Analysing and Improving Shard Allocation Protocols for Sharded Blockchains

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

Abstract— **Sharding is 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, the adversary in such networks can launch the *single-shard takeover attack* by compromising a single shard's consensus. To address these issues, participants should be securely and dynamically allocated into different shards. 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 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 that limit their performance, and their extreme choices between two performance metrics: *self-balance* and *operability*. We observe and prove the fundamental trade-off between these two metrics, and identify a new property *memory-dependency* that enables parametrisation over this trade-off. Based on these insights, we propose WORMHOLE, a correct and efficient shard allocation protocol with minimal security assumptions and parametrisable *self-balance* and *operability*. We implement WORMHOLE and evaluate its overhead and performance metrics in a network with 128 shards and 32768 nodes. The results show that WORMHOLE introduces little overhead, achieves consistent self-balance and operability with our theoretical analysis, and allows the system to recover quickly from load imbalance.

I. INTRODUCTION

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

Given the success of permissioned sharded systems, sharding is regarded as a promising technique for scaling permissionless blockchains, where nodes can join and leave the system at any time. However, permissionless systems need to tolerate Byzantine nodes that may attack the system, whereas traditional sharded systems [4]–[8] only need to tolerate crash faults. In sharded blockchains, the adversary can launch *single-shard takeover attacks* (aka *1% attacks*) [9], [10] by gathering its nodes to a single shard and compromise a shard's consensus. As voting power is split among shards, launching such attacks requires much fewer nodes compared to 51% attacks in non-sharded blockchains. To resist single-shard takeover attacks, sharded blockchains should prevent nodes from choosing shards freely. Without a global view of the network and centralised membership management, a common solution is to randomly allocate nodes into shards.

Achieving the optimal resistance against single-shard takeover attacks requires *load balance*, so that each shard contains a comparable number of nodes. Otherwise, shards with few nodes are likely to be compromised. Meanwhile, permissionless systems suffer from node churn [11]: nodes may join or leave the system at any time. To achieve load balance under node churn, permissionless sharding protocols need to adaptively re-balance nodes over time. An intuitive solution is to randomly shuffle all nodes for every epoch. However, when a node moves to a new shard, it needs to synchronise the new shard's ledger and find new peers, which introduces non-negligible overhead and makes the node temporarily unavailable. The blockchain community recognises this issue as the *reshuffling* problem [12], [13].

To address the above issues, sharded blockchains should employ a mechanism that allocates nodes into shards securely, randomly, and dynamically. Existing works on designing [14]–[20] and analysing [21]–[23] sharded permissionless blockchains focus on cross-shard communication and intra-shard consensus. A systematic study on this core component, which we call *shard allocation*, is still missing. Figure 1 provides an example of shard allocation. Five nodes are allocated in shard #3 and four of them later left the system. To prevent the only node in shard #3 from becoming a single point of failure, the system has to allocate some nodes to shard #3 to re-balance the shards.

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

[¶] Corresponding author.

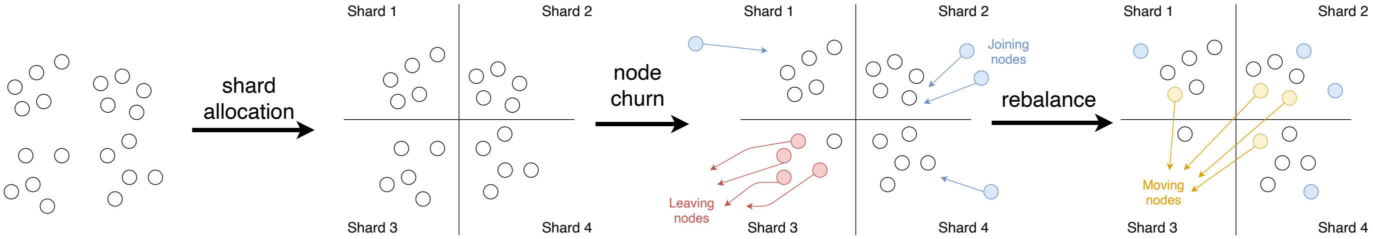


Fig. 1: An example of shard allocation. New nodes (in blue) may join the system and existing nodes (in red) may leave the system. After a state update, a subset of nodes (in yellow) may be relocated.

protocol for permissionless blockchains. Our contributions are summarised as follows.

1) We **provide the first study on formalising the shard allocation protocol for permissionless blockchains (§II)**. The formalisation includes the syntax, correctness properties and performance metrics, and can be used as a framework for evaluating shard allocation protocols.
2) Based on our framework, we **evaluate the shard allocation subprotocols of seven state-of-the-art permissionless sharded blockchains (§III)**, including five academic proposals Elastico [14] (CCS'16), Omniledger [15] (S&P'18), RapidChain [16] (CCS'19), Chainspace [17] (NDSS'19), and Monoxide [18] (NSDI'19), and two industry projects Zilliqa [19] and Ethereum 2.0 [24]. 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* (§IV-A).** Self-balance represents the ability to re-balance the number of nodes in different shards; and operability represents the system performance w.r.t. the cost of re-allocating nodes to a different shard. While this impossibility has been conjectured [12], [13] and studied informally [15], we formally prove it is impossible to achieve optimal values on both, and quantify the trade-off between them. All existing sharded blockchains except for Omniledger make extreme choices on either *self-balance* or *operability*, leading to serious security or performance issues.
4) We **identify and define a property *memory-dependency* that is necessary for shard allocation protocols to support parametrisation over the trade-off between *self-balance* and *operability* (§IV-B)**. *Memory-dependency* (aka *non-memorylessness* in signal processing literatures [25]) specifies that the shard allocation relies on both the current and previous system states. The parametrisation support opens a new in-between design space and makes the system configurable for different application scenarios. We formally prove the necessity of *memory-dependency* for supporting such parametrisation.
5) We **propose** WORMHOLE**, a correct and efficient shard allocation protocol (§V), and analyse how to integrate** WORMHOLE **into sharded blockchains (§VI)**. We formally prove that WORMHOLE achieves all correctness properties, and supports parametrisation of *self-balance* and *operability*. We also classify existing sharded blockchains, and analyse how to integrate WORMHOLE into each type of them.
6) We **implement** WORMHOLE**, and evaluate its overhead and performance metrics in realistic settings (§VII).** We implement WORMHOLE as a prototype in Rust, and evaluate the overhead of integrating WORMHOLE into different designs of sharded blockchains. We simulate WORMHOLE in a sharded blockchain with 128 shards and 32768 nodes, and evaluate the dynamic load balance and operability under different churn conditions. The results show that WORMHOLE achieves consistent load balance and operability with our theoretical analysis, and can recover quickly from load imbalance.

II. FORMALISING SHARD ALLOCATION

In this section, we formalise the shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$, including its system model, syntax, correctness properties, and performance metrics.

A. Overview of a sharded blockchain

A sharded blockchain consists of a fixed number of m shards, each of which maintains a ledger (organised as a chain of blocks) and processes transactions concurrently. The sharded blockchain proceeds in epochs, whose definition differs according to the underlying system. Generally speaking, an epoch t begins when a new global and unique system state st_t is available. Given the new system state st_t as input, new nodes and existing nodes execute the shard allocation protocol to obtain a new shard membership w.r.t. st_t. Then, nodes find peers in the same shard by exchanging shard memberships/proofs if necessary, and execute the consensus protocol to agree on new blocks. Each block includes the block proposer's shard membership and proof, apart from other data common in blockchains. *Node churn* [11] happens at any point of the protocol execution: some new nodes join and some existing nodes leave the sharded blockchain. As we study shard allocation across epochs, we consider node churn happens at the end of each epoch for simplicity.

Epoch and global system state. Most sharded blockchains demand that the system state can be accessed by nodes securely and synchronously. How and when a system state is generated depends on the concrete protocol design. For example, Elastico [14], Omniledger [15], RapidChain [16] and Ethereum 2.0 [24] use a decentralised randomness beacon (DRB) protocol to generate random outputs as system states; Zilliqa [19] merges blocks from all shards in an epoch to a single one, then extracts a global system state from the merged block.

Sybil resistance. To defend against Sybil attacks, where the adversary spawns numerous nodes to compromise the shard's consensus, permissionless sharded blockchains must employ Sybil-resistant mechanisms. For example, Elastico, RapidChain, and Zilliqa require nodes to solve PoW puzzles to obtain shard memberships; Monoxide and Ethereum 2.0 employ Sybil-resistant consensus protocols; and Omniledger employs an identity authority to manage nodes.

B. System model

In a sharded blockchain with m shards, each node i has a pair of secret key sk_i and public key pk_i, and is identified by pk_i. Let n_k^t be the number of nodes in shard $k \in [m]$ and $n^t = \sum_{k=1}^{m} n_k^t$ be the total number of nodes in epoch t, where $[m] = \{1, 2, \ldots, m\}$. To focus on analysing the shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$, we assume the system state generation protocols are secure, and the sharded blockchain has employed a Sybil-resistant mechanism so that the number of nodes controlled by the adversary is bounded. In line with existing proposals, nodes are assumed to have secure and synchronous access to the latest system state. The system model consists of the four following aspects.

Node churn. Let $\alpha_t \geq 0$ and $\beta_t \in [0, 1]$ be the fraction of nodes joining and leaving at the end of epoch t, respectively. For two consecutive epochs t and $t+1$, $n^{t+1} = (1-\beta_t+\alpha_t)n^t$.

Network model. The network model concerns the timing guarantee of delivering messages. Depending on different proposals' settings, the network model is either synchrony, partial synchrony [26], or asynchrony. A network is synchronous if messages are delivered within a known finite time-bound; partially synchronous if messages are delivered within a known finite time-bound after an unknown Global Stabilisation Time (GST); or asynchronous if messages are delivered with an unknown time-bound possibly determined by the adversary.

Adversary's capacity. Let ϕ be the fault tolerance capacity of $\Pi_{\mathsf{ShardAlloc}}$, where ϕ is no bigger than the consensus protocol's fault tolerance capacity Ψ. Otherwise, even when the adversary's nodes are evenly distributed among shards, the adversary can compromise every shard. The adversary is adaptive: at any time, it can corrupt any ϕn^t nodes, i.e., make these nodes Byzantine, where t is the epoch number. The adversary can read internal states of corrupted nodes, and direct corrupted nodes to arbitrarily forge, modify, delay, and/or drop messages from them. The adversary can read and/or delay messages from correct nodes. The delay period is subjected to the network model (e.g., synchrony, partial synchrony, or asynchrony) assumed by the sharded blockchain.

