRF hashes one-by-one. Formally, a VRF is a tuple of five algorithms:

- $\mathsf{VRFKeyGen}(\lambda) \to (sk, pk)$: On input a security parameter λ, outputs the secret/public key pair (sk, pk).
- $\mathsf{VRFHash}(sk, m) \to h$: On input sk and an arbitrary-length string m, outputs a fixed-length hash h.
- $\mathsf{VRFProve}(sk, m) \to \pi$: On input sk and m, outputs the proof π for h.
- $\mathsf{VRFVerify}(pk, m, h, \pi) \to \{0,1\}$: On input pk, m, h, π, outputs the verification result 0 or 1.
- $\mathsf{VRFBatchVerify}(pk, \vec{m}, \vec{h}, \vec{\pi}) \to \{0,1\}$: On input pk, a series of strings $\vec{m} = (m_1, \ldots, m_n)$, outputs $\vec{h} = (h_1, \ldots, h_n)$, and proofs $\vec{\pi} = (\pi_1, \ldots, \pi_n)$, outputs the verification result 0 or 1.

VRF should satisfy the following three properties [60].

- *VRF-Uniqueness*: Given a secret key sk and an input m, $\mathsf{VRFHash}(sk, m)$ produces a unique valid output.
- *VRF-Collision-Resistance*: It is computationally hard to find two inputs m and m' such that $\mathsf{VRFHash}(sk, m) = \mathsf{VRFHash}(sk, m')$.
- *VRF-Pseudorandomness*: It is computationally hard to distinguish the output of $\mathsf{VRFHash}(\cdot)$ from a random string without the knowledge of the corresponding public key and proof.

Randomness beacon [69] aims at generating random outputs periodically, which are usually constructed from distributed randomness generation (DRG) protocols [31,37,58,63,68,72,81,86,92] – a family of protocols where nodes jointly produce a random output. A randomness beacon should satisfy the following properties [88]:

- *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.

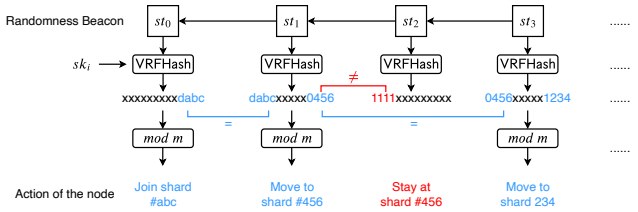


Figure 4: An intuition of WORMHOLE. All numbers are in hexadecimal, $op = 4$ and $m = 16^3$.

5.2 Overall design

WORMHOLE relies on a correct randomness beacon $\mathcal{RB}$. Each random output from $\mathcal{RB}$ updates the system state. According to Theorem 2 and Remark 1, if operability $\gamma \in (\frac{1}{m}, 1)$, then each membership proof should carry information of all historical system states. This keeps the size of membership proofs growing as the system operates, making the shard allocation practical. To limit the size of membership proofs, we introduce a Sybil resistance parameter sr. In round t, a node calculates its allocated shard using the latest $\ell = sr + t \mod sr \in [sr, 2sr)$ system states, rather than all system states. Thus, the membership proof size is bound by $O(\ell) = O(sr)$, and nodes can join the system by computing over only a limited number of system states. The Sybil resistance parameter sr also controls the Sybil resistance degree. With a larger sr, a node should iterate over more system states for joining the system, and the adversary should do a huge amount of computation for spawning Sybil nodes.

