MultiVAC Sharding Yellowpaper
The All-Dimensional Sharded Blockchain

The MultiVAC Foundation *
core@mtv.ac

September 2018

Scalability is the major bottleneck preventing blockchains from reaching industrial capacity. Systems tackling the most promising scaling solutions thus far, blockchain sharding, have produced progress in parallelizing transaction processing but have not achieved full sharding needed for total scalability. MultiVAC was completely designed to serve as a solution: the world's first fast, efficient, and all-dimensional sharded blockchain designed for total scalability, performing sharding parallelization not only for computation but also transmission and storage. In this paper, we present an overview of MultiVAC's sharding and storage solution. MultiVAC uses the Proof of Stake mechanism to prevent Sybil attacks and uses Verifiable Random Functions to divide the network into fragments called shards. Each shard processes transactions in parallel. MultiVAC provides an elegant distributed storage solution for the blockchain, producing a robust architecture that divides storage and transmission across shards. In this fashion, MultiVAC provides a blockchain throughput that can serve the real economy while maintaining blockchain's core values of decentralization, equality, and security.

1 Introduction

Since the publication of the Bitcoin White Paper [1] by Satoshi Nakamoto in 2008, Blockchain technology has taken the world by storm. As exemplified by Bitcoin and Ethereum [2], it has been subject to both scrutiny and progress and today finds commercial usage in fields as diverse as cross-border settlement, supply-chain management, entertainment, and investment.

However, the technology is at a major bottleneck. Low levels of transactions per second (TPS) saddle blockchain systems, creating constraints on their real-world use. Bitcoin's throughput is on average only 4-5 transactions per second (tps) and Ethereum's is 10 tps, causing rampant congestion and backlog for today's best known networks. All this while the dominant provider of payment solutions, VISA, easily processes 2000 tps on average and 45,000 tps at maximum capacity. The current rate of blockchain development is nowhere near capable of servicing millions of transactions that businesses conduct on a routine basis. The entire industry is cognizant that speed and scalability are the factors determining whether or not the use of blockchain will become ubiquitous in modern society.

MultiVAC believes that in the coming decade, blockchain will be pervasively employed for any type of transactions imaginable. To obtain this ease of use, the blockchain community needs to overcome numerous scalability problems, a challenge we have accepted. Academia and the industry have provided numerous proposals to scale blockchains, and we roughly summarize them below. They can be categorized into three camps: partial centralization, off-chain scaling, and on-chain scaling.

1.1 Blockchain Scaling Approaches

- **Partial centralization** is an approach that allows the bulk of consensus processing in a blockchain system to be handled by a small number of high-powered nodes called supernodes. This approach is exemplified by EOS [3], IOST [4], and the root chain system of Quarkchain [5]. In a traditional blockchain, the network's processing capacity is

*This yellowpaper is produced by the MultiVAC technical team under the direction of Dr. Xiang Ying (email: shawn@mtv.ac). We wish to thank Junyu Lu, Zhengyuan Ma, Dr. Ge Li, Zhong Dong, Liang He, Yuan Li, Libo Shen, Weihua Zhang, Nan Lin and Jiajun Wu, Dr. Xiao Tong, Koupin Lyu, Dr. Minqi Zhang, Dr. Hong Sun, David Sebaoun and Lu Lu for their contributions.

bottlenecked by the processing capacities of individual nodes. Partial centralization gives network stewardship over to nodes that fulfill a particularly high processing capacity, thus bringing up the network's total throughput. Such a system is fast but cuts out ordinary users, creating a semi-centralized system that loses blockchain's original value proposition: decentralization.

