Möbius: an Atomic State Sharding Design for Account-Based Blockchains

Srisht Fateh Singh, Panagiotis Michalopoulos, Sidi Mohamed Beillahi, Andreas Veneris, Fan Long
University of Toronto
srishtfateh.singh@mail.utoronto.ca, p.michalopoulos@mail.utoronto.ca, sm.beillahi@utoronto.ca,
veneris@eecg.toronto.edu, fanl@cs.toronto.edu

***Abstract*—This paper presents Möbius, the first cost-efficient state sharding design that remains consensus mechanism agnostic and guarantees atomicity for cross-shard smart contract transactions. In particular, to address the challenges posed by the growing blockchain state, Möbius enables its participants to verify all transactions while only storing a partial state. Unlike previous state sharding systems, the proposed protocol uses a novel vector commitment data structure to reduce the network bandwidth overhead via proof aggregation. Further, it utilizes a novel epoch-based multi-phase commitment technique for guaranteeing atomicity in cross-shard transactions. Experiments presented here show that Möbius reduces the disk requirement of each participant linearly with respect to the number of shards. Further, it presents a 4.7–$7.3\times$ lower network bandwidth overhead when compared to existing state-of-the-art state sharding systems. The outcomes also confirm that existing smart contracts can operate on Möbius in cross-shard scenarios without modifications.**

***Index Terms*—blockchain, sharding, blockchain state sharding, smart contracts, cross-shard transactions, atomicity.**

I. Introduction

Following the success of cryptocurrencies [**?**], [**?**], blockchains have evolved into secure, decentralized, and consistent transaction ledgers at Internet scale. New blockchains like Ethereum [**?**] can encode customized transaction rules as smart contracts, which greatly extend the capability of blockchain ledgers beyond money transfers. These technologies now fuel innovation in real-world applications such as finance, supply chain, and entertainment, among others [**?**].

The recent adoption of this technology has presented a critical challenge in the ever growing size of the *blockchain state*. For example, the full blockchain state size of Ethereum currently exceeds 1TB [**?**]. Since the blockchain state is designed to be persistent, its size keeps growing as the system processes more transactions. Evidently, for systems that exhibit a higher throughput, the state size grows even faster [**?**], [**?**], [**?**]. Further, the recent popularity of Metaverse and non-fungible token applications will further exacerbate this situation as these applications tend to store significantly more data than conventional financial applications [**?**]. One undesirable consequence of a large state is increased storage requirements for each blockchain node resulting in a greater overall cost of maintaining the network. A cascading corollary of this effect is that higher hardware requirements may discourage the number of active nodes and this can cause the system to become more "centralized" and less robust than what was originally envisioned.

To address these issues, the idea of *state sharding* has been proposed to reduce the hardware requirements of storing the growing blockchain state. Under that scheme, each network node can store a part of the global state (*i.e.,* a shard) while still maintaining the capability of verifying all transactions. However, previous state sharding techniques suffer from two drawbacks that limit their adoption. First, to enable nodes to verify transactions from a different shard, each transaction has to attach cryptographic proofs of all the state values the transaction accesses. Since most blockchains use Merkle Patricia Trees (MPT) to store their states, the proof of each state value will have to contain data from up to $O(\log n)$ nodes in MPT, where n is the size of the whole blockchain state. This significantly increases the size of the transaction and introduces a notable network bandwidth overhead for transaction propagation.

The second drawback is that none of the existing techniques appropriately support cross-shard smart contract transactions. Specifically, a transaction is a *cross-shard* one if it accesses state values from multiple shards. The standard approach to handling cross-shard transactions is to relay and process such a transaction for each accessed shard one by one. It can guarantee eventual consistency but not *transaction atomicity*, *i.e.,* a cross-shard transaction should either succeed or abort on all shards, and the intermediate state during the transaction execution should not be visible to other transactions. Since transaction atomicity is essential to the correctness of many smart contracts, these techniques may therefore be impractical without rewriting most of the existing smart contracts.

Möbius: This paper presents Möbius, the first cost-efficient state sharding design that remains consensus mechanism agnostic, and guarantees atomicity for cross-shard smart contract transactions. To enable *efficient state sharding*, Möbius stores its state as a vector commitment tree (VCT) using *Hyperproofs*, a new cryptographic scheme that enables state proof aggregation [**?**]. It organizes consecutive blocks into epochs, where each epoch is referred to as a *round*. Instead of appending a state value proof to each transaction, each shard periodically attaches an aggregated proof for state values accessed in all transactions in the blocks of the last epoch. This mechanism amortizes the proof cost and significantly reduces the bandwidth overhead caused by state sharding.

To guarantee *transaction atomicity*, Möbius operates with a novel epoch-based multi-phase commitment. For a cross-shard transaction whose execution requires data from multiple shards, a node from the initiating shard will pack the transaction in a block as non-committed along with all accessed state values from the initiating shard. Then a node from the subsequent shard can pick up this transaction, and supply additional state values from its shard. This process continues until either all accessed values are available and the transaction commits simultaneously on all shards, or the lifetime of the non-committed transaction expires and the process aborts. This enables Möbius to support arbitrarily complicated smart contract transactions that require data from multiple state shards.

Experimental Results: We have implemented a prototype of Möbius and evaluated it with three representative transaction

979-8-3503-1019-1/23/$31.00 ©2023 IEEE

benchmark traces, *i.e.,* simple transfers, ERC20 transfers, and DeFi swap transactions. These results show 1) that Möbius effectively reduces the stored blockchain state size per node proportionally as we increase the number of shards, 2) that our solution has up to $7.3\times$ less network bandwidth overhead compared to previous techniques using MPT proofs, and 3) that Möbius guarantees atomicity for even complicated DeFi swap transactions that have more than 5 cross-shard hops.

Contributions: This paper makes the following contributions:

- **Möbius:** The first cost-efficient state sharding protocol that is consensus mechanism agnostic, and guarantees atomicity for cross-shard smart contract transactions.
- **VCT with Hyperproofs:** We introduce novel techniques to store blockchain state to significantly reduce the bandwidth overhead of state sharding.
- **Multi-phase Commitment:** A unique epoch-based multi-phase commitment technique to guarantee atomicity for arbitrarily complicated cross-shard smart contract transactions.