Algorithm 2: Calculating the shard with VRF hashes.

```
Algorithm calculateShard(m, op, h_{t-ℓ+1}, ..., h_t):
    idx ← t − ℓ + 1
    for j ∈ [t − ℓ + 2, t] do
        if MSB(op, h_j) = LSB(op, h_idx) then
            idx ← j          // Can be cached for reuse
    shard_id ← h_idx mod m
    return shard_id
```

Figure 4 and Algorithm 2 show the intuition and construction of $\mathsf{calculateShard}(\cdot)$, the function that calculates the allocated shard, respectively. It takes an operability parameter op for controlling the probability that a node moves to a new shard, and VRF hashes of ℓ system states as input. It iterates over ℓ VRF outputs while tracking the index idx that decides the allocated shard. Let $\mathsf{MSB}(z, str)$ and $\mathsf{LSB}(z, str)$ be the z most and least significant bits of str, respectively. For each iteration, say j-th iteration, if $\mathsf{LSB}(op, h_j) = \mathsf{MSB}(op, h_{idx})$, then idx is updated to j, otherwise idx remains unchanged. The allocated shard $shard\_id$ is calaculated as $h_{idx} \mod sr$. With a large op, the probability of moving to a new shard is small. The index idx can be cached. Upon a new VRF hash h_{t+1},

if $(t+1) \mod sr \neq 0$, then computing $\mathsf{calculateShard}(\cdot)$ does not need to go through all iterations, but only needs to check if $\mathsf{MSB}(op, h_{t+1}) = \mathsf{LSB}(op, h_{idx})$, and compute *idx* and *shard_id* accordingly. If $(t+1) \mod sr = 0$, then $t-\ell+1 \neq t-\ell+2$, and $\mathsf{calculateShard}(\cdot)$ should be computed from scratch.

5.3 Detailed construction

System setup (Algorithm 3). The setup algorithm takes security parameter λ as input, and outputs the number of shards m, the Sybil resistance parameter sr, and 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 $\mathsf{calculateShard}(\cdot)$. 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 round $t+1$ node i only needs to calculate one more VRF hash of st_{t+1} and repeats the process in Algorithm 4 over the last ℓ states. When $t \mod sr \neq 0$, only a small set of nodes move to other shards upon a state update. When $t \mod sr = 0$, all nodes are shuffled. By specifying a large sr, such shuffling can be infrequent and the resulting overhead can be acceptable.

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). To verify a membership proof $\pi_{i,st_t,k}$, the verifier executes $\mathsf{VRFBatchVerify}(\cdot)$ to verify ℓ VRF hashes, and executes $\mathsf{calculateShard}(\cdot)$ over these VRF hashes to verify its output against k. Note that verification results can be cached: upon an updated membership proof $\pi_{i,st_{t+1},k'}$, the verifier can reuse most of the computation in verifying $\pi_{i,st_t,k}$, including verification results of previous VRF hashes and $\mathsf{calculateShard}(\cdot)$.

Algorithm 6: Verifying shard membership.

Algorithm $\mathsf{VerifyShard}(pp, pk_i, st_t, k, \pi_{i,st_t,k})$**:**
- $m, op, sr \leftarrow pp$
- $\ell \leftarrow t \mod sr + sr$
- $(pk_i, h_{last}, \ldots, h_t, \pi_{last}, \ldots, \pi_t) \leftarrow \pi_{i,st_t,k}$
- **if** $last \neq t-\ell+1$ **then**
  - **return** 0
- $\vec{st}, \vec{h}, \vec{\pi} \leftarrow$ $(st_{last}, \ldots, st_t), (h_{last}, \ldots, h_t), (\pi_{last}, \ldots, \pi_t)$
- `// Verification results be cached`
- **if** $\mathsf{VRFBatchVerify}(pk_i, \vec{st}, \vec{h}, \vec{\pi}) = 0$ **then**
  - **return** 0
- **if** $k \neq \mathsf{calculateShard}(m, op, h_{t-\ell+1}, \ldots, h_t)$ **then**
  - **return** 0
- **return** 1

5.4 Theoretical analysis

Correctness. We defer full security proofs to Appendix A, and summarise them as follows. As long as $\mathcal{RB}$ works correctly, WORMHOLE satisfies *liveness*, as a node only needs its key pair and system states for $\mathsf{JoinShard}(\cdot)$ and $\mathsf{UpdateShard}(\cdot)$. WORMHOLE satisfies *unbiasibility*, as