- **Off-chain scaling technologies** can be roughly divided into side-chain and state channel approaches to scalability. These approaches attempt to side-step a blockchain's performance bottleneck by processing most transactions outside of the main blockchain. Typical **side-chain** schemes like Cosmos [6] and Aelf [6] process transactions by interfacing the main chain with the side chain, increasing throughout by and order of magnitude. They however also create even more security and performance risks, including cross-chain trust limitations and bottlenecks arising from overburdened main chains. **State channel** schemes are represented by Plasma [8] and the Lightning Network [9] provide specialized transaction channels between users that are based outside of the main blockchain, similarly allowing throughput to scale. They, however, critically require trust requirements between the on- and off-chain links which carry high security risks, reintroducing the central point of failure that blockchain systems were built to avoid.
- **On-chain scaling technologies** roughly include the use of directed acyclic graphs (DAG) and blockchain sharding. Both systems try to change the structure of the blockchain itself to achieve scalability. Directed Acyclic Graphs (DAGs) are represented by Iota [10], Hash Graph [11], Vite [12] and Conflux [13] and is an alternate graph-based data structure which allows blocks to be added asynchronously. DAGs show promise and can run extremely quickly in the laboratory but have proven difficult to apply commercially because they require a huge number of transaction broadcasts which causes a network storm that makes them unwieldy in an open environments. **Sharding** is represented by Zilliqa [14], Quarkchain [5] and Ethereum [15] and is a method of breaking up the processing of transactions on the blockchain into sub-networks called shards. Sharding is a commonly used scalability mechanism in distributed databases and can also significantly improve throughput. It has seen significant accomplishment in public arena, with Zilliqa, for example, having stably run over 2000 tps [14] on its testnet. At the same time, all of the most well-known sharding mechanism only perform partial sharding: They split up the processing labor but still impose the full burden of the network's storage and transmission needs on every node. This inevitably only increases the blockchain's bottleneck.

1.2 MultiVAC Design Principles

MultiVAC was designed with the following question in mind: "Which features of a blockchain are necessary for it to scale to be used on an industrial scale and serve the real economy?" Several conclusions are apparent.

An ideal blockchain scalability architecture should be decentralized at its core, presicely because blockchain technology is touted for its decentralized nature. Blockchain technology was designed to provide a decentralized computing platform to avoid central points of failure, at the cost of requiring a consensus algorithm to finalize outcomes. Reverting to a semi-centralized system would not be a step forward towards a system robust enough to be at the forefront of global commerce. Moreover, an ideal scalability architecture should consist of fundamental upgrades to blockchain at the base layer. Off-chain approaches are thus not ideal as they function more like patches which are sometimes helpful but often bring their own security risks. For this reason, MultiVAC improves blockchain architecture from the bottom-up, using the most dependable of the scaling solutions thus far, blockchain sharding.

Sharding is the parallelization of a blockchain's consensus process.This is similar to how large-scale processing of corporate data today is mostly conducted by multiple servers in parallel. Conducting blockchain operations in parallel is seen as essential to making them capable of handling the computational demand of large corporations. Sharding is the only practical and manageable, and base-layer method to scale a blockchain to industrial capacity while still maintaining a decentralized network with open participation. For this reason, scalability efforts of the best known blockchain networks including Ethereum have been focused on blockchain sharding. There are, however, inherent difficulties of designing sharding over an originally non-sharded system. MultiVAC has the advantage of being a fully sharded system from inception.

Our sharding approach differs from and improves on almost all of the sharding methods existing to date. Central to MultiVAC's sharding design is the important observation that computers do more than just compute data: a large volume of their work is also in transmitting and storing it. Thus, we designed MultiVAC for all-dimensional sharding at the core: sharding not only for computation but also for data storage and transmission, the first blockchain system to design for all three. With a fully sharded architecture, MultiVAC is the first blockchain able to practically and robustly handle industrial-level demand.

We summarize MultiVAC's core technological advantages described in the below paper:

1. Transaction sharding alone is not sufficient to solve blockchain's scalability problem. We are the first

blockchain system providing sharding for transactions, transmission and storage.
2. Fairness, reliability, and security, and are essential to blockchain systems. We utilize a fair resharding technology based on Verifiable Random Functions (VRF) which ensures the reliability and security of every shard.
3. Cross-shard communication is a difficult problem that must be addressed by every sharding system. We design a compact UTXO ledger and decentralized Merkle Root data structure which enables asynchronous secure cross-shard communication, trusted third-party data storage, and trusted data verification in a sharded data environment.