The paper is organized as follows. Section **??** gives background on state sharding and a family of cryptographic schemes known as vector commitment scheme. Section **??** gives a high-level overview of two important design aspects of Möbius. Section **??** gives a detailed view of our state sharding design. In Section **??**, we present an implementation of our sharding design on the Ethereum blockchain and evaluate its performance in Section **??**. Finally, we discuss the related works in Section **??** and conclude the paper in Section **??**.

II. Background

A. Blockchain State Sharding

An account state-based blockchain like Ethereum [**?**] is constituted of a state machine composed of a global state of accounts. The global state of Ethereum is a mapping from 160-bit account addresses to account states. An account's state includes account information such as the native token balance owned by the account, and the transaction counter or *nonce*. Optionally, an account can host a Turing-complete program called *smart contract*. Such an account has a persistent storage state constituting the *storage trie* and is composed of the contract's state and bytecode.

In existing blockchain platforms such as Ethereum, each full node participating in the network stores the entire global state. However, as the number of accounts and smart contracts increase, the state of the blockchain grows tremendously. For instance, the size of Ethereum's state at the time of this writing exceeds 1 TB [**?**]. This may disenfranchise nodes with "regular" machines from participating thus sacrificing the egalitarian nature of the network. *Blockchain state sharding* allows splitting the global state amongst miner/validator nodes where each node stores a chunk or *shard* of the state to cope with the increasing state size and hardware requirements.

B. Vector Commitment Schemes

A *vector commitment scheme* (VCS) [**?**], [**?**], consists of a hash function that takes a vector $\mathbf{z} = [z_0, \ldots, z_{n-1}]$ and outputs a *commitment* C (also called a *digest*) such that one can create a proof that C is the commitment to some vector where the value at the index i is z_{i}. VCSs such as Hyperproofs [**?**] uses *homomorphic polynomial commitments* allowing participants to update the commitments using only the delta *i.e.,* changes in the data point. For instance, $C' = U(C, \delta, \mathsf{i})$ is the computed new commitment value when the value at the index i of the input vector changes by δ. Here C is the old value of the commitment and U is the commitment update function. Hyperproofs are maintained as a *vector commitment tree* (VCT) that can be updated efficiently. Moreover, Hyperproofs provide a suitable solution for aggregating individual proofs. In [**?**], it was shown that Hyperproofs are $10\times$ faster than the proof aggregation scheme SNARKS [**?**] that aggregates Merkle proofs [**?**].

In a sharded blockchain setting, VCT allows nodes in the network to exchange state data along with the authenticity proofs for generated blocks. The receiving nodes can validate the mined blocks by verifying the authenticity of proofs against the vector commitment for the state that they maintain.

III. Protocol Overview

In this paper, we propose the Möbius protocol to enable a secure state sharded blockchain. Since the transmission of individual proofs over the network will result in network congestion, our protocol uses Hyperproofs to batch multiple individual proofs into a single aggregated proof before transmitting them. Hyperproofs delivers the fastest proof aggregation method out of all maintainable proof schemes. We use a multithreaded extension of Hyperproofs over the single-threaded implementation in [**?**] that allows us to improve the aggregation and verification times for 1024 proofs for a vector of length 2^{24} from 109 and 14 seconds to 6.10 and 0.722 seconds, respectively, on a 16-cores machine. However, although the Hyperproofs implementation in our design can achieve a much improved aggregation time over existing approaches, in practice, the proof aggregation time is still a bottleneck for block production. For instance, considering a block containing 1024 simple payment transactions, the block production time will be at least greater than the 6 seconds (1024 proofs for the sender's balance) necessary for proofs aggregation. Thus, the proof aggregation time will significantly impede the overall transaction throughput.

A. Optimistic Proof Aggregation

To address the issue of time-consuming proof aggregation delay, we decouple proof aggregation from block mining by modifying the mining process as follows. When mining a block, a block producer (*i.e.,* a miner node) along with each transaction also includes the input data required to verify and execute the transaction. For example, the input data for a simple payment transaction is the sender's balance to ensure that the sender has enough balance in their account. At this stage, the block producer does not provide the aggregated proof. However, before a fixed deadline (expressed in the number of blocks), at least one node from the same shard as the miner node needs to provide an aggregated proof, which proves the authenticity of the previously included input data. The deadline is set long enough to provide sufficient time to compute the aggregated proof in parallel with transaction packing. When mining the next block, other nodes in the network optimistically assume that the input data provided by the previous miner node is correct and proceed with block execution even though the proof is yet to be included.

To ensure that malicious nodes do not exploit this trust, the protocol requires a stake deposit for every node in the network. Any dishonest behaviors (inferred through the authenticated proofs) are penalized by slashing the stakes of the malicious nodes. In Section **??**, we detail the incentive mechanisms used in Möbius to discourage dishonest behaviors.

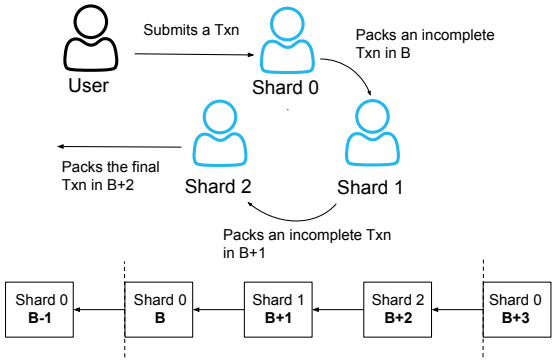


Figure 1. An execution of a successful cross-shard transaction.

B. Atomic Execution of Cross-Shard Transactions

A crucial challenge in any state-sharded blockchain is the handling of transactions that need access to state data from different shards. These are known as *cross-shard transactions.* Möbius ensures the correctness of such transactions by guaranteeing their atomicity and thus preventing interference with other transactions.