Adversary's goal. The adversary aims at breaking some of $\Pi_{\mathsf{ShardAlloc}}$'s correctness properties that we will define later in §II-D. To compromise $\Pi_{\mathsf{ShardAlloc}}$'s safety, the adversary has to launch the *single-shard takeover attack* by gathering more than Ψn_k^t corrupted nodes into a certain shard. To compromise shard allocation's liveness, the adversary has to prevent correct nodes from determining their shards.

To compromise safety, here are some typical attacking strategies: 1) the adversary biases the shard allocation and gather its nodes to a single shard; 2) the adversary predicts the shard allocation results, then chooses key pairs that make nodes to be allocated to the same shard; 3) for probabilistic shard allocation protocols, the adversary can launch the *join-leave attack* [27], i.e., enforce corrupted nodes to keep joining and leaving the blockchain until allocated to the targeted shard. To compromise liveness, if $\Pi_{\mathsf{ShardAlloc}}$ requires nodes to interact with each other, then the adversary can refuse to collaborate and stall the protocol execution.

C. Syntax

We formally define the shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ as follows.

Definition 1 (Shard allocation $\Pi_{\mathsf{ShardAlloc}}$)**.** *A shard allocation protocol* $\Pi_{\mathsf{ShardAlloc}}$ *is a tuple of polynomial time algorithms*

$$\Pi_{\mathsf{ShardAlloc}} = (\mathsf{Setup}, \mathsf{Join}, \mathsf{Update}, \mathsf{Verify})$$

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

$\mathsf{Join}(pp, sk_i, st_t) \rightarrow (k, \pi_{i,st_t,k})$ *: On input secret key* sk_i*, 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*. The input may also be public key* pk_i *of node i, depending on concrete constructions. This also applies to* $\mathsf{Update}(\cdot)$.

$\mathsf{Update}(pp, sk_i, st_t, k, \pi_{i,st_t,k}, st_{t+1}) \rightarrow (k', \pi_{i,st_{t+1},k'})$ *: On input the public parameter pp, secret key* sk_i*, state* st_t*, 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 shard assignment proof* $\pi_{i,st_{t+1},k'}$.

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

Algorithm 1 describes the typical execution of $\Pi_{\mathsf{ShardAlloc}}$ in a sharded blockchain, from a node i's perspective. $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Setup}(\lambda)$ is executed once at the beginning of the protocol execution. When node i joins the system in epoch t, it executes $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ to obtain a shard membership k_* and the associated membership proof π_{i,st_*,k_*}. Afterwards, the node executes consensus with peers in shard k_*. Upon a new epoch $t+1$, node i executes $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ to update its shard membership and the membership proof, and executes consensus accordingly. Given a shard k, public key pk_i and a membership proof π_{i,st_t,k_t}, anyone can execute

$\Pi_{\mathsf{ShardAlloc}}.\mathsf{Verify}(\cdot)$ to verify whether node i is allocated into shard k in epoch t.

Algorithm 1: Typical execution of shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ in a sharded blockchain, from node i's perspective.

```
// Join in epoch $t$
$(k_t, \pi_{i,st_t,k_t}) \leftarrow \Pi_{\mathsf{ShardAlloc}}.\mathsf{Join}(pp, sk_i, st_t)$
// State, shard and proof in epoch $t$
$st_*, k_*, \pi_* \leftarrow st_t, k_t, \pi_{i,st_t,k_t}$
repeat
    Wait for a new state $st_+$
    // Update shard membership and proof
    $(k_*, \pi_{i,st_*,k_*}) \leftarrow$
      $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(pp, sk_i, st_*, k_*, \pi_{i,st_*,k_*}, st_+)$
    $st_* \leftarrow st_+$
    // Messages may attach $k_*$ and $\pi_{i,st_*,k_*}$
    Execute consensus with peers in shard $k_*$
until node $i$ leaves the system
```

D. Correctness properties

Based on the system model, we consider three correctness properties for $\Pi_{\mathsf{ShardAlloc}}$, namely *liveness*, *allocation-randomness*, and *unbiasibility*. We additionally consider *allocation-privacy* as an optional property.

Liveness. This property ensures that correct nodes can obtain valid shard memberships *timely*: given a system state, all correct nodes will finish computing $\mathsf{Update}(\cdot)$ (or $\mathsf{Join}(\cdot)$ if the node newly joins the system) before the next epoch. Otherwise, nodes cannot find their shards or participate in consensus, and consequently, the block producing is stalled.

Definition 2 (Liveness). *A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ satisfies liveness iff for every epoch t, every correct node i will finish computing $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(pp, sk_i, st_{t-1}, k_{t-1}, \pi_{i,st_{t-1},k_{t-1}}, st_t)$ (or $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Join}(pp, sk_i, st_t)$ if $t = 1$) before epoch $t+1$ such that $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Verify}(pp, pk, +i, st_t, k_t, \pi_{i,st_t,k_t}) = 1$, where pp is the public parameter, sk_i is node i's secret key, $(st_{t-1}, k_{t-1}, \pi_{i,st_{t-1},k_{t-1}})$ and $(st_t, k_t, \pi_{i,st_t,k_t})$ are the system state, node i's allocated shard and node i's shard membership proof in epoch $t-1$ and t, respectively.*

Allocation-randomness. This property ensures that each node is allocated to a random shard deterministically [14], [15], [19]. Otherwise, if the adversary can predict shard allocation results, then it can launch the *single-shard takeover attack* by corrupting nodes that will be allocated to a specific shard. 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.

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

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

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

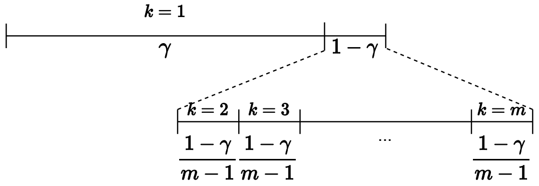


Fig. 2: *Update-randomness.* After executing $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$, the probability that a node stays in its shard (say shard 1) is γ, and the probability of moving to each other shard is $\frac{1-\gamma}{m-1}$.

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 computationally and communicationally intensive, as a node needs to synchronise and verify the new shard's ledger, which can take hundreds of Gigabytes [15], [28]–[30]. 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 be moved within each state update. We define γ as the probability that a node stays in the same shard after a state update. We define *update-randomness* as follows.

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

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

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

When $\gamma = \frac{1}{m}$, $\Pi_{\mathsf{ShardAlloc}}$ achieves optimal *update-randomness*, as all nodes are shuffled randomly under uniform distribution. This definition is intuitively depicted in Figure 2.

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

Unbiasibility. This property ensures that the adversary cannot manipulate the shard allocation results. While *allocation-randomness* defines the probability distribution of shard allocation, *unbiasibility* rules out attacks on manipulating the probability distribution, e.g., the join-leave attack [27], [31].

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

Allocation-privacy. This property ensures that no one can learn a node's shard membership without the node providing them by itself. Compared to *allocation-randomness*, *allocation-privacy* further prevents the adversary from computing a node's membership if the adversary has no access to the node's secret key. We consider *allocation-privacy* to be optional, as it has both advantages and disadvantages. On the positive side, *allocation-privacy* is necessary for the sharded blockchain to resist against the adaptive adversary: if the adversary cannot learn others' shard memberships, then it cannot corrupt nodes in a specific shard, but only a random set of nodes scattered across shards. On the negative side, *allocation-privacy* makes nodes difficult to find peers in the same shard. If the sharded blockchain employs a consensus protocol that requires broadcasting operations, then nodes have to execute an extra peer finding protocol [14], [19] before executing consensus, introducing non-negligible communication overhead. Thus, if the sharded blockchain is not required to resist against an adaptive adversary, then $\Pi_{\mathsf{ShardAlloc}}$ does not need to achieve *allocation-privacy*.

Definition 7 (Join-privacy)**.** *A shard allocation protocol* $\Pi_{\mathsf{ShardAlloc}}$ *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*

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

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

Definition 8 (Update-privacy)**.** *A shard allocation protocol* $\Pi_{\mathsf{ShardAlloc}}$ *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*

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

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

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

E. Performance metrics

While correctness properties are required for the shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$, performance metrics are used for quantitatively measuring the protocol's effectiveness. We consider three performance metrics as follows.

Communication complexity. Communication complexity is the amount of communication (measured by the number of messages) required to complete a protocol [32]. For shard allocation, we consider the *communication complexity* of all correct nodes obtaining shard memberships when joining, and updating shard memberships upon a new epoch. The communication of synchronising new shards is omitted.

Self-balance. 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 [18], [33]. Due to node churn and lack of a 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 $\Pi_{\mathsf{ShardAlloc}}$ recovers from load imbalance.

Definition 10 (Self-balance)**.** *When executing* $\Pi_{\mathsf{ShardAlloc}}$ *on m equal-sized shards in epoch t (i.e., $n_i^t = n_j^t$ for all $i, j \in [m]$),* $\Pi_{\mathsf{ShardAlloc}}$ *is μ-*self-balanced *iff*

$$\mu = 1 - \max_{\forall i,j \in [m]} \frac{|n_i^{t+1} - n_j^{t+1}|}{n^t}$$

Value μ measures the level of imbalance among shards after an epoch. When $\mu = 1$, $\Pi_{\mathsf{ShardAlloc}}$ achieves the optimal *self-balance*: regardless of how many nodes join or leave the system during the last epoch, the system can balance itself within an epoch.

Operability. To balance shards, $\Pi_{\mathsf{ShardAlloc}}$ 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 [15]. We define *operability* as the probability that a node stays at its shard upon a state update. Operability is straightforward to compute for protocols satisfying update-randomness (Definition 4): if $\Pi_{\mathsf{ShardAlloc}}$ satisfies update-randomness with γ (Definition 4), then its operability is γ, i.e., γ-*operable*. When $\gamma = 1$, $\Pi_{\mathsf{ShardAlloc}}$ is most *operable*: nodes will never move after joining the network.

III. 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 I) shows that none of them is fully correct or achieves satisfactory performance.

A. Evaluation criteria

The evaluation framework includes the system model, correctness properties (§II-D), and performance metrics (§II-E). The system model concerns the network model and fault tolerance capacity mentioned in §II-B, plus the trusted components that some proposals assume in order to guarantee the correctness. The node churn and adversary's goal mentioned in §II-B are common in all proposals, and thus are omitted.