VRFHash$(\cdot)$ and calculateShard$(\cdot)$ are deterministic functions, and system states are *unbiasible*, guaranteed by $\mathcal{RB}$. WORMHOLE satisfies *join-randomness*, as VRF produces uniformly distributed outputs. WORMHOLE satisfies *update-randomness*. When $t \neq k \cdot sr$, the probability that two random outputs share the same op-bit substring is $\frac{1}{2^{op}}$. Within the probability $\frac{1}{2^{op}}$, the probability that two random outputs result in the same shard is $\frac{1}{m}$. This leads to $\gamma = 1 - \frac{1}{2^{op}} \cdot \frac{m-1}{m} = 1 - \frac{m-1}{m \cdot 2^{op}}$. When $t = k \cdot sr$, all nodes will be shuffled, leading to $\gamma = \frac{1}{m}$. Thus, WORMHOLE satisfies *allocation-randomness*. WORMHOLE satisfies *allocation-privacy*, as one cannot compute JoinShard$(\cdot)$ or UpdateShard$(\cdot)$ for a node without knowing its secret key. The probability of guessing shard allocation follows the proof of *allocation-randomness*.

Performance metrics. The *communication complexity* of JoinShard$(\cdot)$ and UpdateShard$(\cdot)$ of WORMHOLE are $O(1)$, as node only needs to receive a constant number of system states for executing JoinShard$(\cdot)$ and UpdateShard$(\cdot)$. WORMHOLE resists Sybil attacks using computation. To execute JoinShard$(\cdot)$, a node should executes VRFHash for ℓ times, where $\ell = t \mod sr + sr$. To increase the Sybil cost without affecting verification overhead, one can incorporate VRFs with time-asymmetric cryptographic primitives, such as Verifiable Delay Functions [32] and PoW. WORMHOLE's *operability* $\gamma = 1 - \frac{m-1}{m \cdot 2^{op}}$. By Proof A,

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

WORMHOLE's *self-balance* $\omega = 1 - \beta_t + \frac{\beta_t}{2^{op}}$. By Definition 1,

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

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

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

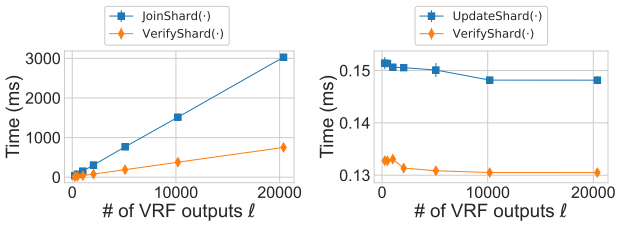
$$= 1 - (1 - \frac{1}{2^{op}})\beta_t \tag{11}$$

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

5.5 Experimental evaluation

To demonstrate WORMHOLE's efficiency, we implement and benchmark WORMHOLE. Our evaluation shows that WORMHOLE introduces negligible overhead.

Implementation. We implement WORMHOLE in Rust programming language. We use `rug` [6] for large integer arithmetic and `bitvec` [4] for bit-level operations. We use `w3f/schnorrkel` [19], which implements the standardised VRF [61] with the Schnorr-style aggregatable discrete log equivalence proofs (DLEQs) [44] based on the Ed25519 [9] elliptic curve with Ristretto [18] compressed points. The size of keys, VRF hashes and proofs are 32 Bytes, 32 Bytes and 96 Bytes, respectively. We use random strings using `rand` [5] for simulating randomness beacon. We sample 20 executions for each configuration, i.e., a function with a unique group of parameters. We write the benchmarks using `cargo-bench` [3] and `criterion` [7]. We specify the O3-level optimisation for compilation, and conduct all experiments on a MacBook Pro with a 2.2 GHz 6-Core Intel Core i7 Processor and a 16 GB 2400 MHz DDR4 RAM.



(a) Newly joined nodes. (b) Existing nodes.

Figure 5: Evaluation of WORMHOLE.