Motivational Example: We use the example in Fig. **??** to explain the mechanism used in Möbius that ensures transaction atomicity. The example consists of a cross-shard transaction Txn that first requires data from shard 0, then from shard 1, and finally from shard 2. Txn is executed in three phases where each sub-transaction, denoted by Txn_i, is executed as follows. First, a miner serving shard 0 does preliminary execution of Txn_0 *i.e.,* no state changes are applied, rather, any modifications in state data are stored temporarily. When the preliminary execution halts due to the requirement of state data outside of shard 0, shard 1 in this case, the miner attaches the input data it consumed from shard 0 to Txn_0 and includes it in block B as an *incomplete transaction*. Afterward, the incomplete transaction is resumed as a new sub-transaction Txn_1 by a miner serving shard 1 when producing block $B+1$. The preliminary execution of Txn_1 will be an extension of Txn_0's since the miner now has the data from both shard 0 (from Txn_0) and shard 1. When the above preliminary execution halts and requires state data from the next shard, shard 2, then Txn_1 along with the consumed input data from shard 1 is packed as incomplete in block $B+1$.

Finally, a miner from shard 2 handles Txn_2 when mining block $B+2$. It preliminary executes Txn_2 by accessing the state data from shards 0 and 1 included in the previous incomplete transactions, and data from its own shard. Since all the required data is available this time, an atomic execution is performed *i.e.,* all state changes in the miner's shard are permanently registered. Txn_2 is then packed in block $B+2$ as a *complete transaction* and all nodes in the network register state changes in their respective shards.

In the above example, we assumed that the included data from shards 0 and 1 is not modified by any other transaction until Txn_2 is successfully mined. This assumption, however, might not always be valid. Section **??** describes how Möbius is robust to such in-process state modifications.

IV. Möbius: A Detailed View

The protocol is designed to extend existing account-based blockchain platforms, such as Ethereum, where every miner/-validator node in the blockchain mines/validates all the blocks and there is no execution sharding.

A. States Sharding

In Möbius, the blockchain state is divided into k shards $\mathsf{sh}_0, \ldots, \mathsf{sh}_{k-1}$. The shard of an account ac with address $\mathsf{adr}(\mathsf{ac})$ is given by the following relation:

$$\mathsf{sh}(\mathsf{ac}) = \mathsf{adr}(\mathsf{ac}) \pmod{k} \tag{1}$$

The shard of the state data of ac including the balance and smart contract state in the storage trie is defined to be $\mathsf{sh}(\mathsf{ac})$. The state of a shard sh_i is organized as a vector of constant size sz and their proofs are maintained as a VCT of size sz with a vector commitment $\mathsf{C}(\mathsf{sh}_\mathsf{i})$. The vector and VCT's index for an account ac is derived from its address $\mathsf{adr}(\mathsf{ac})$ as follows:

$$\mathsf{id}_\mathsf{ac} = \mathsf{adr}(\mathsf{ac}) \pmod{\mathsf{sz}} \tag{2}$$

For a smart contract associated with an account ac and a variable ϑ_ac stored at a slot index $\mathsf{i}_{\vartheta_\mathsf{ac}}$ of the contract, a hash key $\mathsf{k}_{\vartheta_\mathsf{ac}}$ is generated by applying the KECCAK256 hashing function on the concatenation of both the contract address and the slot index as follows:

$$\mathsf{k}_{\vartheta_\mathsf{ac}} = hash(\mathsf{adr}(\mathsf{ac}) \parallel \mathsf{i}_{\vartheta_\mathsf{ac}}) \tag{3}$$

The input state for ac is represented as a key-value pair. For ac and its balance, the key refers to $\mathsf{adr}(\mathsf{ac})$ whereas for ϑ_ac, the key refers to $\mathsf{k}_{\vartheta_\mathsf{ac}}$. Similar to above, we use the key $\mathsf{k}_{\vartheta_\mathsf{ac}}$ to generate the VCT index $\mathsf{id}_{\vartheta_\mathsf{ac}}$ associated with ϑ_ac as follows:

$$\mathsf{id}_{\vartheta_\mathsf{ac}} = \mathsf{k}_{\vartheta_\mathsf{ac}} \pmod{\mathsf{sz}} \tag{4}$$

Since the shard of ϑ_ac is the same as that of ac, all the smart contract variables reside on the same shard. This reduces cross-shard communication. Moreover, to ensure that all nodes can validate any transaction, the protocol requires every shard to maintain a smart contract's code.

Every mining node is associated with at least one shard, say shard sh_i, making it a *proof serving node* for shard sh_i or $\mathsf{psn}_{\mathsf{sh}_\mathsf{i}}$. To enroll as a psn, a node deposits a fixed stake sk per shard which is in addition to any other existing stake. As such, $\mathsf{psn}_{\mathsf{sh}_\mathsf{i}}$ maintains the corresponding shard's state, *i.e.,* executing transactions that update the shard's assigned accounts, stores the shard's commitment $\mathsf{C}(\mathsf{sh}_\mathsf{i})$ and the Hyperproofs VCT that is used to generate commitments and proofs. Lastly, every node participating in the network maintains the commitment $\mathsf{C}(\mathsf{sh}_\mathsf{i})$ for every shard sh_i.

B. Transactions in Möbius

The shard of a fresh transaction is the shard of the transaction sender's account. A transaction with a shard sh_i is first handled by $\mathsf{psn}_{\mathsf{sh}_\mathsf{i}}$. We distinguish two types of transactions: single-shard and cross-shard transactions. A single-shard transaction is a transaction whose end-to-end execution requires data from one shard that corresponds to the shard of the transaction's sender account. On the other hand, a cross-shard transaction is a transaction whose end-to-end execution requires data from more than one shard. Thus, the execution of this transaction involves data exchanges between nodes serving different shards. Of course, it is always possible that the different shards involved in a cross-shard transaction are served by the same node.

In Algorithm **??**, we show the function $mine$ to mine a transaction tx. A miner serving the transaction's shard begins preliminary execution of tx using the sub-procedure $executePrelim$. During the execution, the miner performs validation checks, attaches input state (key-value pairs) to tx from its shard and no state changes are applied. If tx is successfully executed, the miner applies the state changes to

its shard and packs tx in the block. This corresponds to the case for a single-shard transaction if the miner received tx directly from its sender. On the other hand, if the miner cannot complete executing tx due to the lack of data from some shard $\mathsf{sh_j}$, it marks tx as incomplete, then it attaches the gas used, it sets the next shard of tx to $\mathsf{sh_j}$, and finally it includes tx in the block.