As the evaluation framework focuses on shard allocation, other subprotocols in sharded blockchains – e.g., system state

TABLE I: Evaluation of seven permissionless shard allocation protocols. Red indicates strong assumptions, unsatisfied correctness properties, and relatively weaker 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	Trusted components	Fault tolerance	Public verifiability	Liveness	Allocation-rand.	Unbiasibility	Privacy^o	Join comm. compl.	Update comm. compl.	Self-balance	Operability
Elastico	New block	Sync.	-	$\frac{1}{3}$	✓	✓	✓	✗	✓	$O(n^f)$	$O(n^f)$	1	$\frac{1}{m}$
Omniledger	Identity authority	Part. sync.	Identity authority	$\frac{1}{3}$	✓	✓	✓	✓	✗	$O(n)$	$O(n) \sim O(n^3)$	$1 - \frac{(2m-3)\beta_t}{3m-3}$	$\frac{2}{3}$
RapidChain	Nodes joining	Sync.	-	0	✗	✓	✓	✓	✗	$O(n^2)$	$O(n^2)$	$1 - \beta_t$	$max(1 - \kappa\alpha_t n, 0)$
Chainspace	-	Async.	Smart contracts	1	✓	✓	✗	✗	✗	-	$O(n)$	$1 - \beta_t$	-
Monoxide	-	Async.	-	1	✓	✓	✗	✓	✗	0	0	$1 - \beta_t$	1
Zilliqa	New block	Async.	-	1	✓	✓	✓	✗	✓	$O(n)$	$O(n)$	1	$\frac{1}{m}$
Ethereum 2.0	-	Async.	-	1	✓	✓	✗	✓	✗	0	0	$1 - \beta_t$	1
WORMHOLE (Our proposal in §V)	New rand.	Async.	Rand. Beacon*	1	✓	✓	✓	✓	✓	$O(n)$	$O(n)$	$1 - \beta_t + \frac{\beta_t}{2^{op}}$	$1 - \frac{m-1}{m \cdot 2^{op}}$

^{o} Optional. $*$ WORMHOLE can rely on an external randomness beacon, or allow a group of nodes to run a decentralised randomness beacon protocol similar to Elastico, Omniledger, RapidChain and Ethereum 2.0.

generation, consensus and cross-shard communication – are assumed to be secure.

Remark on fault tolerance capacity. We define the fault tolerance capacity only for shard allocation rather than the sharded blockchain, where other subprotocols 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 to be 1, as it satisfies all correctness properties even when all nodes are Byzantine.

B. Overview of evaluated proposals

We choose seven state-of-the-art sharded blockchains, including five academic proposals Elastico [14], Omniledger [15], Chainspace [17], RapidChain [16], and Monoxide [18], and two industry projects Zilliqa [19] and Ethereum 2.0 [24]. We briefly describe their shard allocation protocols below, and defer their details to Appendix D.

Elastico, Omniledger and RapidChain rely on distributed randomness generation (DRG) protocols for shard allocation. In Elastico, nodes in a special shard called *final committee* run a commit-and-reveal DRG protocol [34] to produce a random output. Each node then solves a PoW puzzle derived from the random output and its identity, and will be assigned to a shard according to its PoW solution. Omniledger uses the *RandHound* [35] DRG protocol, and relies on a trusted identity authority who samples pending nodes (who wait to join the system) and shuffles existing nodes. 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 [36]. Given the latest random output, the identity authority samples a subset of pending nodes to join the system, and shuffles $\frac{1}{3}$ existing nodes in the network to different shards. In RapidChain, each node solves a PoW puzzle before joining the system. To prevent precomputing PoW puzzles, the PoW puzzle's input includes a random output, which is generated by a Feldman Verifiable Secret Sharing (VSS) [37]-based DRG protocol. A special shard called *reference committee* allocates joined nodes into different shards following the *Commensal Cuckoo* rule [38]: each node is mapped to an ID $x \in [0, 1)$, and when a new node joins with ID x, the reference committee moves nodes with IDs close to x to other shards randomly.

Chainspace, Monoxide, Zilliqa and Ethereum 2.0 do not rely on DRG protocols for shard allocation. In Chainspace, a node can apply to move to another shard at any time, and other nodes vote to decide on the applications. The voting works over a special smart contract `ManageShards`, whose execution is assumed to be correct and trustworthy. Monoxide and Ethereum 2.0 allocate nodes into different shards according to their addresses' prefixes. Zilliqa is built upon Elastico, but it uses the last block's hash value as the current epoch's random output.

C. System model

Network model. A protocol is synchronous if its safety is broken when the adversary can delay messages more than the upper bound Δ. Therefore, Elastico and RapidChain are synchronous, as they employ the synchronous DRG protocols [34], [37]. A protocol is partially synchronous if such $> \Delta$ delay only affect liveness but not safety, and liveness can be resumed once the network becomes synchronous. Therefore, Omniledger assumes partially synchronous networks, as it employs the partially synchronous*RandHound* DRG protocol. A protocol is asynchronous if such $> \Delta$ delay only affect liveness but not safety, and liveness can be resumed once certain messages are delivered. Chainspace, Monoxide, Zilliqa and Ethereum 2.0 assume asynchronous networks: in Chainspace, a node submits a smart contract transaction to obtain or update a shard membership, and the liveness is achieved once the transaction is received by the smart contract; Monoxide and Ethereum 2.0 allow nodes to calculate shards locally without communicating with others; and Zilliqa replaces the DRG [34] in Elastico by using block hashes as system states that can be accessed synchronously by assumption, and nodes can calculate their shards locally given the system states.

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

Fault tolerance capacity. Elastico and Omniledger achieve the fault tolerance capacity of $\phi = \frac{1}{3}$, which is inherited from their DRG protocols. RapidChain cannot tolerate any faults, as one faulty node can make the Feldman VSS lose liveness by withholding shares. Chainspace, Monoxide, Zilliqa, and Ethereum 2.0 can tolerate any fraction of Byzantine nodes. For Monoxide and Ethereum 2.0, computing shards is offline. Chainspace assumes trusted smart contracts. For Zilliqa, blocks are produced correctly by assumption and shard computation is offline.

D. Correctness properties

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.

Liveness. All shard allocation protocols satisfy *liveness*.

Allocation-randomness. Elastico, Omniledger, RapidChain, and Zilliqa satisfy *allocation-randomness*, as all nodes are shuffled for each epoch. Chainspace does not satisfy *allocation-randomness*, as nodes can choose which shard to join. Monoxide and Ethereum 2.0 do not satisfy *allocation-randomness*, as nodes can choose their preferred shards by choosing addresses.

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 less challenging to solve, allowing the adversary to solve multiple puzzles within an epoch and choose a preferred shard to join. Chainspace does not achieve *unbiasibility*, as it does not satisfy *allocation-randomness* and nodes are free to choose shards.

Allocation-privacy. Elastico and Zilliqa satisfy *allocation-privacy*, as the allocated shard remains secret if the node does not reveal its PoW solution. Therefore, Elastico and Zilliqa employ an extra peer finding mechanism called "overlay setup", where a special shard called "directory committee" collects and announces nodes' allocated shards. 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 Ethereum 2.0 do not satisfy *allocation-privacy*, as nodes' addresses are publicly known.

E. Performance metrics

Communication complexity. Elastico's shard allocation requires $O(n^f)$ messages per epoch, where n and f are the number of nodes and faulty nodes, respectively. For each epoch, the final committee needs to run the DRG protocol, which consists of a vector consensus [39] with communication complexity $O(n^f)$. Ideally, the final committee in Elastico has $\frac{n}{m}$ nodes, and the communication complexity is $O(\frac{n}{m}^{\frac{f}{m}}) = O(n^f)$ (m is constant). For Omniledger, $\mathsf{Join}(\cdot)$ requires $O(n)$ communication, as each node requests to the identity authority for joining the system. The communication complexity of $\mathsf{Update}(\cdot)$ is $O(n)$ or $O(n^3)$: the best case of $\mathsf{Update}(\cdot)$ is that the leader election and RandHound are both successful, leading to $O(n)$ messages; and the worst case is that nodes fallback to run the asynchronous DRG [36] with communication complexity $O(n^3)$. RapidChain's shard allocation requires $O(n^2)$ messages per epoch, which is inherited from Feldman VSS [37]. Monoxide and Ethereum 2.0 requires no communication for shard allocation, as nodes decide their shards locally. Zilliqa requires $O(n)$ messages per epoch as each node needs to retrieve the latest block.

Self-balance. The *self-balance* of Elastico and Zilliqa is 1, as all nodes are shuffled for each epoch. In Omniledger, $\frac{1}{3}$ nodes are shuffled for each epoch, leading to operability $\gamma = \frac{2}{3}$. According to Lemma 1 (which will be introduced later in §IV-A), Omniledger's *self-balance* $\mu = 1 - \frac{(2m-3)\beta_t}{3m-3}$. The *self-balance* of RapidChain, Chainspace, Monoxide and Ethereum 2.0 is $1 - \beta_t$. In the worst case where no nodes newly join the system and βn^t nodes in the same shard leave the system, self-balance becomes $\frac{n - \beta_t n}{n} = 1 - \beta_t$.

Operability. The *operability* of Elastico and Zilliqa are $\frac{1}{m}$, as all nodes are shuffled for each new epoch. The *operability* of Omniledger is $\gamma = \frac{2}{3}$, as $\frac{1}{3}$ nodes are shuffled for each epoch. 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 join churn rate in epoch t. In epoch t, there are $\alpha_t n$ nodes joining the network, and 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 Ethereum 2.0 have the *operability* of 1, as nodes in Monoxide and Ethereum 2.0 never move to other shards.

IV. Observation and insights

Table I shows that no shard allocation protocols achieves optimal *self-balance* and *operability* simultaneously. We formally prove that it is impossible to achieve optimal values on both simultaneously. We then identify a new property *memory-dependency* that enables parametrisation of the trade-off between them. The parametrisation provides a new in-between design space, and can make shard allocation protocols configurable for different application scenarios.

A. Impossibility and trade-off

According to Table I, except for Omniledger and RapidChain, *self-balance* μ is either $1 - \beta_t$ or 1, and *operability* γ is either 1 or $\frac{1}{m}$. This shows that achieving optimal *self-balance* and *operability* simultaneously still remains as an open problem. Its possibility has also been discussed in the

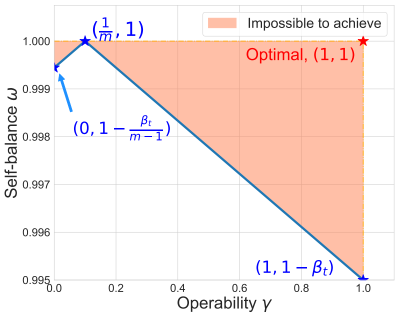


Fig. 3: Relationship between *operability* and *self-balance*. We pick $m = 10$ and $\beta_t = 0.005$ as an example. No shard allocation protocol can go above the blue line to reach the orange area.

blockchain community [12], [13]. We prove that, however, this is impossible for any correct shard allocation protocol. The proof starts from analysing the relationship between *self-balance* μ and *operability* γ.