2 MultiVAC Protocol at a Glance

We first present an overview of how MultiVAC achieves sharding for processing, storage and transmission. We use the following legends in the diagrams :

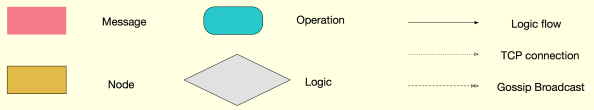


Figure 1: *Legend for Figure 2 and Figure 3*

2.1 Foundations at a Glance

Nodes Nodes are the different types of machines connected to the network. MultiVAC uses three types of nodes: Light Nodes, Miner Nodes and Storage Nodes. **Light nodes** or *clients* are nodes which submit new transactions and perform no processing. They function like users in the system who own accounts and make transactions. **Miner nodes** are nodes running the consensus algorithm and are reassigned to different shards or fragments of the network every couple of minutes. They are the bookkeepers in the system with very low hardware barriers-of-entry, and are incentivized for running consensus. **Storage nodes** are nodes assigned to particular shards that are responsible for storing and serving up transactions. They function like utilities similar to the internet infrastructure on which the network is run, providing services to miners that allow them to perform their tasks more quickly and efficiently. A large number of each node type make up the MultiVAC network.

Verifiable Random Function MultiVAC uses **Verifiable Random Functions** (VRF) to allocate miners to shards, and to further select subgroups of sharded miners to complete bookkeeping tasks. A VRF is a pseudorandom function that produces a trusted source of randomness, that is, a function that both acts as a random number generator for a node in a trustless network and also produces a proof statement allowing other nodes to verify the generated number is legitimately random and not manipulated in any way. The VRF has two other features; its unpredictablility ahead of time and unbiasedness over time. The VRF requires other miners to verify that the node assignment is fair before assigning miners to tasks.

Node Setup All nodes are connected over a semi-synchronous network such that almost all miners can communicate with each other within a certain small time limit. Each node has a cryptographic public key and private key pair. The private key is only known by the node while the public key is known publicly. The node may sign computations with his private key which is then verifiable by all other nodes using his public key.

2.2 Dynamic Sharding

MultiVAC utilizes a *dynamic sharding* mechanism, where miners are allocated to shards which are dynamically changed every couple of minutes. New miners can join a shard when it is periodically re-allocated and miners are not limited to participating in just one shard. The entire process is described in Figure 2.

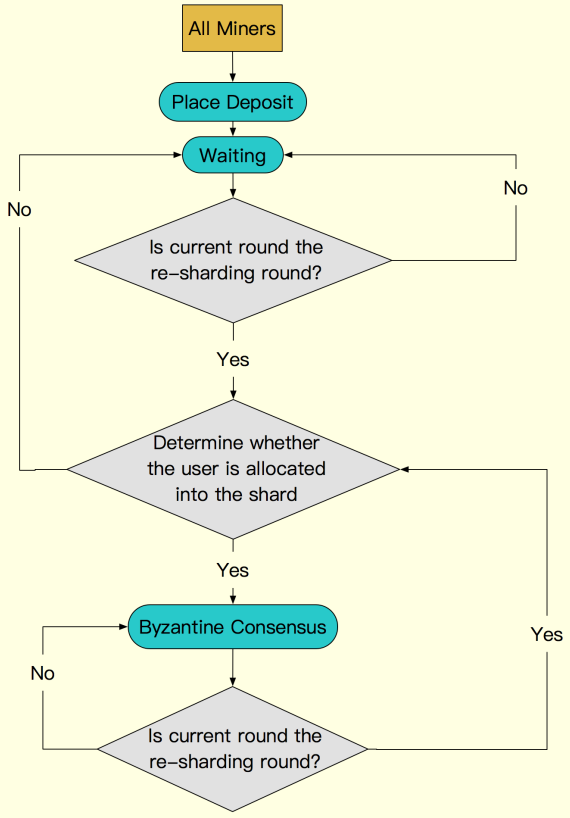


Figure 2: *Dynamic Miner Sharding Flow Diagram*

MultiVAC is a Proof of Stake system. In a Proof of Stake system, miners have to prove ownership of a

certain amount of tokens (stake) to be able to participate in consensus. For more details on why we choose a Proof of Stake rather than a Proof of Work system, please refer to Appendix 1. In MultiVAC, we enforce Proof-of-Stake by requiring each miner to lock up a deposit before a shard can be joined.