An incomplete transaction flag instructs the node verifying blocks to not apply changes to the state. If a miner serving shard $\mathsf{sh_j}$ executes the above block containing incomplete tx, as shown in the function $execute$, and decides to continue executing tx, it adds tx to the pool of incomplete transactions. When this miner produces a future block, it creates a new transaction tx' that points to tx and then preliminary executes tx'. If successful, it marks tx' as complete, appends gas information, and applies state changes. If unsuccessful, it sets the next shard for tx', appends gas, and adds it to the block for further execution. This process can go on until the first transaction (in the very first phase of execution) reaches its lifetime.

Thus, nodes executing a block only apply changes to the state if a transaction is complete. For a cross-shard transaction, state changes are applied after the last complete transaction is successfully executed. Incomplete transactions are therefore used to collect data from different shards before their final execution. This allows the protocol to implement an atomic execution of cross-shard transactions.

Algorithm 1: Transaction mining and execution.

```
Function mine(tx):
    (done, nextShard, gas) = executePrelim(tx);
    if done then
        state.applyChange();
        tx.setComplete(gas);
    else
        tx.setIncomplete(nextShard, gas);
Function execute(tx):
    executePrelim(tx);
    if tx.isComplete() then
        state.applyChange();
    else
        incompletePool.add(tx);
```

C. Authenticated Proofs in Möbius

Transaction blocks are generated and clustered in *rounds*. The k^{th} block in the canonical chain belongs to the round $\lfloor \frac{\mathsf{k}}{\mathsf{R}} \rfloor$ where R is the number of blocks in one round. The updates to a shard's proof and commitment occur at the beginning of each round and they remain unchanged throughout the round. In the round $\mathsf{r}+1$, the psns serving a shard $\mathsf{sh_i}$ submit authenticity proofs for all input data from this shard that were included in transactions packed in round r. The submitted proofs certify the authenticity and correctness of the transactions that were packed in round r.

A block producer always attaches the input state from its served shard to a transaction in the form of key-value pairs. This input data is sufficient to execute the transaction and update all the shard's commitments whose data is modified after the transaction's execution. For instance, if a data point ϑ_{ac} in shard $\mathsf{sh_i}$ is set to a new value during transaction execution, the block producer includes the previous value for the key $\mathsf{k}_{\vartheta_{\mathsf{ac}}}$ in the transaction so that all nodes can update $\mathsf{C}(\mathsf{sh_i})$ and VCT (if any). If a transaction Txn is included in the block b in round r, the value in the data tuple that it contains should correspond to the *beginning of the round* r. Because each node executes every block, it is able to fetch the latest value for a key even if the value has changed since the beginning of the round. Thus, the protocol implicitly requires nodes to maintain a temporary state to fetch latest values. An implementation of this for Ethereum is presented in Section **??**.

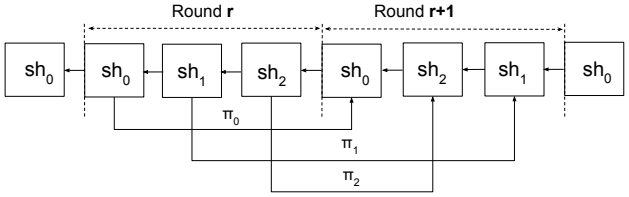


Figure 2. An instance of a successful round r as the psns provide timely aggregated proof before the end of round $r+1$. sh represents the shard of the block producer.

In Fig. **??**, we show an instance of Möbius network that has 3 shards and a round consists of 3 blocks. In round r, the psns associated with shards sh_0, sh_1, and sh_2 generate one block each. In round $\mathsf{r}+1$, the aggregated proofs for the updates to shards sh_0, sh_2, and sh_1 in round r are posted in the first, second, and third blocks respectively.

Delayed aggregated proofs: To disjoint the proof aggregation delay from block production time, we ensure that the delay provided by a round to produce R blocks is sufficient for a node to aggregate n individual proofs from a previous round. In particular, let $D_{\mathsf{sh_i,r}} = [d_{\mathsf{sh_i},0}, d_{\mathsf{sh_i},1}..d_{\mathsf{sh_i,j}}]$ represents all the key-value pairs from shard $\mathsf{sh_i}$ that are attached to the transactions packed in blocks of round r. Then, all the psns serving $\mathsf{sh_i}$ collectively generate a batch of aggregated proofs $\pi_{\mathsf{sh_i,r}}$, propagate them into the network by wrapping them as a proof transaction that are appended to the blockchain before the end of the round $\mathsf{r}+1$. The batch $\pi_{\mathsf{sh_i,r}}$ can consists of several aggregated proofs $\pi_{\mathsf{sh_i,k}}$, each representing n individual proofs for a fixed number n. For instance, for $\mathsf{n} = 1024$, $\pi_{\mathsf{sh_i,k}}$ contains proofs for the data $[d_{\mathsf{sh_i},1024\mathsf{k}}, d_{\mathsf{sh_i},1024\mathsf{k}+1}..d_{\mathsf{sh_i},1024\mathsf{k}+1023}]$. Since every node updates the commitments for each shard at the beginning of a round, then an aggregated proof batch $\pi_{\mathsf{sh_i}}$ from shard $\mathsf{sh_i}$ is verified using the commitment from the previous round $\mathsf{C}(\mathsf{sh_i})$, *i.e.,* $Verify(\pi_{\mathsf{sh_i}}, \mathsf{C}(\mathsf{sh_i}))$.

Let T_π denote the time to aggregate n proofs and let T_R denote the time to produce R blocks which is also the length of a round. Möbius ensures that $T_\pi \leq T_R$. Note that a block containing input values is not confirmed until a valid aggregated proof is included in the chain. This period can be as long as $2 \times T_R$ when a block is produced at the beginning of some round and the aggregated proof for the block's input data is provided at the end of the next round.