Lemma 1. *If a correct shard allocation protocol* $\Pi_{\mathsf{ShardAlloc}}$ *with* m *shards satisfies update-randomness with* γ*, the self-balance of* $\Pi_{\mathsf{ShardAlloc}}$ *is* $\mu = 1 - \left|\frac{(\gamma m-1)\beta_t}{m-1}\right|$ *, where* β_t *is the percentage of nodes leaving the network in epoch* t*.*

Appendix A provides the proof of Lemma 1. Figure 3 visualises the relationship between *self-balance* and *operability* revealed by Lemma 1. The line never reaches the point $(1, 1)$, indicating that $\Pi_{\mathsf{ShardAlloc}}$ can never achieve both optimal *self-balance* and optimal *operability*. With *operability* increasing, the *self-balance* increases to 1 when $\gamma \leq \frac{1}{m}$, then decreases when $\gamma \geq \frac{1}{m}$. 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 nodes, during $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ it loses fewer nodes but receives more nodes from other shards. When $\gamma = \frac{1}{m}$, *self-balance* becomes 1, i.e., optimal.

Therefore, it is impossible to achieve optimal values for both *self-balance* and *operability* simultaneously. Theorem 1 formally states the impossibility, and Appendix A provides the full proof of Theorem 1.

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

B. Parametrising the trade-off

As shown in Figure 3, $(1, 1 - \beta_t)$ and $(\frac{1}{m}, 1)$ are two extreme cases in the trade-off 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 prove that, to parametrise this trade-off, sharding protocols should be *memory-dependent*, where the shard allocation result does not only depend on the current system state, but also the previous ones. In signal processing literatures, this property is also known as *non-memorylessness*, where the output signal does not only depend on the current input, but also some previous inputs [25]. Formally, memory-dependency is defined as follows.

Definition 11 (Memory-dependency)**.** *A shard allocation protocol* $\Pi_{\mathsf{ShardAlloc}}$ *is memory-dependent iff for any secret key* sk_i*, public parameter* pp*, and shard* k*, the output of* $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(pp.sk_i, st_t, k, \pi_{i,st_t,k}, st_{t+1})$ *depends on system states earlier than* st_t*.*

By Definition 4, both *self-balance* and *operability* are related to the probability γ of nodes staying at the same shard. To parametrise *self-balance* and *operability*, a shard allocation protocol should incorporate shard allocation results of previous epochs. To summarise, when $\gamma \in (\frac{1}{m}, 1)$, the probability distribution of *allocation-randomness* is non-uniform, and the membership proof of each epoch t depends on that in the previous epoch $t-1$. As the membership proof of epoch $t-1$ also depends on that of epoch $t-2$, recursively, each membership proof depends on all historical membership proofs. Appendix A provides the proof of Theorem 2.

Theorem 2. *If a correct shard allocation protocol* $\Pi_{\mathsf{ShardAlloc}}$ *is* μ*-self-balanced and* γ*-operable where* $\mu \in (0, 1-\beta_t)$ *and* $\gamma \in (\frac{1}{m}, 1)$*, then* $\Pi_{\mathsf{ShardAlloc}}$ *is memory-dependent.*

V. WORMHOLE: MEMORY-DEPENDENT SHARD ALLOCATION

Based on the gained insights, we propose WORMHOLE, a correct and efficient shard allocation protocol. WORMHOLE relies on a randomness beacon (RB) to generate the system states, and a verifiable random function (VRF) (e.g., [40]–[42]) to guide the nodes in computing their shards. As WORMHOLE does not require nodes to interact with each other, it works correctly regardless of the network environment or the portion of Byzantine nodes. By being memory-dependent, WORMHOLE supports parametrisation of *self-balance* and *operability*. We formally analyse WORMHOLE's correctness, and its communication and computational complexity.

A. Primitives: RB and VRF

Randomness beacon. Similar to existing sharded blockchains such as Elastico, Omniledger, and Zilliqa, WORMHOLE allocates nodes based on some randomness. RB [43] is a service that periodically generates random outputs. The RB can be instantiated by either an external party or by a group of nodes via a decentralised randomness beacon (DRB) protocol. RB satisfies the following properties [44]:

- *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 to its advantage.

- *RB-Public-Verifiability*: Everyone can verify the correctness of the random output.

RB schemes are both readily available and widely used. Public external RBs are maintained by countries such as the US [43], Chile [45], and Brazil [46], as well as reputable institutions such as Cloudflare [47], EPFL [48], and League of Entropy [49]. DRB protocols can be constructed from Publicly Verifiable Secret Sharing (PVSS) [35], [44], [50], Verifiable Delay Functions [51], [52], Nakamoto consensus [53], and real-world entropy [54], [55]. Several sharded blockchains, including Elastico, Omniledger, and RapidChain, employ DRB to produce the system states already; Ethereum 2.0 uses DRB for its consensus; emerging projects such as Filecoin [56] rely on an external RB for its consensus.

Verifiable random function. A VRF [40] is a public-key version of a hash function, which computes an output and a proof from an input string and a secret key. Anyone with the associated public key and the proof can verify 1) whether the output is from the input, and 2) whether the output is generated by the owner of the secret key. Some VRFs support *batch verification* [57], [58], i.e., verifying multiple VRF outputs at the same time, which is faster than verifying VRF outputs one-by-one. Formally, a VRF is a tuple of four algorithms:

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

VRF should satisfy the following three properties [59].

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

B. Protocol design

Key challenge and strawman designs. The key challenge in designing a memory-dependent shard allocation protocol is the *recursive dependency* problem: to verify a node's shard membership in epoch t, the node needs to prove its shard membership in epoch $t-1$ (i.e., "the memory"); however, verifying the shard membership in epoch $t-1$ requires to verify that in epoch $t-2$, and so on. Therefore, an extra mechanism is necessary to bound the number of history proofs.

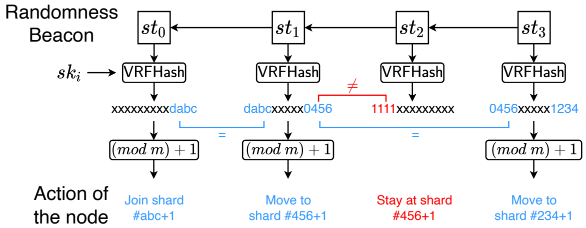


Fig. 4: Intuition of WORMHOLE $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$. All numbers are in hexadecimal. We use $op = 4$ and $m = 16^3$ as an example, and assume epoch 0 is the last non-memory-dependent epoch.

A strawman design is to prescribe a fixed number of history proofs, so that all shard allocations but the earliest one is verifiable. However, this approach allows an attacker to enumerate all the shards as the earliest shard, and only releases one that leads them to the target shard, similar to a well-known *grinding attack* against proof-of-stake protocols.

Another strawman design is to periodically abandon memory-dependency, so that nodes only need to provide history proofs up to the last non-memory-dependent epoch. Specifically, we define each w-epoch unit as an *era*, which begins when $t \mod w = 0$ and ends when $t \mod w = t-1$, where t is the epoch number. At each era's beginning, every node reshuffules its shard, and the first shard membership is non-memory-dependent, i.e., deterministic with its secret key and the system state. However, simultaneous global reshuffling temporarily lowers the operability, as all nodes have to find new peers in their new shards and synchronize the ledgers.

Our basic idea. WORMHOLE addresses the above challenge by (1) prescribing a non-memory-dependent shard allocation per node per era and (2) randomising this special reallocation epoch for each node, so that the number of history proofs in a membership proof is bounded and all nodes are not reshuffled simultaneously.

The non-memory-dependent shard allocation epoch and the target shard are computed as follows. When an era starts at epoch t (when $t \mod w = 0$), node i calculates $\mathsf{VRFEval}(sk_i, st_t) \rightarrow (g_{i,t}, \pi_{i,t})$, where st_t is RB's output, i.e., the system state, in epoch t. The node will then move to shard $(g_{i,t} \mod m) + 1$ at epoch $t + (g_{i,t} \mod w)$. Note that both the reallocation epoch and the target shard are non-memory-dependent, and this happens exactly once per era.

Shard allocation in the other $w-1$ epochs are memory-dependent. At epoch t, node i computes $\mathsf{VRFEval}(sk_i, st_t) \rightarrow (h_{i,t}, \pi_{i,t})$. A parameter op is used for balancing *operability* and *self-balance*. The node determines whether it needs to move to another shard by comparing op least significant bits (LSB) of $h_{i,t-1}$ and op most significant bits (MSB) of $h_{i,t}$: when $\mathsf{LSB}(op, h_{i,t-1}) \neq \mathsf{MSB}(op, h_{i,t})$, the node stays in the same shard, i.e., $k_{i,t} = k_{i,t-1}$; otherwise $k_{i,t} = (h_{i,t} \mod m) + 1$. Increasing op improves *operability* but reduces *self-balance*, and vice versa. Figure 4 illustrates this idea.

A membership proof includes the g_i values of the current and previous eras' non-memory-dependent epochs, and all h_i

values since the last non-memory-dependent epoch, as well as their corresponding proofs.

Detailed process. Algorithm 2 provides the full construction of WORMHOLE. When node i joins the system, it executes $\mathsf{Join}(\cdot)$: it calculates VRF outputs and proofs since the last non-memory-dependent epoch, and executes $\mathsf{calcShard}(\cdot)$ to calculate its allocated shard ID k. The shard membership proof $\pi_{i,st_t,k}$ includes a sequence of VRF outputs $(h_{last}, \ldots, h_t)$ and their VRF proofs $(\pi_{last}, \ldots, \pi_t)$, where $last$ is the last non-memory-dependent epoch calculated from $\mathsf{calcNMDEpoch}(\cdot)$.

Upon epoch $t+1$, node i executes $\mathsf{Update}(\cdot)$ as follows. It first calculates one more VRF output of st_{t+1}. If epoch $t+1$ is memory-dependent, then $\mathsf{calcShard}(\cdot)$ only needs to check if $\mathsf{MSB}(op, h_{t+1}) = \mathsf{LSB}(op, h_{\text{idx}})$ and compute idx and shard_id accordingly, where h_{idx} is cached from epoch t. If epoch $t+1$ is non-memory-dependent, then the previous proofs are discarded and the shard ID is $(h_{t+1} \mod m) + 1$.