After locking in the deposit, the miner waits until one of the shards initiates re-sharding, which takes a few minutes. MultiVAC runs shards in an asynchronous manner, so miners may be selected for multiple shards proportional to their deposit size. The miner runs his VRF on his private key and the shard's VRF seed to determine whether or not he can enter the shard. A miner node that has not been selected for inclusion by a shard waits until it is included.

When shards are overburdened, MultiVAC supports incremental shard splitting to increase the number of shards. When a shard splitting is triggered, an overburdened shard is split into two shards. Light nodes belonging to the original shard are allocated to the new shard according to their addresses, and storage nodes are given some leeway to choose between the old and new shards. The new shards then are both allocated with miners by the re-sharding algorithm.

2.3 Transaction Confirmation, Consensus, and Storage

In MultiVAC, storage nodes store all transactions. MultiVAC adapts a variation of the UTXO model used in Bitcoin in which the input of every transaction is the output of a previously confirmed transaction. The system traces outputs and their states (whether or not the transaction has already been spent) to prevent double spending. MultiVAC stores outputs and their states in the Merkle Tree data structure.

When a new transaction is produced and signed by a light node, it is submitted to the storage nodes in the corresponding shard. The storage node will broadcast the transactions to all the shard's current miners, and enabling them to receive rewards. The blocks thus reach all the shard's current miners.

Each shard runs a fast Byzantine Consensus algorithm to reach blockchain consensus. Miners are selected for different consensus tasks by the VRF, the first of which is block proposal. All miners selected for block proposal include their pending transactions on the new block. The block is further processed and voted on by miners selected by the VRF until consensus is reached.

After achieving consensus on the block, it is broadcast to all storage nodes in the shard, which is stored on their hard disks. The corresponding block header is broadcasted to all miners in the network but is kept outside of the shard for space efficiency. At this point the transaction is confirmed.

The whole process is briefly illustrated in Figure 3.

2.4 Formal Definitions

We now provide definitions for the MultiVAC system in detail below.

2.4.1 Transactions

- **Transaction**: A message submitted from one client to another representing a movement of tokens from one account to another account. Each transaction is a block of data that includes a number of inputs to spend, a number of outputs to produce, and the id of the transaction type.
- **Input**: An input is a gained unit of money (a output) which must be referenced to be spent. Inputs contain a reference to the output of a previous transaction and a script signature unlocking the input.
- **Output**: Outputs are produced by transactions occurring in MultiVAC. They contain a number denoting the value of the output and a public key script corresponding to the receiver's address.
- **Output State**: An output state is a binary (0-1) flag which determines if the output has been already used as an input to a transaction.
- **Transaction Type**: The transaction type is a code which may be one of three types: normal transaction, deposit transaction or withdraw transaction.
  - **N, Normal Transaction**: a regular transaction of funds from one account to another account.
  - **D, Deposit Transaction**: a transaction used to place a deposit. The payer and payee must be the same account, and the token included in the deposit transaction can only be used in a withdraw transaction.
  - **W, Withdraw Transaction**: a transaction withdrawing the deposit and unlocking it for future use.
- **Output Message**: The output message is a message submitted from one shard's storage nodes to another which allows the second shard to reconstruct outputs deposited to the second shard by the first shard.

2.4.2 Nodes and Clients

- **Miner Node** $\mathscr{M}$: Also known as **users**, miner nodes run MultiVAC's Byzantine consensus and receive rewards when selected as block proposer. Any user running MultiVAC software is eligible to be selected as miner as long as he or she first locks up a stake deposit. Miners are required to listen to all messages from their own shard and block headers from all shards. Miners keep track of block headers, Merkle roots, and some partial Merkle tree structure from all shards in order to confirm the validity of pending transactions. Finally, miner