Computation overhead. We benchmark WORMHOLE's JoinShard$(\cdot)$, UpdateShard$(\cdot)$ and VerifyShard$(\cdot)$. For VerifyShard$(\cdot)$, previous results can be cached. We test $\ell \in [256, 512, 1024, 2048, 5096, 10192, 20384]$. Recall that $\ell \in [sr, 2sr)$, when $\ell = 20384$, sr is at least 10192, i.e., all nodes are shuffled for every 10192 new system states. In Bitcoin's setting where a block is generated for every ten minutes, it takes 142 days for each global shuffling.

Figure 5 shows the evaluation results. For newly joined nodes, the execution time of JoinShard$(\cdot)$ and VerifyShard$(\cdot)$ increases linearly with ℓ. When $\ell = 20384$, JoinShard$(\cdot)$ and VerifyShard$(\cdot)$ take 3000 and 800 milliseconds, respectively. For existing nodes, UpdateShard$(\cdot)$ and VerifyShard$(\cdot)$ take approximately 0.15 and 0.13 millisecond regardless of ℓ, respectively. To increase Sybil cost, one can incorporate VRFs with time-asymmetric primitives such as Verifiable Delay Functions [32] and PoW.

Storage overhead. A membership proof includes a public key, ℓ VRF hashes and ℓ VRF proofs. Thus, in our implementation, a membership proof takes $32 + 32\ell + 96\ell = 32 + 128\ell$ Bytes. When $\ell = 20384$, each membership proof takes 2.5 MB. Consider Bitcoin's setting that, each node maintains persistent connections and synchronises states with at most 125 peers [1]. This takes at most $125 * 2.5\text{MB} \approx 312.5\text{MB}$, which is negligible compared to the ledger that can take hundreds of Gigabytes. The proof size can be further optimised if the VRF supports aggregation of signatures, and we consider constructing such VRFs as future work.

Communication overhead. In WORMHOLE, each node executes JoinShard$(\cdot)$ or UpdateShard$(\cdot)$ locally to obtain its shard membership. The only communication overhead in-

troduced by WORMHOLE is nodes exchanging membership proofs. According to the storage overhead analysis, when $\ell = 20384$, each node receives at most 312.5 MB at the time of joining the network. Upon a new system state, each node computes the VRF on it and broadcasts the VRF hash and proof, which take 128 Bytes. Thus, upon each system state, a node receives at most $125 * 128\text{Bytes} \approx 15.6\text{KB}$. This is negligible compared to exchanging blockchain transactions.

6 Discussions

Replacing DRG with randomness beacon. DRG usually relies on strong assumptions and suffer from high communication complexity. Replacing them with a randomness beacon can significantly simplify assumptions, improve security, and reduce communication overhead of shard allocation protocols. For a fairer comparison, we evaluate DRG-based shard allocation protocols while replacing DRG protocols with an external randomness beacon.

The evaluation results in Table 3 show that all protocols improve much in terms of the system model and communication complexity. However, these shard allocation protocols still suffer from some problems they originally have, and WORMHOLE still outperforms them. For example, 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*.

The randomness beacon assumption. We consider randomness beacon as an acceptable assumption. First, randomness beacon is a weak trusted third party – weaker than the identity authority and the smart contract in Omniledger and Chainspace, respectively. In addition, assuming randomness beacons has been accepted in some works [33]. Moreover, there have been secure and practical randomness beacons based on blockchains [34], Publicly Verifiable Secret Sharing (PVSS) [38, 88, 92], Verifiable Delay Functions [54, 75], Nakamoto consensus [64], and/or real-world entropy [34, 41]. Some countries [2, 17, 69] and organisations [8, 20] also deploy their own public randomness beacons.

Construction without allocation-privacy. As mentioned in §2.2, *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 the system state, and $H(\cdot)$ is a cryptographic hash function.

7 Conclusion

Designing permissionless sharded blockchains remains as an open challenge, and one of the key reasons is the overlooked shard allocation protocol. In this paper, we fill this gap by formally defining the permissionless shard allocation protocol, evaluating existing shard allocation protocols, observing insights and constructing correct and efficient shard allocation protocol WORMHOLE. In the future, we will keep improving WORMHOLE, and propose concrete solutions on integrating WORMHOLE into sharded blockchains and other permissionless systems.