To verify a membership proof $\pi_{i,st_t,k}$, $\mathsf{Verify}(\cdot)$ verifies the correctness of the last non-memory-dependent epoch number, executes $\mathsf{VRFBatchVerify}(\cdot)$ to verify all VRF outputs, and executes $\mathsf{calcShard}(\cdot)$ over these VRF outputs to verify its output against k. Previous verification results can be cached and reused: 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 outputs and $\mathsf{calcShard}(\cdot)$.

Construction without allocation-privacy. As mentioned in §II-D, *allocation-privacy* is not always a desired property. To remove *allocation-privacy* from $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$, one can replace $\mathsf{VRFEval}(sk_i, st_t)$ 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.

C. Theoretical analysis

Correctness. Appendix B provides the full security proofs, and we summarise them below. WORMHOLE satisfies *liveness* as a node can compute $\mathsf{Join}(\cdot)$ and $\mathsf{Update}(\cdot)$ locally. WORMHOLE satisfies *unbiasibility*, as $\mathsf{VRFEval}(\cdot)$ and $\mathsf{calcShard}(\cdot)$ are deterministic functions, and system states are *unbiasible*, guaranteed by RB. WORMHOLE satisfies *join-randomness*, as VRF produces uniformly distributed outputs. When the epoch is a memory-dependent epoch, 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 the epoch is a non-memory-dependent epoch, the node will be shuffled, leading to $\gamma = \frac{1}{m}$. Thus, WORMHOLE satisfies *allocation-randomness*. WORMHOLE satisfies *allocation-privacy*, as one cannot compute $\mathsf{Join}(\cdot)$ or $\mathsf{Update}(\cdot)$ 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 $\mathsf{Join}(\cdot)$ and $\mathsf{Update}(\cdot)$ of $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ are $O(n)$ where n is the number of nodes, as each node needs to receive a constant number of system states for executing $\mathsf{Join}(\cdot)$ and $\mathsf{Update}(\cdot)$. A $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ proof contains $[3, 2w+2)$ VRF outputs/proofs, where w is the era length. $\mathsf{Join}(\cdot)$ invokes $\mathsf{VRFEval}(\cdot)$ for $[1, 2w)$ times, leading to computational complexity $O(w)$. $\mathsf{Update}(\cdot)$ invokes $\mathsf{VRFEval}(\cdot)$ for once, leading to computational complexity $O(1)$. $\mathsf{Verify}(\cdot)$ invokes $\mathsf{VRFBatchVerify}(\cdot)$ over $[1, 2w)$ VRF outputs/proofs, or $\mathsf{VRFVerify}(\cdot)$ for once if the verification results are cached, leading to computational complexity $O(w)$ or $O(1)$, respectively.

As analysed in the proof of Lemma 6, $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$'s *operability* $\gamma = 1 - \frac{m-1}{m} \cdot \frac{1}{2^{op}} = 1 - \frac{m-1}{m \cdot 2^{op}}$. By Definition 1, $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$'s *self-balance* $\mu = 1 - \left| \frac{(\gamma m - 1)\beta_t}{m-1} \right| = 1 - \beta_t + \frac{\beta_t}{2^{op}}$.

D. Comparison with existing shard allocation protocols

Table I summarises the comparison result. It shows that WORMHOLE is the only shard allocation protocol that is fully correct and achieves satisfactory performance, without relying on strong assumptions. To make a fair comparison, we also evaluate shard allocation protocols while assuming RB, and the evaluation results are summarised in Table II. Chainspace, Monoxide and Ethereum 2.0 are omitted as their shard allocation protocols do not rely on randomness.

According to Table II, these shard allocation protocols improve much in terms of the system model and communication complexity. However, they 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 *allocation-privacy*. In addition, Omniledger should still assume an identity authority for approving nodes to join the system. Moreover, all of them still suffer from weak *operability* except for Omniledger.

VI. INTEGRATION OF WORMHOLE

Given the different architectures, sharded blockchains differ in when and where a shard membership is stored and verified, and thus differ in integrating WORMHOLE. In this section, we analyse how to integrate WORMHOLE into different types of sharded blockchains.

A. Design choices related to WORMHOLE

The overhead introduced by WORMHOLE can be affected by two design choices of the sharded blockchain, namely the existence of identity registry and the choice of consensus protocol.

Existence of identity registry. Some sharded blockchains employ an identity registry that tracks identities of nodes in the system. For example, Elastico, RapidChain and Zilliqa allow a special shard to be the identity registry; Omniledger relies on a trusted identity authority; and Chainspace allows nodes to maintain a special smart contract managing identities.

The existence of identity registry decides where a shard membership is verified and stored. If the sharded blockchain employs an identity registry, then the identity registry can maintain and verify all shard memberships and proofs, and

Algorithm 2: Full construction of WORMHOLE $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$.

```
Algorithm calcShard ($m, op, h_x, h_{x+1}, \ldots, h_y$):
    idx $\leftarrow x$
    for $j \in [x+1, y]$ do
        if MSB$(op, h_j)$ = LSB$(op, h_{\text{idx}})$ then
            idx $\leftarrow j$  // Can be cached
    $shard\_id \leftarrow (h_{\text{idx}} \mod m) + 1$
    return $shard\_id$

Algorithm calcNMDEpoch ($w, sk_i, st_t$):
    // Get last non-memory-dependent (nmd) epoch
    $t_{\text{prev\_era}} \leftarrow t - w - (t \mod w)$
    $g^-_{i,t}, \pi^-_{i,t} \leftarrow$ VRFEval$(sk_i, st_{\text{prev\_era}})$
    $t^-_{\text{nmd}} \leftarrow t_{\text{prev\_era}} + (g^-_{i,t} \mod w)$
    $t_{\text{era}} \leftarrow t - (t \mod w)$
    $g_{i,t}, \pi_{i,t} \leftarrow$ VRFEval$(sk_i, st_{\text{era}})$
    $t_{\text{nmd}} \leftarrow t_{\text{era}} + (g_{i,t} \mod w)$
    last $\leftarrow t^-_{\text{nmd}} < t < t_{\text{nmd}}\ ?\ t^-_{\text{nmd}} : t_{\text{nmd}}$
    return $(last, (g^-_{i,t}, \pi^-_{i,t}, g_{i,t}, \pi_{i,t}))$

Algorithm Setup ($\lambda$):
    $m, op, w \leftarrow \lambda$
    return $(m, op, w)$

Algorithm Join ($pp, sk_i, st_t$):
    $m, op, w \leftarrow pp$
    $(\text{last}, \pi_{\text{range}}) \leftarrow$ calcNMDEpoch$(w, sk_i, st_t)$
    for $j \in [last, t]$ do
        $h_j, \pi_j \leftarrow$ VRFEval$(sk_i, st_j)$
    $k \leftarrow$ calcShard$(m, op, h_{\text{last}}, \ldots, h_t)$
    $\pi_{i,st_t,k} \leftarrow (\text{last}, \pi_{\text{range}}, h_{\text{last}}, \ldots, h_t, \pi_{\text{last}}, \ldots, \pi_t)$
    Store $\pi_{i,st_t,k}$ in memory
    return $k, \pi_{i,st_t,k}$

Algorithm Update ($pp, sk_i, st_t, k, \pi_{i,st_t,k}, st_{t+1}$):
    $m, op, w \leftarrow pp$
    $(\text{last}, \pi_{\text{range}}, h_{\text{last}}, \ldots, h_t, \pi_{\text{last}}, \ldots, \pi_t) \leftarrow \pi_{i,st_t,k}$
    $(\text{last}^+, \pi^+_{\text{range}}) \leftarrow$ calcNMDEpoch$(w, sk_i, st_{t+1})$
    Remove $(h_j, \pi_j)$ from memory for $j \in [\text{last}, \text{last}^+)$
    $h_{t+1}, \pi_{t+1} \leftarrow$ VRFEval$(sk_i, st_{t+1})$
    $k' \leftarrow$ calcShard$(m, op, h_{\text{last}^+}, \ldots, h_{t+1})$
    $\pi_{i,st_{t+1},k'} \leftarrow (\text{last}^+, \pi^+_{\text{range}}, h_{\text{last}^+}, \ldots, h_{t+1}, \pi_{\text{last}^+}, \ldots, \pi_{t+1})$
    Store $\pi_{i,st_{t+1},k'}$ in memory
    return $k', \pi_{i,st_{t+1},k'}$

Algorithm Verify ($pp, pk_i, st_t, k, \pi_{i,st_t,k}$):
    $m, op, w \leftarrow pp$
    $(\text{last}, \pi_{\text{range}}, h_{\text{last}}, \ldots, h_t, \pi_{\text{last}}, \ldots, \pi_t) \leftarrow \pi_{i,st_t,k}$
    $(g^-_{i,t}, \pi^-_{i,t}, g_{i,t}, \pi_{i,t}) \leftarrow \pi_{\text{range}}$
    $t_{\text{prev\_era}}, t_{\text{era}} \leftarrow t - w - (t \mod w), t - (t \mod w)$
    $t^-_{\text{nmd}}, t_{\text{nmd}} \leftarrow t_{\text{prev\_era}} + (g^-_{i,t} \mod w), t_{\text{era}} + (g_{i,t} \mod w)$
    // Verify memory range
    if $t^-_{nmd} < t < t_{nmd} \wedge last \neq t^-_{nmd}$  $\vee$
       $t_{nmd} < t \wedge last \neq t_{nmd}$  $\vee$
       VRFVerify$(pk_i, st_{prev\_era}, g^-_{i,t}, \pi^-_{i,t}) = 0$  $\vee$
       VRFVerify$(pk_i, st_{era}, g_{i,t}, \pi_{i,t}) = 0$  then
        return 0
    $\vec{st}, \vec{h}, \vec{\pi} \leftarrow (st_{\text{last}}, \ldots, st_t), (h_{\text{last}}, \ldots, h_t), (\pi_{\text{last}}, \ldots, \pi_t)$
    if VRFBatchVerify$(pk_i, \vec{st}, \vec{h}, \vec{\pi}) = 0$ then
        return 0  // Can be cached
    if $k \neq$ calcShard$(m, op, h_{last}, \ldots, h_t)$ then
        return 0
    return 1
```

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

		System model			Correctness					Performance metrics			
	State update	Network model	Trusted components	Fault tolerance	Public verifiability	Liveness	Allocation-rand.	Unbiasibility	Privacy^o	Join comm. compl.	Update comm. compl.	Self-balance	Operability
Elastico	New block	Async.★	Rand. Beacon*	1★	✓	✓	✓	✗	✓	$O(n)$★	$O(n)$★	1	$\frac{1}{m}$
Omniledger	Identity authority	Async.★	Identity auth. Rand. Beacon*	1★	✓	✓★	✓	✓	✗	$O(n)$	$O(n)$★	$1 - \frac{(2m-3)\beta_t}{3m-3}$	$\frac{2}{3}$
RapidChain	Nodes joining	Async.★	Rand. Beacon*	1★	✗	✓	✓	✓	✗	$O(n)$★	$O(n)$★	$1 - \beta_t$	$max(1 - \kappa\alpha_t n, 0)$
Zilliqa	New block	Async.★	Rand. Beacon*	1	✓	✓	✓	✗	✓	$O(n)$	$O(n)$	1	$\frac{1}{m}$
WORMHOLE (Our proposal in §V)	New rand.	Async.	Rand. Beacon*	1	✓	✓	✓	✓	✓	$O(n)$	$O(n)$	$1 - \beta_t + \frac{\beta_t}{2^{op}}$	$1 - \frac{m-1}{m \cdot 2^{op}}$

^{o} Optional. * Shard allocation protocols can rely on an external randomness beacon, or allow a group of nodes to run a decentralised randomness beacon protocol.

a node can query other nodes' shard memberships over the identity registry rather than verifying it. Otherwise, a node has to receive and verify other nodes' shard memberships when executing other subprotocols (e.g., consensus).