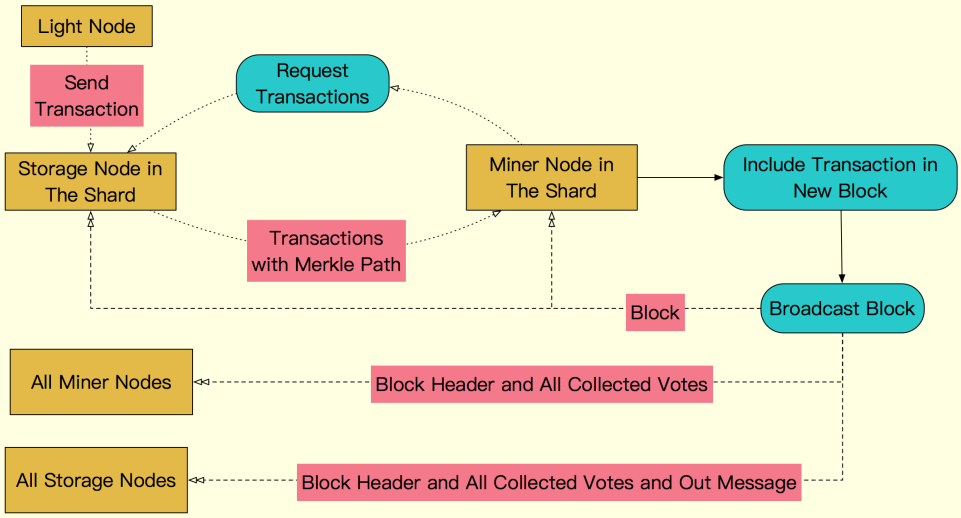


Figure 3: *Block Generation Flow Diagram*

nodes maintain a pending transaction pool to aggregate transactions for potential block proposal. The hardware requirements to be a miner node are quite low and any modern personal computer would suffice. $\mathscr{M}$ denotes the set of all miner nodes.

- **Storage Node** $\mathscr{S}$: Storage nodes are assigned to particular shards and are responsible for storing all of that shard's historical transactions. Storage nodes are designed to respond to and serve requests from miner nodes and are incentivized for providing storage; they have higher hardware requirements than miner nodes. Storage nodes are only providers of services to the network and do not participate in consensus and thus do not have decisions making "power" in the network; in particular they do not pose a centralization risk. $\mathscr{S}$ denotes the set of all storage nodes.
- **Light Node** $\mathscr{C}$: Light nodes or **clients** are nodes which submit new transactions. They are not required to store any block information except records of their own transactions. Light nodes have minimum hardware requirements, allowing a common mobile device to serve as a light node. $\mathscr{C}$ denotes the set of all light nodes.

2.4.3 Data Storage

- **Merkle Tree**: A Merkle tree is a binary tree shaped data structure for data storage. Every leaf node represents a data block and every non-leaf node up to the root is the cryptographic hash of its child nodes. This hierarchy of hashes allows for extremely fast and secure verification of the contents stored in the Merkle Tree, for example, the data of transactions representing a certain amount of monetary value.
- **Merkle Root**: A Merkle root is the root (top node) of a Merkle tree. Users with the Merkle root of a Merkle tree can quickly verify the existence of any content in the Merkle tree by checking the content's Merkle path.
- **Merkle Path**: A Merkle path is a path from a leaf of the Merkle tree to its root node, allowing for secure verification of the leaf node's content.
- **Main Merkle Tree**: The Main Merkle Tree is the name of MultiVAC's primary Merkle Tree data structure used to store transactions in storage nodes. The leaves of the Main Merkle Tree comprise all historical outputs and their output states aggregated from all the shards.
- **Block Merkle Tree**: A Block Merkle Tree is a Merkle tree in each block that stores the transactions outputs recorded in that block. The Block Merkle Trees are appended to the leaves of the Main Merkle Tree as blocks get confirmed.
- **Top Main Merkle Tree**: The Top Main Merkle Tree is the top section of the Main Merkle Tree, excluding the leaves which are Block Merkle Trees. The leaves of the Top Main Merkle Tree are instead the Merkle Roots of the corresponding Block Merkle Trees.