References

[1] Bitcoin core documentation. https://github.com/bitcoin/bitcoin/blob/master/doc/README.md.

[2] Brazilian beacon. https://beacon.inmetro.gov.br/.

[3] cargo-bench. https://doc.rust-lang.org/cargo/commands/cargo-bench.html.

[4] crates/bitvec. https://crates.io/crates/bitvec.

[5] crates/rand. https://crates.io/crates/rand.

[6] crates/rug. https://crates.io/crates/rug.

[7] criterion.rs. https://github.com/bheisler/criterion.rs.

[8] Distributed randomness beacon | cloudflare. https://www.cloudflare.com/leagueofentropy/.

[9] Ed25519: high-speed high-security signatures. https://ed25519.cr.yp.to/.

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

[11] ethereum/eth2.0-specs. https://github.com/ethereum/eth2.0-specs.

[12] ethereum/eth2.0-specs at v0.9.0. https://github.com/ethereum/eth2.0-specs/tree/v0.9.0.

[13] ethereum/wiki. https://github.com/ethereum/wiki.

[14] ethereum/wiki at 6aee544ccc427490e443639ed29a1e4597cb898e. https://github.com/ethereum/wiki/tree/6aee544ccc427490e443639ed29a1e4597cb898e.

[15] Phase 0 for humans. https://notes.ethereum.org/@djrtwo/Bkn3zpwxB?type=view.

[16] Randao: A dao working as rng of ethereum. https://github.com/randao/randao.

[17] Random uchile - random uchile. https://beacon.clcert.cl/en/.

Table 3: Evaluation of shard allocation protocols that replace DRG with a randomness beacon. Meanings of colours are same as Table 2. ★ means the metric is improved by replacing DRG with a randomness beacon.

	State update	System model: Network model	System model: Trust assumption	System model: Fault tolerance	Correctness: Public verifiability	Correctness: Liveness	Correctness: Allocation-rand. Unbiasibility	Correctness: Privacy^o	Performance metrics: Join comm. compl.	Performance metrics: Update comm. compl.	Performance metrics: Sybil cost	Performance metrics: Self-balance	Performance metrics: Operability
Elastico	New block	Async.★	Rand. Beacon	1★	✓	✓	✓	✗*	✓	$O(1)$★	$O(1)$★	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	✓	✓	✓	✗*	✓	$O(1)$	$O(1)$	Comp.	1	$\frac{1}{m}$
WORMHOLE (Our proposal in §5)	New rand.	Async.	Rand. Beacon	1	✓	✓	✓	✓	✓	$O(1)$	$O(1)$	Comp.	$1-\beta_t+\frac{\beta_t}{2^{o\rho}}$	$1-\frac{m-1}{m\cdot 2^{o\rho}}$

^{o} Optional. * Protected by PoW puzzles.

[18] The ristretto group. https://ristretto.group/.

[19] Schnorr vrfs and signatures on the ristretto group. https://github.com/w3f/schnorrkel.

[20] Unicorn beacon by lacal. http://trx.epfl.ch/beacon/index.php.

[21] Warp sync - wiki parity tech documentation. https://wiki.parity.io/Warp-Sync.

[22] Zilliqa - the next generation, high throughput blockchain platform. https://zilliqa.com/.

[23] The zilliqa design story piece by piece: Part 1 (network sharding). https://blog.zilliqa.com/https-blog-zilliqa-com-the-zilliqa-design-story-piece-by-piece-part1-d9cb32eale65.

[24] Zilliqa developer portal · technical and api documentation for participating in the zilliqa network. https://dev.zilliqa.com/.

[25] Zilliqa/zilliqa at v5.0.1. https://github.com/Zilliqa/Zilliqa/tree/v5.0.1.

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

[27] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: managing datastore locality at scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460, 2018.

[28] Brice Augustin, Timur Friedman, and Renata Teixeira. Measuring load-balanced paths in the Internet. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 149–160, 2007.