Choice of consensus protocol. Existing research [60], [61] suggests to classify consensus protocols into two types, namely BFT-style consensus and Nakamoto-style consensus. In BFT-style consensus, given the latest blockchain, nodes propose blocks, vote to agree on a unique block, and append the agreed block to the blockchain. In Nakamoto-style consensus, given the latest blockchain, nodes compete to solve a cryptographic puzzle. If a node solves a puzzle, then it can append a new block associated to the puzzle solution to the blockchain. Nodes follow a chain selection rule to decide the main chain among all forks, and eventually the main chains of different nodes converge to the same one.

The choice of consensus protocol decides when a shard membership is queried or verified. If the sharded blockchain employs BFT-style consensus, then for each block, a node has to additionally verify a quorum of nodes' shard memberships. If the sharded blockchain employs Nakamoto-style consensus, then for each block, a node has to additionally verify the block proposer's shard membership.

Classification of sharded blockchains. Among our evaluated sharded blockchains, Elastico, Omniledger, Chainspace, Zilliqa and Ethereum 2.0 employ an identity registry and

BFT-style consensus, while Monoxide employs Nakamoto-style consensus without an identity registry. There exists no sharded blockchains that employ BFT-style consensus without an identity registry, or employ an identity registry and Nakamoto-style consensus simultaneously.

B. Integration analysis

We then analyse how to integrate WORMHOLE into the above two cases, namely Nakamoto-style sharded blockchains without identity registry and BFT-style sharded blockchains with identity registry.

With identity registry, BFT-style consensus. In this case, every node executes WORMHOLE to obtain a shard membership with proof and submits them to the identity registry for verification. For each new epoch, a node needs to compute a VRF output with proof and send them to the identity registry. The identity registry needs to send each node the set of all peers' identities and the shard size. Every node then executes BFT-style consensus with peers to agree on blocks. For each vote, a node looks up the the voter node's identity within the set. A block needs to obtain a quorum of votes to be valid. The identity set can be replaced with a cryptographic accumulator (e.g., bloom filter), where the size and the lookup complexity can be independent with the set size. The identity registry also manages nodes' identities and handles Sybil attacks.

Thus, the identity registry needs to additionally receive, store and verify shard memberships and proofs for all nodes. For each new epoch, each node needs to additionally submit a VRF output and proof to the identity registry and receive the set of identities and an integer, while the identity registry needs to verify a VRF output and update the shard membership. For each block, each node needs to look up a quorum of nodes' shard memberships within the set.

No identity registry, Nakamoto-style consensus. In this case, every node executes WORMHOLE to obtain a shard membership with proof, and keeps solving puzzles to propose blocks over the main chain decided by the chain selection rule. Each block additionally attaches the miner's shard membership and proof. Upon receiving a block, the node additionally verifies the miner's shard membership. Similar to Elastico, Chainspace, Zilliqa and Ethereum 2.0, a node has to solve a cryptographic puzzle in order to obtain an identity in the system. To support permissionless settings, the puzzle's difficulty is controlled by a difficulty adjustment mechanism.

Thus, for every block, a node needs to additionally receive, store and verify a shard membership and proof.

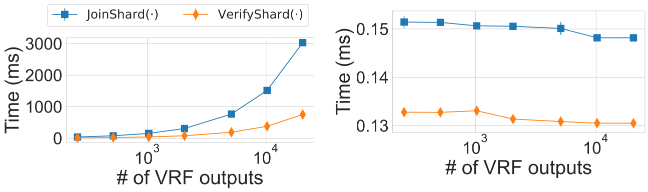
VII. EXPERIMENTAL EVALUATION OF WORMHOLE

In this section, we implement WORMHOLE (§VII-A) and evaluate its overhead introduced in sharded blockchains (§VII-B) and performance metrics achieved in the wild (§VII-C). The evaluation results show that WORMHOLE introduces little overhead and achieves performance metrics consistent with the theoretical values.

A. Implementation and experimental setup

We implement WORMHOLE in Rust. We use `rug` [62] for large integer arithmetic and `bitvec` [63] for bit-level operations. We use `w3f/schnorrkel` [64], which implements the standardised VRF [42] over the Curve25519 elliptic curve with Ristretto [65] compressed points. The VRF batch verification is based on the Schnorr-style aggregatable discrete log equivalence proofs (DLEQs) [58]. The size of keys, VRF outputs and proofs are 32, 32 and 96 Bytes, respectively. System states are simulated by `rand` [66]. We write the benchmarks using `cargo-bench` [67] and `criterion` [68]. We specify the O3-level optimisation for compilation, and sample 20 executions for each unique group of parameters. All experiments were conducted on a MacBook Pro with a 2.2 GHz 6-Core Intel i7 processor and a 16 GB RAM.

B. Overhead analysis



(a) Newly joined nodes.

(b) Existing nodes.

Fig. 5: Evaluation of WORMHOLE.

We benchmark Join(·), Update(·) and Verify(·) for WORMHOLE. Recall that with era length w, a node reaches a non-memory-dependent epoch for every w epochs on average. We choose w ranging from 256 to 10192 epochs. In Bitcoin's setting where a block is generated for every ten minutes, 256 and 10192 epochs take about 4 and 142 days, respectively.

Figure 5 shows the results. For newly joined nodes, the execution time of Join(·) and Verify(·) increases linearly with the number of random outputs. With 256 random outputs, Join(·) and Verify(·) take 39 and 12 ms, respectively. With 20384 random outputs, Join(·) and Verify(·) take 3000 and 800 ms, respectively. For existing nodes, Update(·) and Verify(·) take about 0.15 and 0.13 ms, respectively. A shard membership takes at most 4 Bytes, which can support 2^{4*8} shards. When VRF outputs and proofs are 32 and 96 Bytes, a membership proof takes $(32 + 96) * 2w = 256w$. The proof size S_π is 64 and 2548 KB with $w = 256$ and 10192, respectively; and updating a proof takes 128 Bytes.

We analyse the concrete overhead that a node takes for executing WORMHOLE for two cases in §VI-B separately.

With identity registry, BFT-style consensus. In this case, the identity registry needs to receive, store and verify shard memberships and proofs. This incurs one-time overhead of $S_\pi * n$ KB on storage and communication, and n non-cached Verify(·) invocations. For each epoch, each node sends a VRF output and proof to the identity registry, the identity registry verifies it, and sends back the set of identities and the shard size. This incurs the overhead of $32 * \frac{n}{m} + 128$ Bytes on communication, and a cached Verify$(\cdot)$ invocations for each node. If replacing the set with a constant-size accumulator of s Bytes, then the per-node communication overhead can be reduced to $s + 128$ Bytes. For each block, each node has to look up a quorum of nodes' identities, which introduces computation overhead of $\frac{n}{m}$ lookup operations.

No identity registry, Nakamoto-style consensus. In this case, for each block, a node needs to additionally receive, store and verify a shard membership and proof. This incurs the overhead of S_π KB for each node, and a non-cached Verify$(\cdot)$ invocation.

C. Simulation

While we define self-balance and operability metrics for shard allocation in §II-E, we simulate WORMHOLE in a network with 128 shards and 32768 nodes, and perform experimental analysis to confirm the theoretical results on WORMHOLE's self-balance and operability guarantees in §V-C.

Observed load balance. Following existing distributed systems research [69], the observed load balance is quantified as the *coefficient of variation (CV)*, namely the ratio between the standard deviation $std(\cdot)$ and the mean value $mean(\cdot)$ of the node distribution across shards. Specifically, the observed load balance in epoch t is $\frac{std(\mathcal{N}^t)}{mean(\mathcal{N}^t)}$, where $\mathcal{N}^t = \{n_k^t\}_{k\in[m]}$ is the number n_k^t of nodes in every shard k in epoch t. When CV is zero, then the system achieves optimal load balance, where every shard contains the same number of nodes. When CV is smaller than 1, then it means the distribution is low-variance and the system achieves satisfactory load balance.

Observed operability. The observed operability is quantified as the ratio between the number of moved nodes and the number of existing nodes. Specifically, the observed operability in epoch t is $1 - \frac{n_{\text{moved}}^t}{n^t}$, where n^t and n_{moved}^t are the total number of nodes and the number of moved nodes in epoch t, respectively.

Simulation setup. We simulate WORMHOLE with $m = 128$ shards, $w = 10192$, and operability parameter $\gamma = 0.95$. There are $n = 128*256 = 32768$ nodes at the beginning. The system executes for 500 epochs with variant churn rate. Figure 6(a) outlines the simulated maximum join and leave rate (α, β) over 500 epochs. As our analysis in §V-C shows that leave churn rate incurs more impact on the performance metrics, we test more volatile β (ranging from 0.03 to 0.15) compared to α (ranging from 0.03 to 0.06). The churn rate (α_t, β_t) in each epoch t are uniformly random over interval $[0, \alpha]$ and $[0, \beta]$, respectively.

Simulation results (Figure 6). While Figure 6 (b) summarises the execution results, Figure 6 (c) and (d) outline the observed self-balance and operability over the 500 epochs, respectively.