D. A Multi-layered Vector Commitment Scheme

A VCS such as Hyperproofs uses pre-computed *proving keys* that are used to generate proofs. The size of these proving keys is proportional to the size of the underlying vector and becomes as large as 96 GiB for a vector of size 30 bits. On the other hand, keys derived in (**??**) are usually of size 256 bits. This induces a possibility of collision when mapping an address or smart contract key to a VCT index. Increasing the VCT any further requires more storage and defeats the original goal of state sharding. Therefore, we propose multi-layered Hyperproofs to prevent collision amongst addresses

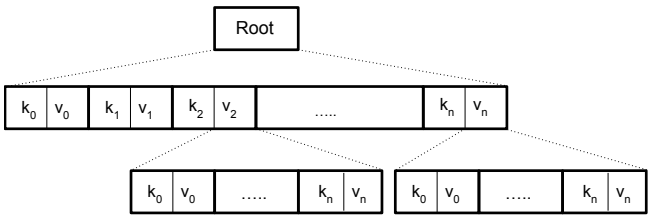


Figure 3. Multi-layering of a vector commitment scheme.

while each level Hyperproof is less than 30 bits. As shown in Fig. **??**, when the index k_2 of address adr'_2 collides with the index of address adr_2, a new VCT is generated and the digest of the newly generated VCT is stored as the value of the index k_2. Given that the original VCT has a length of 2^{l_1} and the subsequent layers have length 2^{l_2}, the effective VCT has the length of $2^{l_1+l_2}$ since each slot can now host a 2^{l_2}–sized VCT. This scheme does not require an overhead with regard to Hyperproofs proving key since the same keys can be reused for the subsequent layer as long as $l_2 \leq l_1 \leq 30$. If there is a collision in indices in both layers, a third subsequent layer can be generated similarly.

E. Reward and Fee Upgrades

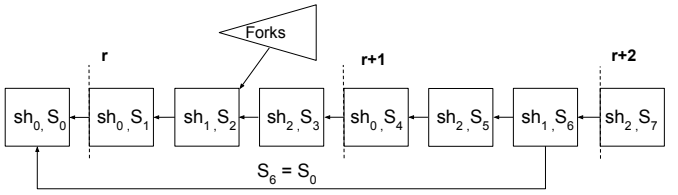
The reward mechanism is designed to prevent imbalanced state sharding. The protocol allows shards to have different gas prices to prevent the concentration of smart contracts accounts in certain shards thus leading to imbalances. Besides the ordinary block reward and transaction fee, transactions that include aggregated proofs get rewards as well. In particular, let sk be the proof sender's stake in the shard sh_i where $\mathsf{SK}_{\mathsf{sh}_\mathsf{i}}$ is the total deposited stake. Then, the associated reward for the aggregated proof transaction, π, is $reward_{\pi,\mathsf{sh}_\mathsf{i}} = \frac{\mathsf{sk}}{\mathsf{SK}_{\mathsf{sh}_\mathsf{i}}} \times \mathsf{Rw}$ where Rw is some constant reward.

Note that a shard with a large transaction volume requires a proportionally large number of aggregated proofs since each aggregated proof contains a fixed number n of key-value pairs. Thus, the above reward structure incentivizes the psns to enroll in shards with either very few psns or with a very large transaction volume. For cross-shard transactions, the gas fee is distributed amongst miners who participate in the execution of the transaction and include data from their shards. The fee is distributed after the complete transaction is executed.

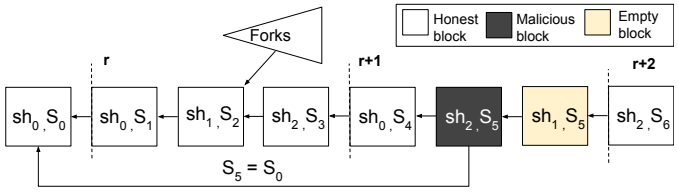
F. Failure to Provide Proof

The protocol is designed with an incentive mechanism to discourage nodes in the network from misbehaving. If the psns serving a shard sh_i fail to include the proof of any data point that was included in a round r before the end of the next round $\mathsf{r}+1$ (shown in Fig. **??**), portions of their stake deposits are slashed. The slashed amounts are distributed to honest psns from other shards who produced correct blocks and provided on-time aggregated proofs as incentives. Another case corresponds to the scenario where a block producer serving shard sh_i includes an incorrect input value in a transaction packed in a round r (shown in Fig. **??**). To detect this malicious behavior, a fraud-proof, i.e., the fraud-detected key and the correct value, must be generated by another psn serving the same shard certifying that the block producer generated incorrect values. Möbius in this case slashes the deposit of the malicious block producer to distribute it to the rest of the honest participants in the rounds r and $\mathsf{r}+1$.

In both cases, the states of all the shards revert to their states at the beginning of round r and all the blocks included in round r and $\mathsf{r}+1$ are considered invalid. In the second case, any further blocks subsequent to the malicious block until the conclusion of round $\mathsf{r}+1$ remain empty. Therefore, the reversions only affect the state while the underlying consensus protocol and fork rules remain unaffected.



(a) psns fail to aggregate proof for round r before round $\mathsf{r}+1$ ends



(b) A malicious block is spotted

Figure 4. Examples of state reverts for two cases. sh represents shard of the block miner and S represents the state.

V. Implementation

In this section, we describe an implementation of the proposed sharding protocol on OpenEthereum [**?**], one of the fastest Ethereum clients written in the Rust programming language. The consensus mechanism of the blockchain uses the Proof-of-Authority engine consisting of permissioned authority nodes generating blocks in a round-robin fashion. The number of blocks R in a round equals to the number of authority nodes. For proof aggregation, we use the Hyperproofs library written in the Golang programming language that we subsequently connect to OpenEthereum. In Figure **??**, we show the modules of OpenEthereum that were modified and their interactions with each other and with the Hyperproofs library.

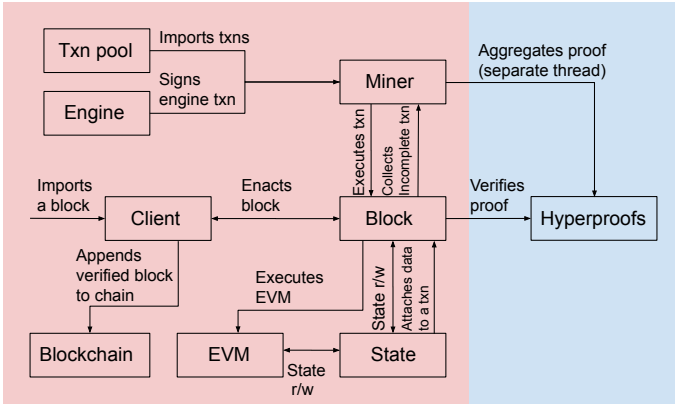


Figure 5. Möbius client architecture. The portion in red is implemented in Rust while the portion in blue is implemented in Golang.