3 MultiVAC Sharding in Detail

Sharding is central to the design of MultiVAC. Using a strategy similar to distributed computing used by all major modern corporations to process huge volumes of data, MultiVAC separates nodes into network fragments called shards in order to perform parallel processing of transactions. This allows the network to scale and handle the high-volume business computations required by the modern economy. As a fundamental and integral part of MultiVAC, sharding is run automatically by the MultiVAC protocol. New shards are allocated if

the network burden on any existing shard becomes too great, and miner and storage nodes assigned to a new shard can immediately start constructing new blocks.

A **shard** $S_{i,r}$ is a subset of the MultiVAC network,

$$(\mathbb{C}_{i,r}, (B_{i,h})_{1\leq h\leq r}, (H_{i,h})_{1\leq h\leq r}, MT_{i,r}, \mathscr{M}_{i,r}, \mathscr{C}_{i,r}, \mathscr{S}_{i,r}),$$

where i is the shard index, r is the current height of the shard's internal blockchain, $\mathbb{C}_{i,r}$ is the consensus algorithm running within the shard, $B_{i,h}$ is the block of shard i at height h, $H_{i,h}$ is the header of $B_{i,h}$, $MT_{i,r}$ is the current Main Merkle Tree maintained by the shard, and $\mathscr{M}_{i,r} \subseteq \mathscr{M}$, $\mathscr{C}_{i,r} \subseteq \mathscr{C}$ and $\mathscr{S}_{i,r} \subseteq \mathscr{S}$ are the current subsets of miner nodes, client nodes and storage nodes belonging to the shard, respectively.

Each shard maintains a separate and distinct in-shard blockchain comprising a log of transactions. Clients are separated into different shards for service based on their public keys, with transactions assigned to the shard of the payer's public key. Each shard is periodically reassigned a group of randomly selected miners responsible for generating new blocks and running Byzantine consensus. Each shard also has a number of storage nodes which maintain records of all blocks generated by the shard and update all of the shard's assigned output states.

3.1 Miner Selection

MultiVAC is a shard-based blockchain system where miners perform consensus as part of network fragments called shards. The central question here is the selection process of the miners who would perform such tasks. It is apparent that a fair selection mechanism must be random and not biased towards any subset of miners over others. A naive random selection method is not however sufficient for the task. In the absence of a robust selection mechanism, malicious miners can perform Sybil attacks by passing off as multiple users so as to dominate the network. A classic random selection mechanism that prevents Sybil attacks is *Proof of Work* (PoW), which require miners to solve computational puzzles in order to contribute to consensus. However, as discussed many times in the blockchain literature, Proof of Work is an extremely energy-inefficient and wasteful mechanism.

MultiVAC instead uses a *Proof of Stake* (PoS) based algorithm which avoids wasteful energy expenditures while ensuring an equally high level of security. Each miner has certain amount of MultiVAC tokens that he or she can lock up in deposit which would correspond to his or her current stake. Together with a random seed produced by the shard and a selection likeliness proportional to the stake, a miner calculates a random number with the VRF to determine whether or not he is allocated to a shard. Miners may be selected for inclusion in more than one shard.

Miner Deposits MultiVAC requires each miner to place a deposit to be a potential candidate for shard inclusion. To place a deposit, a miner submits a special deposit transaction to a storage node and waits until the transaction is confirmed in a block. The deposit transaction has a special output, the deposit output, which cannot be spent by the user until it is unlocked by a withdraw transaction.

The deposit transaction consists of the following:

- Input: one or several outputs belonging to the user;
- Output: an output from the user to himself;
- Transaction Type: "D" for deposit transaction.

After the deposit transaction is confirmed, it is recorded in the block header. The miner broadcasts a heartbeat message to the whole network to signal that it is live, and is then eligible for selection into a shard. The miner then waits for a shard to perform re-sharding, the process of which is described below, and checks if it has been allocated to a shard by the VRF. Upon allocation, other miners can verify that the miner by verifying his random selection number as well as his deposit-confirmation message.

The miner's heartbeat message is valid for a period of around 24 hours, to avoid the allocation of inactive miners. After the heartbeat expires, the miner is required to re-send another heartbeat.