Figure 6(c) shows the observed load balance. We simulate both best-case and worst-case execution. In the best-case execution, a random set of nodes leave the network, and each shard is likely to lose a similar number of nodes. In the worst-case execution, nodes in the same shard leave the network, making the shards less balanced. We observe that when $\beta = 0.03$, the observed load balance is about 0.07 and 0.3 in the best-case and worst-case execution, respectively. When β gradually increases to 0.15, the observed load balance increases to 0.3 and 1.1 in the best-case and worst-case execution, respectively. The observed load balance is less than 1 in most cases, meaning that WORMHOLE achieves satisfactory load balance guarantee under high leave rate. In addition, in the worst-case execution, when β recovers from 0.15 to 0.01, the observed load balance reduces from 1.1 to 0.5 monotonically within about 25 epochs. This shows that WORMHOLE can recover from temporary load imbalance in a short time period.

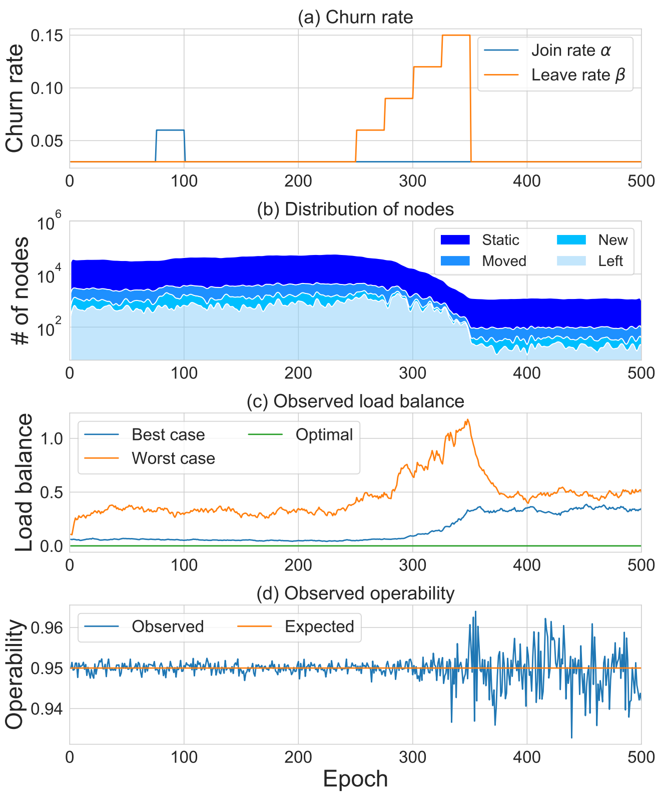


Fig. 6: Simulation results of executing WORMHOLE for 500 epochs in different churn conditions. **(a)** Simulated maximum churn rate (α, β) over epochs. **(b)** The distribution of nodes over epochs. A node is static if it stays in the same shard compared to the last epoch; is moved if it moves to another shard compared to the last epoch; is new if it newly join the system in this epoch; and is left if it leaves the system in this epoch. **(c)** The observed load balance in the best-case and worst-case execution. In the best case, a random set of nodes leave the system, while in the worst case nodes in the same shard leave the system. **(d)** The observed operability, compared with the expected one.

Figure 6(d) shows the observed operability. We observe that while the expected operability is 0.95, the observed operability is 0.95 ± 0.015, meaning that WORMHOLE can achieve the parametrised operability with little bias. In addition, when β recovers from 0.15 to 0.01, the maximum bias remains stable.

This is because the number of nodes has been reduced, making the statistical results more volatile.

VIII. 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 filled this gap by formally defining the permissionless shard allocation protocol, evaluating existing shard allocation protocols, observing trade-offs, and constructing a new shard allocation protocol Wormhole. Theoretical analysis and experimental evaluation show that Wormhole is secure and efficient.

References

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

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

[3] G. Danezis and S. Meiklejohn, "Centrally banked cryptocurrencies," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016*, 2016.

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

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

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

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

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

[9] "The zilliqa design story piece by piece: Part 1 (network sharding)," 2020, https://blog.zilliqa.com/https-blog-zilliqa-com-the-zilliqa-design-story-piece-by-piece-part1-d9cb32ea1e65.

[10] "Ethereum sharding: Overview and finality," 2020, https://medium.com/@icebearhww/ethereum-sharding-and-finality-65248951f649.

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

[12] V. Buterin. (2020). Serenity design rationale. https://notes.ethereum.org/@vbuterin/rkhCgQteN?type=view.

[13] (2020). On sharding blockchains faqs. https://eth.wiki/sharding/Sharding-FAQs.

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

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

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

[17] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018*, 2018.

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

[19] Z. Team *et al.*, "The ZILLIQA Technical Whitepaper," *Retrieved September*, 2017.

[20] (2020). Ethereum/eth2.0-specs. https://github.com/ethereum/eth2.0-specs.

[21] G. Wang, Z. J. Shi, M. Nixon, and S. Han, "Sok: Sharding on blockchain," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019*, 2019.

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

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

[24] (2020). Ethereum/wiki. https://eth.wiki/sharding/.

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

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

[27] J. Dinger and H. 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)*, IEEE, 2006, 8–pp.

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

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

[30] (2020). Warp sync - wiki parity tech documentation. https://wiki.parity.io/Warp-Sync.

[31] J. R. Douceur, "The sybil attack," in *International workshop on peer-to-peer systems*, Springer, 2002.

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

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

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

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

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

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

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

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

[40] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *40th Annual Symposium on Foundations of Computer Science*, IEEE, 1999.

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

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

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

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

[45] (2020). Random uchile - random uchile. https://beacon.clcert.cl/en/.

[46] (2020). Brazilian beacon. https://beacon.inmetro.gov.br/.

[47] (2020). Distributed randomness beacon — cloudflare. https://www.cloudflare.com/leagueofentropy/.

[48] (2020). Unicorn beacon by lacal. http://trx.epfl.ch/beacon/index.php.

[49] (2020). Drand - distributed randomness beacon. https://drand.love/.

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

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

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

[53] R. Han, J. Yu, and H. Lin, "Randchain: Decentralised randomness beacon from sequential proof-of-work," *IACR Cryptol. ePrint Arch.*, 2020.

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

[55] J. Clark and U. Hengartner, "On the use of financial data as a random beacon.," *EVT/WOTE*, vol. 89, 2010.

[56] J. Benet and N. Greco, "Filecoin: A decentralized storage network," *Protoc. Labs*, pp. 1–36, 2018.

[57] S. Hohenberger and B. Waters, "Constructing verifiable random functions with large input spaces," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2010, pp. 656–672.

[58] A. Davidson, I. Goldberg, N. Sullivan, G. Tankersley, and F. Valsorda, "Privacy pass: Bypassing internet challenges anonymously," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 3, pp. 164–180, 2018.

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

[60] J. Neu, E. N. Tas, and D. Tse, "Ebb-and-flow protocols: A resolution of the availability-finality dilemma," *arXiv preprint arXiv:2009.04987*, 2020.

[61] S. Sankagiri, X. Wang, S. Kannan, and P. Viswanath, "Blockchain cap theorem allows user-dependent adaptivity and finality," *arXiv preprint arXiv:2010.13711*, 2020.

[62] "Crates/rug," 2020, https://crates.io/crates/rug.

[63] "Crates/bitvec," 2020, https://crates.io/crates/bitvec.

[64] "Schnorr vrfs and signatures on the ristretto group," 2020, https://github.com/w3f/schnorrkel.

[65] "The ristretto group," 2020, https://ristretto.group/.

[66] "Crates/rand," 2020, https://crates.io/crates/rand.

[67] "Cargo-bench," 2020, https://doc.rust-lang.org/cargo/commands/cargo-bench.html.

[68] "Criterion.rs," 2020, https://github.com/bheisler/criterion.rs.

[69] V. Ramasubramanian and E. G. Sirer, "The design and implementation of a next generation name service for the internet," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 331–342, 2004.

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

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

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

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

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

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

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

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

[78] M. Jelasity and A.-M. Kermarrec, "Ordered slicing of very large-scale overlay networks," in *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*, IEEE, 2006, pp. 117–124.

[79] A. Fernández, V. Gramoli, E. Jiménez, A.-M. Kermarrec, and M. Raynal, "Distributed slicing in dynamic systems," in *27th International Conference on Distributed Computing Systems (ICDCS'07)*, IEEE, 2007, pp. 66–66.

[80] V. Gramoli, Y. Vigfusson, K. Birman, A.-M. Kermarrec, and R. van Renesse, "A fast distributed slicing algorithm," in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, 2008, pp. 427–427.

[81] F. Maia, M. Matos, R. Oliveira, and E. Riviere, "Slicing as a distributed systems primitive," in *2013 Sixth Latin-American Symposium on Dependable Computing*, IEEE, 2013, pp. 124–133.

[82] D. J. DeWitt, J. F. Naughton, and D. F. Schneider, "Parallel sorting on a shared-nothing architecture using probabilistic splitting," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1991.

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

[84] B. Schoenmakers, "A simple publicly verifiable secret sharing scheme and its application to electronic voting," in *Annual International Cryptology Conference*, Springer, 1999, pp. 148–164.

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

[86] (2020). Zilliqa - the next generation, high throughput blockchain platform. https://zilliqa.com/.

[87] (2020). Zilliqa developer portal · technical and api documentation for participating in the zilliqa network. https://dev.zilliqa.com/.

[88] (2020). Zilliqa/zilliqa at v5.0.1. https://github.com/Zilliqa/Zilliqa/tree/v5.0.1.

[89] (2020). Randao: A dao working as rng of ethereum. https://github.com/randao/randao.

Appendix

A. Proofs of impossibility and memory-dependency

Lemma 2. *If a correct shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ with m shards satisfies update-randomness with γ, the self-balance μ of $\Pi_{\mathsf{ShardAlloc}}$ is*

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

where β_t is the percentage of nodes leaving the network in epoch t.