A. OpenEthereum

We describe here the modifications to OpenEthereum to implement the transaction handling mechanisms of Möbius.

Transaction: A regular Ethereum's transaction consists of fields such as the addresses of the sender and receiver. We extend such a transaction to include the following additional fields:

- aggregated Hyperproof serialized to a string (optional);

- keys corresponding to the proof (optional);
- key-value tuples for the transaction's input data;
- gas consumed so far;
- current and next shard;
- a flag for complete and incomplete transaction;
- block number of the transaction from sender;

The key used here is 160 bit uninterpreted hash type and the value is a 256 bit unsigned integer.

Miner: This module packs transactions collected in the transaction pool and produces a block. With our protocol, the miner only includes transactions that belong to the client's shard. The miner also creates and signs transactions, using the *Engine* module, to include aggregated proofs from the previous round. If the *Client* module receives an incomplete transaction with the client's shard as its next shard, it passes the transaction to the Miner, which in turn creates and signs a new transaction pointing to the previously received transaction.

Additional temporary state: The client maintains a map $tInput$, which consists of the input key-value tuples (corresponding to the beginning of the round) collected for a transaction. Also, the client maintains the latest value of the keys that are modified for all the shards in $mLatest$. The client periodically flushes all the keys in $mLatest$ that have been stored for L rounds. This serves as a temporary global state with a predetermined lifetime allowing the client to fetch the latest value of the keys included in a transaction. With this, $\mathsf{L} \cdot \mathsf{R}$ becomes the lifetime of a transaction where R is the number of blocks in a round. In addition to this, whenever the value of a key in the client's shard is modified, it stores the value corresponding to the beginning of the round in $mRound$. This storage is renewed at the start of every round. Lastly, the client maintains a top-level cache $mExec$ to store the value of the keys that are modified during transaction execution.

When the client wants to fetch the latest value v_l for a key k, when calling $SLOAD$ opcode or $balance$ method, it uses the function `fetchLatest` given in Fig **??**. It first looks in $mExec$ and if not found, looks in $tInput$. If k does not belong to the client's shard (returned by `matchShard`), then $tInput$ must contain k otherwise the transaction is marked as incomplete and $mExec$ is cleared. If k matches client's shard and not found in $tInput$, it is fetched from the state. Otherwise if found in $tInput$, the client chooses the latest value between $tInput$ and $mLatest$. Note, every time a key is fetched from the state, its value at the beginning of the round is inserted in the transaction and $tInput$.

The client executes `store` when it wants to store k with value v for opcode $SSTORE$, and methods $balanceAdd$, and $balanceSub$. The key is first stored in $mExec$. Then, it fetches the previous value v_p using `fetchLatest`. Afterwards, it dummy-inserts (denoted by `dInsert`) k, v to $mLatest$ and if the key belongs to the client's shard, dummy-inserts k, v in the state and dummy-inserts k, v_p in $mRound$. v_p is also important because the client needs to know v_l–v_p to update Hyperproof tree and commitments. If the transaction is fully executed, every dummy-insert becomes permanent, else, dummy-inserts are ignored.

Transaction verification and execution: The *Block* module performs the transaction verification either when the Miner packs transactions or when the Client enacts an externally imported block. The verification includes checking the validity of the transaction nonce and sufficient balance for value transfer and gas fee. If verified, the transaction is then executed using the *EVM* module. We modified the EVM module of the client for only those opcodes that either read from or write to the state shard. This includes SLOAD, SSTORE, BALANCE and SELFBALANCE. If during smart contract execution, the EVM module is not able to fetch the required data, the execution is stopped and the incomplete flag and the next-shard flags are set.

```
fn fetchLatest(k) -> <value, status> {
  if mExec.contains(k)
    v_l = mExec.get(k)
  else if tInput.contains(k)
    if mLatest.contains(k)
      v_l = mLatest.get(k)
    else v_l = tInput.get(k)
  else
    if k.matchShard()
      v_l = State.get(k)
    else
      mExec.clear()
      return <none, false>
  return <v_l, true>
}
fn store(k,v) -> status{
  mExec.insert(k,v)
  <v_p, status> = fetchLatest(k)
  if status
    mLatest.dInsert(k,v)
    if k.matchShard()
      State.dInsert(k,v)
      if !mRound.contains(k)
        mRound.dInsert(k,v_p)
  return status
}
```

Figure 6. Variable fetch and store algorithms.

B. Hyperproofs

We glued Hyperproof with OpenEthereum using the Foreign Function Interface (FFI). The Hyperproofs library creates and maintains an instance of the Hyperproofs VCT for the client's shard. It also maintains and updates commitments for every other shard. When shard state changes occur, the client shares the VCT index, the delta in the value, and the shard id with the Hyperproofs module. At the beginning of every round, the Hyperproofs library updates the commitments and the tree using the accumulated data and the update keys[1] and stores the previous round's commitment. When the Miner finishes producing a block, it shares the transaction input key-value tuples with the Hyperproofs module, which then starts aggregating the proofs in a parallel thread. Threads communicate via asynchronous channels and the Miner fetches proofs from the channel before its turn in the next round. Lastly, during transaction execution, if the client discovers an aggregated proof, it calls the Hyperproof module for its verification.

C. A Theoretical Throughput Upperbound

Assume an aggregated Hyperproof has 2048 proofs. In the above implementation, each authority node runs proof aggregation in parallel to the execution of imported blocks. During a round with R blocks, an authority node computes one aggregation, $R-1$ proof verifications (1 for each imported block) and transaction executions. Therefore, the theoretical limit for throughput at maximum load is given by $\frac{n_{tx}}{t_{total}}$ where:

$$t_{total} = t_{ex} + t_{agg} + (R-1) \cdot t_{ver} \tag{5}$$

Here n_{tx} is the total number of transactions executed in R blocks, t_{ex} is the time to execute n_{tx} transactions, t_{agg} is the

[1]Hyperproofs require a trusted setup to generate $\mathcal{O}(n)$-sized public parameters for the update keys where n is the size of the vector. Hyperproofs [**?**] highlights this issue and proposes to implement the trusted setup using multi-party computation protocols that are shown to be viable for $n \leq 2^{27}$. Since Möbius employs multi-layered VCS, we can use smaller vector sizes with $n \leq 2^{27}$.