[29] Georgia Avarikioti, Eleftherios Kokoris-Kogias, and Roger Wattenhofer. Divide and Scale: Formalization of Distributed Ledger Sharding Protocols. *arXiv preprint arXiv:1910.10434*, 2019.

[30] Baruch Awerbuch and Christian Scheideler. Robust random number generation for peer-to-peer systems. In *International Conference On Principles Of Distributed Systems*. Springer, 2006.

[31] Manuel Blum. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, 15(1):23–27, 1983.

[32] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

[33] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. *IACR Cryptol. ePrint Arch.*, 2020:25, 2020.

[34] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On Bitcoin as a public randomness source. *IACR Cryptology ePrint Archive*, 2015:1015, 2015.

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

[36] Vitalik Buterin. Serenity design rationale. https://notes.ethereum.org/@vbuterin/rkhCgQteN?type=view.

[37] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[38] Ignacio Cascudo and Bernardo David. Albatross: publicly attestable batched randomness based on secret sharing.

[39] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[40] Shuai Che, Gregory Rodgers, Brad Beckmann, and Steve Reinhardt. Graph coloring on the GPU and some techniques to improve load imbalance. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 610–617. IEEE, 2015.

[41] Jeremy Clark and Urs Hengartner. On the use of financial data as a random beacon. *EVT/WOTE*, 89, 2010.

[42] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[43] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. *arXiv preprint arXiv:1505.06895*, 2015.

[44] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proceedings on Privacy Enhancing Technologies*, 2018(3):164–180, 2018.

[45] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[46] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10. IEEE, 2013.

[47] David J DeWitt, Jeffrey F Naughton, and Donovan F Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1991.

[48] Prithula Dhungel, Di Wu 0001, Brad Schonhorst, and Keith W Ross. A measurement study of attacks on BitTorrent leechers. In *IPTPS*, volume 8, pages 7–7, 2008.

[49] Diego Didona and Willy 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)*, pages 79–94, 2019.

[50] Jochen Dinger and Hannes Hartenstein. Defending the sybil attack in p2p networks: Taxonomy, challenges, and a proposal for self-registration. In *First International Conference on Availability, Reliability and Security (ARES'06)*, pages 8–pp. IEEE, 2006.

[51] Yevgeniy Dodis. Efficient construction of (distributed) verifiable random functions. In *International Workshop on Public Key Cryptography*, pages 1–17. Springer, 2003.

[52] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*. Springer, 2002.

[53] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[54] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Continuous verifiable delay functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 125–154. Springer, 2020.

[55] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (FOCS 1987)*, 1987.

[56] Antonio Fernández, Vincent Gramoli, Ernesto Jiménez, Anne-Marie Kermarrec, and Michel Raynal. Distributed slicing in dynamic systems. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 66–66. IEEE, 2007.

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

[58] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999.

[59] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

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

[61] Sharon Goldberg, Jan Vcelak, Dimitrios Papadopoulos, and Leonid Reyzin. Verifiable random functions (VRFs). 2018.

[62] Vincent Gramoli, Ymir Vigfusson, Ken Birman, Anne-Marie Kermarrec, and Robbert van Renesse. A fast distributed slicing algorithm. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 427–427, 2008.

[63] J Alex Halderman and Brent Waters. Harvesting verifiable challenges from oblivious online sources. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.

[64] Runchao Han, Jiangshan Yu, and Haoyu Lin. Randchain: Decentralised randomness beacon from sequential proof-of-work. *IACR Cryptol. ePrint Arch.*, 2020.

[65] Susan Hohenberger and Brent Waters. Constructing verifiable random functions with large input spaces. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 656–672. Springer, 2010.

[66] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure information networks*, pages 258–272. Springer, 1999.

[67] Mark 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)*, pages 117–124. IEEE, 2006.

[68] Aniket Kate and Ian Goldberg. Distributed key generation for the internet. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 2009.

[69] John Kelsey, Luís TAN Brandão, Rene Peralta, and Harold Booth. A reference for randomness beacons: Format and protocol version 2. Technical report, National Institute of Standards and Technology, 2019.