Proof. By Definition 10, in epoch t, the number n_k^t of nodes in any shard k is $\frac{n^t}{m}$. By *join-randomness*, newly joined nodes will be uniformly allocated into shards. Thus, without the loss of generality, we assume at the end of epoch t, no node joins the network ($\alpha = 0$) and $\beta_t n^t$ nodes leave the network. Let Δn_k^t be the number of leaving nodes in shard $k \in [m]$ in epoch 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 $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$, and its resulting shard complies with the probability distribution in Definition 4. After executing $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$, 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 in epoch $t+1$ is

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

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

$$\begin{aligned} \frac{|n_i^{t+1} - n_j^{t+1}|}{n^t} &= \frac{|\frac{\gamma m-1}{m-1}(n_i^t - \Delta n_i^t) - \frac{\gamma m-1}{m-1}(n_j^t - \Delta n_j^t)|}{n^t} \\ &= \frac{|\frac{\gamma m-1}{m-1}(\Delta n_i^t - \Delta n_j^t)|}{n^t} \end{aligned}$$

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 μ can be calculated. 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 $\beta_t n^t$. Therefore, μ can be calculated as

$$\begin{aligned} \mu &= 1 - \max_{\forall i,j \in [m]} \frac{|n_i^{t+1} - n_j^{t+1}|}{n^t} \\ &= 1 - \max_{\forall i,j \in [m]} \frac{|\frac{\gamma m-1}{m-1}(\Delta n_i^t - \Delta n_j^t)|}{n^t} \\ &= 1 - \frac{|\frac{\gamma m-1}{m-1}\beta_t n^t|}{n^t} = 1 - \left|\frac{(\gamma m - 1)\beta_t}{m-1}\right| \end{aligned}$$

□

Theorem 3. *Let β_t be the percentage of nodes leaving the network in epoch t. It is impossible for a correct shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ 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* $\mu = 1$ and *operability* $\gamma = 1$. According to Lemma 1, $\mu = 1$ only when either $\beta_t = 0$ or $\gamma m = 1$. As $\gamma = 1$ and $m > 1$, $\gamma m > 1$. Thus, $\Pi_{\mathsf{ShardAlloc}}$ can achieve $\mu = 1$ and $\gamma = 1$ simultaneously only when $\beta_t = 0$. However, $\beta_t > 0$, which leads to a contradiction. □

Theorem 4. *If a correct shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ is μ-self-balanced and γ-operable where $\mu \in (0, 1-\beta_t)$ and $\gamma \in (\frac{1}{m}, 1)$, then $\Pi_{\mathsf{ShardAlloc}}$ is memory-dependent.*

Proof. We prove this by contradiction. Assuming $\Pi_{\mathsf{ShardAlloc}}$ is memoryless, i.e., the output of $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(pp.sk_i, st_t, k, \pi_{i,st_t,k}, st_{t+1})$ 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 $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ is *non-uniform*, given the *update-randomness* property. In this case, executing $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ 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 $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ – should enable verifiers to verify node i is at shard k in epoch t.

Verifying node i is at shard k in epoch t is achieved by verifying $\pi_{i,st_t,k}$. Thus, $\pi_{i,st_{t+1},k'}$ 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, $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ does not rely on any prior system state. When $\gamma = \frac{1}{m}$, the resulting shard of $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ is uniformly distributed, so $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ can just assign nodes randomly according to the incoming system state. When $\gamma = 1$, the resulting shard of $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ 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.*

B. Proofs of WORMHOLE $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$

Lemma 3. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies liveness.*

Proof. By RB-Availability, the RB is always producing random outputs, and therefore new system states regularly. Given a new system state, any honest node can execute $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ (or $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ for newly joined nodes). As both $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ and $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ can be computed locally without interacting with other nodes, the execution of them will eventually terminate. □

Lemma 4. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies unbiasibility.*

Proof. We prove this by contradiction. Assuming that $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not satisfy *unbiasibility*: given a system state, an adversary can manipulate the probability distribution of the output shard of $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ or $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ with non-negligible probability. This consists of three attack vectors: 1) the adversary can manipulate the system state; 2) when $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ or $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ 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 RB is unbiasible, so the system state of $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ is unbiasible. By VRF-Uniqueness, given a secret key, the VRF output of the system state is unique, which eliminates the last two attack vectors. In addition, the uniqueness of the VRF output of the unbiasible system state indicates that the VRF output is unbiasible. The output shard of $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ or $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ is a modulus of the VRF output, which is also unbiasible. This eliminates the first attack vector. Thus, if $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not resist against the first attack vector, then this contradicts RB-Unbiasibility; and if $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not resist against the second and/or the last attack vectors, then this contradicts VRF-Uniqueness. □

Lemma 5. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies join-randomness.*

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

Lemma 6. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies update-randomness.*

Proof. We prove this by contradiction. Assuming that $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ 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 epoch t is a non-memory-dependent epoch, the node will be shuffled. By VRF-Pseudorandomness, the probability of moving to each shard is same. Thus, there is a $\gamma = \frac{1}{m}$ that makes the output shard of $\mathsf{Update}(\cdot)$ to comply with the distribution in Definition 4.

When t is a memory-dependent epoch, the last VRF output remains unchanged. In $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$, given the last VRF output, the probability that the *op* MSBs of the new VRF output equal to *op* LSBs of the last VRF output is $\frac{1}{2^{op}}$. By VRF-Pseudorandomness, the probability of moving to each other shard is same. Thus, there is a $\gamma = 1 - \frac{1}{2^{op}} \cdot \frac{m-1}{m} = 1 - \frac{m-1}{m \cdot 2^{op}}$ that makes the output shard of $\mathsf{Update}(\cdot)$ to comply with the distribution in Definition 4.

Thus, if $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not satisfy *update-randomness*, then this contradicts VRF-Pseudorandomness. □

Lemma 7. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies allocation-privacy.*

Proof. This follows proofs of Lemma 5 and 6. □

C. 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 tolerant (CFT) systems to raise their throughput. Allocating nodes to shards in a CFT system is straightforward, as there is no Byzantine adversaries in the system, and the total number of nodes is fixed and known to everyone [1], [4], [70]. The main challenge is to balance the computation, communication, and storage workload among shards. Despite a large number of load-balancing algorithms [5]–[8], [71], 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 [72], [73] employ Distributed Hash Tables (DHT) [74] 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 [75]–[77], therefore are not suitable for blockchains, which demands strong consistency on financial data.

Distributed Slicing. Distributed Slicing [78] 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 [79]–[82] 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. [21] propose an evaluation framework based on Elastico's architecture; Avarikioti et al. [22] formalise sharded blockchains by extending the model of Garay et al. [83]. 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.

D. Details of evaluated shard allocation protocols

1) **Elastico:** In Elastico, a new block will trigger the shard allocation protocol. In Elastico's shard allocation protocol, all nodes in a special shard called *final committee* execute a commit-then-reveal Distributed Randomness Generation (DRG) protocol [34] to produce a random output. With a the random output as input, a node derives a PoW puzzle, and needs to solve it to obtain a valid shard membership. The prefix of a valid PoW solution is ID of the allocated shard.

The DRG protocol [34] 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. Nodes receiving $\geq \frac{2}{3} n_s$ hashes will execute a vector consensus [39] to agree on a set of hashes. The vector consensus works under synchronous networks, and has the *communication complexity* of $O({n_s}^{f_s})$. After the consensus, each node broadcasts its original random string to other nodes. Each node can XOR arbitrary $\frac{1}{2} n_s{+}1$ received strings to obtain a valid random output.

2) **Omniledger:** Similar to Elastico, Omniledger's shard allocation protocol is also constructed from DRG. In Omniledger, all nodes in the network jointly execute *RandHound* [35] - a leader-based DRG protocol tolerating $\frac{1}{3}$ faulty nodes - to generate random outputs. RandHound adapts Publicly Verifiable Secret Sharing (PVSS) [84] to make random outputs publicly verifiable, and CoSi [85] to improve the communication and space complexity of generating multi-signatures. RandHound has the *communication complexity* of $O(n)$, works under asynchronous network, and tolerates $\frac{1}{3}$ faulty nodes [35].

As RandHound is leader-based, nodes should elect a leader before running RandHound. To elect a leader, nodes run a verifiable random function (VRF) [40]-based cryptographic sortition, which works as follows. Each node first obtains the whole list of peers from the identity authority. Then, each node computes a ticket by running $\mathsf{VRFEval}(sk_i, "leader"||peers||v)$, where sk_i is its secret key, $peers$ is the list of peers, and v is a view counter starting from zero. Each node then broadcasts its ticket, and waits for a timeout Δ. After Δ, each node takes the one with smallest ticket as the leader. If the leader does not start executing 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. If the sortition fails for five times, nodes quit the leader election and RandHound, and instead execute an asynchronous coin-tossing protocol [36] to generate the random output. The coin-tossing protocol [36] guarantees safety under asynchronous networks, but suffers from the communication complexity of $O(n^3)$.

After generating a random output, for each shard, the centralised identity authority will randomly select $\leq \frac{n}{m}$ pending nodes and allow them to join the shard. In addition, $\frac{1}{3}$ existing nodes will be shuffled across shards. Given a new random output and the list of nodes queried from the identity authority, each node can permute an order of nodes. Compared to the previous epoch's permutation, $\frac{1}{3}$ nodes will be in different slots and moved to random shards.

3) **RapidChain:** In RapidChain, a node has to solve a PoW puzzle to obtain a shard membership. The PoW puzzle is derived from the random output produced by a DRG protocol. The DRG prevents the long range attack, where one pre-computes PoW solutions in order to take advantage of consensus in the future. The DRG is executed by nodes in a special shard called *reference committee*. The DRG protocol works as follows. First, each node chooses a random string and shares it to others using Feldman Verifiable Secret Sharing (VSS) [37]. 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 network and cannot tolerate any faults, as any node failing to broadcast shares to all other nodes will make the protocol to restart, breaking the liveness.

After solving a PoW puzzle, a node applies the Commensal Cuckoo rule [38] to obtain a shard membership. 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 joins the network, it will "push forward" nodes in a constant-size interval surrounding itself to other shards.

As Feldman VSS assumes synchrony, RapidChain's shard allocation protocol should assume synchrony to remain correct. Commensal Cuckoo assumes crash faults only, as the shard membership is not publicly verifiable, and a Byzantine node can deviate from the protocol and stay in any shard. With only crash faults, the load across shards is balanced adaptively with new nodes joining.

4) **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.

5) **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.

6) **Zilliqa:** Zilliqa [86] is a permissionless sharded blockchain that claims to achieve the throughput of over 2,828 transactions per second. It follows the design of Elastico [14], but with several optimisations. Our evaluation is based on Zilliqa's whitepaper [19], Zilliqa's developer page [87], and Zilliqa's source code (the latest stable release v5.0.1) [88].

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.

7) **Ethereum 2.0:** In Ethereum 2.0 [20], each account has a unique ID, and accounts are assigned to different shards according to their IDs. More specifically, Ethereum 2.0 employs Proof-of-Stake (PoS)-based consensus, where the voting power is proportional to the cryptocurrency deposits a.k.a. staking power. Note that Ethereum 2.0 also employs a DRG protocol (i.e., RANDAO [89]), which is used for sampling block producers (aka *validators*) rather than shard allocation.