Table I
COMPARISON OF TRANSACTION SIZE

Name	Key-val Pairs	Legacy (B)	Single Shard (B)	Aggregated Hyperproof (B)	Proof w/o Aggregation (B)	Merkle Proof (B)
Payment	1	107	153	36.9	1609	173.20
ERC-20	3	201	306	110.7	4827	751.38
Swap	12	363	724	442.9	19308	3227.52

time to aggregate 2048 proofs, and t_{ver} is the time to verify an aggregated proof. n_{tx} can be a maximum of $2048 \cdot R$ since a block has 2048 key-value slots (same as the size of batch of proofs). Neglecting t_{ex}, we get a theoretical upper bound of:

$$\frac{2048 \cdot R}{t_{agg} + (R-1) \cdot t_{ver}} \tag{6}$$

for throughput of transactions utilizing 1 key-value slot. Thus, this value is also equal to the state access throughput. In the above example, the values of t_{agg}, t_{ver} are $12.11s$, $1.45s$ respectively. Therefore, for $R \geq 9$, the denominator in (**??**) is dominated by t_{ver}. In other words, for a round size greater than 9 blocks, proof verification becomes the throughput bottleneck.

VI. EVALUATION

In this section, we evaluate the performance of the Möbius upgrade implemented on OpenEthereum. Our experiments answer the following questions: 1) How does the state of a node vary with increasing shards? 2) What is the reduction in bandwidth requirement in Möbius compared to conventional methods? 3) How close does our implementation stand to the theoretical state access throughput? 4) How well does Möbius atomically execute complicated cross-shard smart contract transactions? 5) What happens when a node provides false data in a transaction?

A. Methodology

The following experiments are performed on 16 authority nodes with equal nodes per shard on *AWS EC2 c6a.8xlarge* with 32vCPU, 64GB memory, and 1TB SSD storage. Each block contains a maximum of 2048 slots for key-value pairs that can be used by the transactions.

For the evaluation of performance metrics such as transaction throughput, we use a transaction benchmark trace of size 16k. This size of trace saturates the throughput for each subsequent experiment thus giving a fair comparison between them. We consider three types of transactions:

- Simple value transfer: this transaction involves a sender transferring the blockchain native currency (for example Ether for Ethereum) to a recipient.
- ERC20 token transfer: this transaction calls an ERC20 smart contract transferring the contract token by debiting sender's balance and crediting the receiver's. For cross-shard trace, we deployed four distinct contracts, one on each shard. The call to the contracts is distributed uniformly.
- Token swap: this transaction calls *Uniswap-v2* which is a decentralized exchange smart contract that swaps ERC20 tokens. A user can specify the minimum amount of $token\ B$ that they expect for a given input of $Token\ A$ *(SwapExactTokensForTokens)* or SETFT, or specify a maximum amount of input $TokenA$ for a given amount of output $TokenB$ *(SwapTokensForExactTokens)* or STFET. For cross-shard trace, we deploy two Uniswap smart contracts and two ERC20 token smart contracts on four distinct shards. The transaction trace uniformly calls STFET and SETFT methods.

We quantify the complexity of a cross-shard transaction using the *hops* metric. Hops is the number of blocks in which a transaction is included, either as incomplete or complete, until successful execution. As we will see in the subsequent sections, a complex swap transaction can take as many as 6 hops before successful execution.

B. Result summary

1) **State size vs shards:** Fig. **??** shows the size of the state of the client averaged over all the participant nodes. Our results show that given a uniformly distributed load over the shards, state size reduces proportionally with the number of shards. This highlights that Möbius solves the problem of the overwhelmed disk utilization by blockchain state.
2) **Bandwidth comparison:** Table **??** shows the size of transactions and authenticated proofs for different types of transactions. The second column represents the number of input values required for execution and included in the modified transaction. The third and fourth column represents the size of Ethereum transaction and Möbius transaction in a single shard setup respectively. The fifth and sixth column shows the amortized size of Hyperproofs with and without aggregation respectively. The last column represents the corresponding Merkle proof size. Aggregation is effective as it compresses Hyperproofs by $43.6\times$. This reduces the bandwidth requirement in Möbius by 4.7–$7.3\times$ compared to contemporary designs utilizing Merkle proofs. To achieve this reduction, we pay a cost of 43–99.4% overhead in transaction size, which is small when compared to proof reduction benefits.
3) **Theoretical throughput:** We accelerate the Hyperproof aggregation and verification by more than $10\times$ and t_{agg}, t_{ve} for a batch of 2048 proofs equals $12.11s, 1.45s$ respectively. According to (**??**), the theoretical limit for state access throughput turns out to be $967.7/s$. Our implementation of Möbius achieves a throughput of $766.9, 744.2$, and 732.8 in payment, ERC20, and swap transactions respectively for a single shard setup. This accounts for 20.8–24.3% deviation from the theoretical upperbound. This shows that as a result of efficient packing of Hyperproofs in Möbius, throughput is primarily bottlenecked by proof verification delay[2].
4) **Cross-shard accounting:** Table **??** shows the throughput and gas consumption for an extreme case of cross-shard smart contract transactions where all the contracts reside on four different shards. The second column shows the average hops taken by the transaction. The third and fifth columns show the transaction throughput and gas consumption while the fourth and sixth columns show the throughput penalty and gas overhead relative to their single shard counterparts. Our results highlight that Möbius maintains atomicity even while executing a cross-shard smart contract transaction requiring 5.75 average hops. This comes at a cost of throughput penalty (between 23.2–38.2%) and gas overhead (between 35.4–92.4%) since each miner executes the smart contract from the beginning during each hop.

[2]Note that the proof aggregation and verification benchmarks do not include the time to read the setup (public parameters). Since the setup is read once in the beginning and maintained in memory, this accounts for a fixed cost.