Withdraw Deposit The output value of a deposit is locked and not spendable as an input of a further transaction. If deposits were allowed to be spent, a Sybil attacker may stake a deposit to send a confirmation-message but then transfer the deposit to one of the fake Sybil accounts. This account can then use the money to stake another deposit and transfer it to yet another account, resulting in a large number of deposits backed by nothing at all. We thus require a special withdraw transaction to unlock a deposit.

3.2 Re-sharding

Suppose N is the total number of miners in the network and c is a pre-determined honesty threshold such that as long as the number of honest miners $N_h \geq cN$, the outcome of consensus is trustworthy. Define $P_t(N_h \geq cN)$ to be the probability that the number of honest nodes is always greater than cN after the network is running for a period of time t. The security assumptions of traditional blockchains depend on the fact that, over a large time period t, N is sufficiently great such that $P_t(N_h \geq cN) \geq 1 - p$ for negligibly small p.

Suppose N_i is the total number of miners in shard i, and $N_{h,i}$ is the number of honest miners in the same shard. It can be seen that $P_t(N_h \geq cN) \geq P_t(N_{h,i} \geq cN_i)$ because N_i is significantly lesser than N. In order to maintain the same security level, we need to restrict $t' < t$ such that $P_{t'}(N_{h,i} \geq cN_i) \geq 1 - p$.

In order to achieve this, we perform a re-allocation of miners to shards at intervals of time t' (for example, every a few minutes) in order to eliminate the probability that any particular shard is compromised. As such, the threat of compromising a shard is negligible as it its quite impossible to take control of a super majority of a shard containing several hundred random nodes in several minutes and carry out double spending transactions. The most damage an attacker can inflict on a shard is to prevent it from processing blocks in the current round. Even then, the damage will be resolved when the miners are re-sharded.

Verifiable Random Functions (VRF) Our re-sharding mechanism is based on Verifiable Random Functions (VRF), a pseudo-random number generator first introduced by Micali, Rabin and Vadhan in [16] and improved by Dodis and Yampolskiy in [17]. A VRF's output can be verified without further communication with the generator, making the VRF an ideal tool for random selection. Dfinity [18], Algorand [19], and Ouroboros Paros [20] all implement VRF as a random generator as part of their consensus schemes. MultiVAC adopts the VRF construction [21], where a detailed analysis of the VRF's security and pseudo-randomness is provided.

Let $a : \mathbb{N} \to \mathbb{N} \cup \{*\}$, $b : \mathbb{N} \to \mathbb{N}$, and $s : \mathbb{N} \to \mathbb{N}$ be three polynomial-time functions. A VRF with input length $a(\lambda)$, output length $b(\lambda)$, and security level $s(\lambda)$ is a suite of three polynomial-time algorithms (F_{GEN}, VRF, F_{VER}) such that

- F_{GEN} is the *key generation function*, a probabilistic function which takes in a unary string with length λ and which outputs a public key PK and private key SK.

$$F_{GEN}(1^\lambda) = (PK, SK).$$

- VRF is the *main random number generator*, a deterministic function which receives a private key SK and a seed x and which outputs two binary strings: the generated random value δ and the verification proof π.

$$VRF(SK, x) = (\delta(SK, x), \pi(SK, x)).$$

- F_{VER} is the *randomness verification function*, a probabilistic function that verifies the value of δ based on π and the public information.

$$F_{VER}(PK, x, \delta, \pi) = True \text{ or } False.$$

A VRF must have the following properties:

1. Correctness: the following two conditions hold with probability $1 - 2^{-\Omega(k)}$:
   - Domain Range Correctness: for any $x \in \{0,1\}^{a(\lambda)}$, $\delta(SK, x) \in \{0,1\}^{b(\lambda)}$.
   - Complete Provability: for any $x \in \{0,1\}^{a(\lambda)}$, $P(F_{VER}(PK, x, \delta, \pi) = True) > 1 - 2^{-\Omega(\lambda)}$ if $VRF(SK, x) = (\delta, \pi)$.
2. Unique Provability: For any $PK, x, \delta_1, \delta_2, \pi_1, \pi_2$ with $\delta_1 \neq \delta_2$, then