[70] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[71] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[72] Eleftherios Kokoris-Kogias, Alexander Spiegelman, Dahlia Malkhi, and Ittai Abraham. Bootstrapping Consensus Without Trusted Setup: Fully Asynchronous Distributed Key Generation.

[73] Yoram Kulbak, Danny Bickson, et al. The eMule protocol specification. *eMule project, http://sourceforge.net*, 2005.

[74] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[75] Arjen K Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptol. ePrint Arch.*, 2015:366, 2015.

[76] Chuang Lin, Yixin Jiang, Xiaowen Chu, Hongkun Yang, et al. An effective early warning scheme against pollution dissemination for BitTorrent. In *GLOBECOM 2009-2009 IEEE Global Telecommunications Conference*, pages 1–7. IEEE, 2009.

[77] Xiaosong Lou and Kai Hwang. Collusive piracy prevention in P2P content delivery networks. *IEEE Transactions on Computers*, 58(7):970–983, 2009.

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

[79] Francisco Maia, Miguel Matos, Rui Oliveira, and Etienne Riviere. Slicing as a distributed systems primitive. In *2013 Sixth Latin-American Symposium on Dependable Computing*, pages 124–133. IEEE, 2013.

[80] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science*. IEEE, 1999.

[81] Olumuyiwa Oluwasanmi and Jared Saia. Scalable byzantine agreement with a random beacon. In *Symposium on Self-Stabilizing Systems*. Springer, 2012.

[82] Alan V Oppenheim. *Discrete-time signal processing*. Pearson Education, 1999.

[83] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[84] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *International Workshop on Peer-to-Peer Systems*, pages 205–216. Springer, 2005.

[85] Xiaoyao Qian. *Improved authenticated data structures for blockchain synchronization*. PhD thesis, 2018.

[86] Michael O Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256–267, 1983.

[87] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, 2001.

[88] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. HydRand: Efficient Continuous Distributed Randomness. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 32–48.

[89] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference*, pages 148–164. Springer, 1999.

[90] Siddhartha Sen and Michael J Freedman. Commensal cuckoo: Secure group partitioning for large-scale services. *ACM SIGOPS Operating Systems Review*, 46(1):33–39, 2012.

[91] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006.

[92] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[93] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities" honest or bust" with decentralized witness cosigning. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

[94] ZILLIQA Team et al. The ZILLIQA Technical Whitepaper. *Retrieved September*, 2017.

[95] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019*, 2019.

[96] Jiaping Wang and Hao Wang. Monoxide: Scale out Blockchains with Asynchronous Consensus Zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[97] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[98] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing (preliminary report). In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 209–213, 1979.

[99] Jiangshan Yu, David Kozhaya, Jeremie Decouchant, and Paulo Verissimo. RepuCoin: Your reputation is your power. *IEEE Transactions on Computers*, 2019.

[100] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948. ACM, 2018.

[101] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J Knottenbelt. Sok: Communication across distributed ledgers. Technical report, IACR Cryptology ePrint Archive, 2019: 1128, 2019.

[102] Jun Zhao, Jiangshan Yu, and Joseph K Liu. Consolidating Hash Power in Blockchain Shards with a Forest. In *International Conference on Information Security and Cryptology*, pages 309–322. Springer, 2019.

A Security proofs of WORMHOLE

Lemma 2. *If the deployed $\mathcal{RB}$ generates $\lfloor \tau \cdot s \rfloor$ random outputs in any s round (where $\tau \in \mathbb{R}^+$ and $s \in \mathbb{N}^+$), 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 $\lfloor \tau \cdot s \rfloor$ 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 JoinShard$(\cdot)$ or UpdateShard$(\cdot)$ with non-negligible probability. This consists of three attack vectors: 1) the adversary can manipulate the system state; 2) when $\mathcal{SA}$.JoinShard$(\cdot)$ or $\mathcal{SA}$.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 JoinShard$(\cdot)$ or 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 $\frac{1}{m}+\varepsilon$ for some k and non-negligible ε. Running JoinShard$(\cdot)$ requires the execution of 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 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 UpdateShard$(\cdot)$ to comply with the distribution in Definition 4.