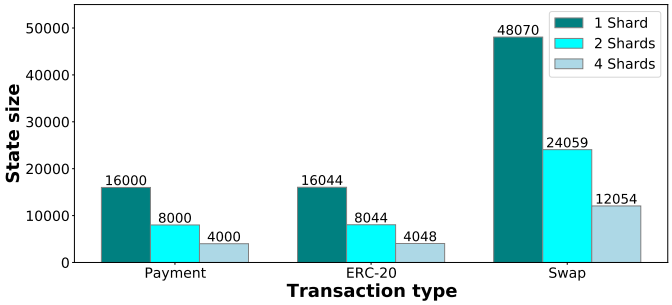


Figure 7. Reduction in average state of a node with increasing shards.

Table II
COMPARISON OF CROSS-SHARD TRANSACTIONS

Name	Average hops	Transaction throughput	Throughput penalty (%)	Average gas (Kgas)	Gas overhead (%)
Payment	1	766.87	0	20.99	0
ERC-20	1.75	190.48	23.2	71.14	35.4
Swap	5.75	37.74	38.2	305.25	92.4

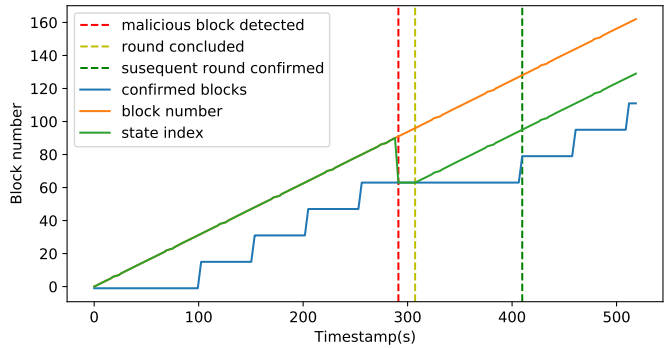


Figure 8. Block evolution when a block providing malicious data is detected.

A trace of real-world ethereum transactions was analyzed in [?] showing that cross-shard transactions account for less than 20% of total load for 10 shards. Therefore, the cross-shard overhead in practical scenarios reduces further in expectation.

5) **Malicious behaviour:** Fig. ?? shows an instant of attack where a miner includes false data for a transaction in the 4^{th} round. The proof for correct value was provided in the 5^{th} round in block number 91 and the state reverted to the last confirmed block number 63 (beginning of round 4). Then, empty blocks are generated until block number 95 and a new round (round 6) began afterwards. After conclusion of two rounds marked by block number 127, round 6 is marked confirmed. This shows that Möbius quickly recovers from incorrect state data attack.

VII. Related Work

Full Sharding: This scheme partitions the network of miners into shards to scale blockchains with parallel execution at the cost of reduced security. The earlier attempts including Elastico [?], and Omniledger [?] implement full sharding on a UTXO-based blockchain. However, the global state is maintained by all the miners. Designs like Rapidchain [?] implement full sharding while allowing all nodes to maintain a fraction of the state whereas designs like SSChain [?] require certain nodes to maintain the global state and handle cross-shard transactions.

Subsequent to these, several full sharding designs have been proposed for account-based blockchains. Zilliqa [?] implements this with all nodes maintaining the global state. Pyramid [?] uses the help of b-nodes that serve as a bridge for a cross-shard transaction. Other works such as Monoxide [?] handle cross-shard transactions by splitting their execution based on the shard of the input state. Similar ideas of partial execution of cross-shard transactions have been implemented in Elrond [?], Nightshade [?], and Aeolus [?]. Unlike the above designs, Möbius does not sacrifice the security of miner's network as each block is executed by all miner nodes.

State Sharding: Blockchain state sharding is an active area of research [?], [?], [?]. Vault [?], primarily focused on fast bootstrapping, implements state sharding for account-based blockchains. It introduces adaptive sharding of the Merkle tree to reduce the size of a Merkle proof. Although this works well for payment transactions, Vault's design cannot handle cross-shard smart contract transactions. A more recent system, RainBlock [?] shards the Merkle tree into subtrees such that each subtree is kept in a storage node and proposes I/O-helpers nodes that pre-fetch transaction execution state from storage nodes for the miners. This decouples I/O from the critical path of transaction processing resulting in a higher transaction throughput.

Layer-2 and Rollups: Rollups [?], [?], [?], [?] extend the state of the main chain by moving data and computation off-chain. Compared to sharding, rollups allow to scale both the state and throughput of a blockchain without significantly compromising security (especially when using zero knowledge (ZK) [?] based rollups instead of optimistic rollups). However, rollups have certain limitations in functionalities: Unlike sharding, two rollup states are independent of each other and cannot communicate without a bridge. For instance, one cannot simply swap two ERC20 tokens with addresses that are stored on two different rollups. On the other hand, a state sharding protocol like Möbius with a cross-shard capability allows to scale blockchain state without compromising security and maintaining the functionality of cross-shard interactions. This however comes with certain throughput costs for cross-shard transactions.

Proofs and commitments: Möbius uses authenticated proofs allowing nodes with partial or no state to validate blocks. Authenticated proofs including vector commitment schemes have been studied for stateless clients [?], [?], [?], [?], [?]. EDRAX [?] was one of the earliest works on stateless clients for payment-only cryptocurrencies using VCS. Pointproofs [?] extends the above idea by including the functionality of smart contract execution. Finally, Hyperproofs [?] uses homomorphic commitments that are efficient to maintain, and fast to aggregate, thus extending the functionality of proof serving nodes.

VIII. Conclusion and Future Work

Our theoretical & experimental results for Möbius demonstrate that state sharding is a practical solution to the ever-growing blockchain state size bottleneck. With its novel VCT structure and multi-phase commitment techniques, Möbius significantly reduces the network overhead of state sharding and preserves transaction atomicity for cross-shard smart contract transactions.

In the future we might extend Möbius along a few dimensions. First, we might extend Möbius to be both an execution and state-sharding protocol. Second, we might investigate borrowing the idea of I/O-helpers nodes from RainBlock [?] to speed-up the execution of cross-shard transactions in Möbius.