When $t \neq k \cdot sr$, the last VRF hash remains unchanged. In 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 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. □

B Related work

We briefly review existing research on sharding distributed systems and compare our contributions with two studies systematising blockchain sharding protocols.

Sharding for CFT 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 [39, 42, 74]. The main challenge is to balance the computation, communication, and storage workload. Despite a large number of load-balancing algorithms [27, 28, 40, 45, 49], 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 [73, 84] employ Distributed Hash Tables (DHT) [87] 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 [48, 76, 77], therefore are not suitable for blockchains, which demands strong consistency on financial data.

Distributed Slicing. Distributed Slicing [67] aims at grouping nodes with heterogeneous computing and storage capacities in a P2P network to optimise resource utilisation. In line with CFT systems, these algorithms [47, 56, 62, 79] require nodes to honestly report their computing and storage capacities, therefore are not suitable in a Byzantine environment.

Evaluation of sharded blockchains. Wang et al. [95] propose an evaluation framework based on Elastico's architecture; Avarikioti et al. [29] formalise sharded blockchains by extending the model of Garay et al. [57]. 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.

C Details of evaluated shard allocation protocols

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.

C.1 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).

C.2 Elastico

Algorithm 7 describes the process of Elastico's shard allocation protocol. 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 [83] 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.3 Omniledger

Algorithm 8 describes the process of Omniledger's shard allocation protocol. Similar to Elastico, Omniledger's shard allocation protocol is also constructed from DRG. In Omniledger, all nodes in the network jointly run *RandHound* [92] - a leader-based DRG protocol tolerating $\frac{1}{3}$ faulty nodes - to generate the randomness. *RandHound* adapts Publicly Verifiable Secret Sharing (PVSS) [89] to make the randomness publicly verifiable, and CoSi [93] 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 [92].

As *RandHound* is leader-based, nodes should elect a leader before running *RandHound*. In Omniledger, nodes run a VRF [80]-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 $\mathsf{VRFHash}(sk_i, \text{"}leader\text{"}||peers||v)$, 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 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 [37]. The coin-tossing protocol [37] 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.

C.4 RapidChain

Algorithm 9 describes the process of RapidChain's shard allocation protocol. 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) [55]. 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 [90] 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.

C.5 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.

C.6 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.

C.7 Zilliqa

Zilliqa [22] is a permissionless sharded blockchain that claims to achieve the throughput of over 2,828 transactions per second. It follows the design of Elastico [78], but with several optimisations. Our evaluation is based on Zilliqa's whitepaper [94], Zilliqa's developer page [24], and Zilliqa's source code (the latest stable release v5.0.1) [25]. Algorithm 10 describes the process of Zilliqa's shard allocation protocol.

Different from Elastico which runs a DRG, 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 it 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.

C.8 Ethereum 2.0

Ethereum 2.0 (ETH 2.0) is the next generation of Ethereum [97]. 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 [13] and blogs [15, 36], and the Ethereum community provides the specification [11] 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 [14] and the ETH 2.0 specification v0.9.0 Tonkatsu [12].

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 a 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 [16]), but it is used for sampling "validators" (a subset of nodes that can produce blocks) rather than shard allocation.

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

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 [37].

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**
    $ticket_i \leftarrow \mathsf{VRFHash}(sk_i, \text{“}leader\text{”}||peers||v)$
    Broadcast $ticket_i$
    Collect tickets until $timer$ that timeouts in Δ
    Find the smallest ticket as $ticket_j$
    if $ticket_i == ticket_j$ **then**
      $\pi_{ticket,i} \leftarrow$ $\mathsf{VRFProve}(sk_i, \text{“}leader\text{”}||peers||v)$
      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 $p[pk_i] == k$

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) **:**
  $nonce, out \leftarrow$ $\mathsf{PoWWork}(st_t.T, timestamp||pk_i||st_t.R)$
  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

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