$$P(F_{VER}(PK, x, \delta_i, \pi_i) = True) < 2^{-\Omega(\lambda)},$$

   for any $i \in \{1, 2\}$.
3. Residual Pseudorandomness: Let $T = (T_E, T_J)$ be any pair of algorithms taking 1^λ as the input and taking execution count less than $s(\lambda)$ steps. Then for $* \neq x$, let

$$T_E^{VRF(SK,*)}(1^\lambda, PK) = (x, \hat{\pi}),$$

   where PK, SK are generated by F_{GEN}.
   Now, define a random variable X taking on two states with equal probability. Depending on the state of X, a value for $\hat{\delta}$ is determined either randomly or from $\delta(SK, x)$:
   - $P(X : \hat{\delta} = \delta(SK, x)) = 0.5$;
   - $P(X : \hat{\delta} \to^R \{0,1\}^{b(\lambda)}) = 0.5$.

   We require that no prediction algorithm T_J is able to accurately predict within the safety margin $s(\lambda)$ the actual state of X that generated δ:

$$P(T_J^{VRF(SK,*)}(1^\lambda, \hat{\delta}, \hat{\pi}) = x) \leq 0.5 + s(\lambda)^{-1}.$$

The Re-sharding Process All miners in MultiVAC receive block headers produced by all shards. The block header includes

- the current height of that shard's in-shard chain h
- the block's random seed Q^r

Miners determine when a shard is re-allocated by the current height of the in-shard chain. Currently we set the protocol to re-shard when the in-shard chain expands every n blocks or when the shard hits a timeout t_s after the shard's first allocation t_o, so miners will trigger a re-shard when either of the below is true:

$$\begin{cases} h \equiv 0 \pmod{n}, \\ t = t_s + t_o. \end{cases}$$

After the commencement of re-sharding, all miners that have placed a deposit may choose to participate in the new shard. The deposits are not counted in absolute token count, but in terms of deposit units, the size of which adjusts with the shards' level of overcrowding as well as the overall system's security needs.

If the deposit of a miner m is α_m token units, the miner generates a uniform random number $prob$ and a corresponding verification ver using VRF with their private key SK and the random seed Q_r:

$$prob, ver = VRF(SK, Q^r), \qquad 0 \leq prob < 1.$$

Now, given a security level $0 < s' < 1$ (discussed below), the probability that a miner is allocated into the shard is s' is $1-(1-s')^{\alpha_m}$, and the miner is selected if $prob < 1-(1-s')^{\alpha_m}$. Due to the pseudorandomness of the VRF, we have

$$|P(m \in \mathscr{M}_{i,r}) - (1-(1-s')^{\alpha_m})| < \epsilon,$$

for very small ϵ, equivalent to the statement that under negligible probability, the following statements hold:

- $P(m \in \mathscr{M}_{i,r}) > P(m' \in \mathscr{M}_{i,r})$ if $\alpha_m > \alpha_{m'}$;
- $P(m \in \mathscr{M}_{i,r}) < P(m' \in \mathscr{M}_{i,r})$ if $\alpha_m < \alpha_{m'}$;
- $P(m \in \mathscr{M}_{i,r}) = P(m \in \mathscr{M}_{i',r})$ for any i and i'.

That is, miners with the same deposit are equally likely to be chosen for a shard and miners with higher deposit have a higher likelihood to be chosen for a shard.

The Security Level s' In each shard the security level s' is a parameter that affects the number of allocated miners. A high s' gives miners a greater likelihood of allocation into the shard and causes the shard to have more miners serving it on average. This causes the shard to be both

- more secure, as it is less likely that an adversary can corrupt the majority of the shard to cause denial of service, and
- less efficient, as it is more likely that a greater amount of communication is required to reach in-shard consensus.

Security requirements for transactions and for monetary processing are fixed at a very high rate. A default setting for s' currently used is

$$s' = \frac{1}{\#\text{ all shards}}$$

with the same number of expected miners for each shard. In other applications, it can, however, be feasible for users to choose a range of s' levels under which they wish to run their applications, allowing them to obtain faster speed for less security-intensive applications. We call this technique flexible sharding. Flexible sharding would give users significantly more leverage in customizing their applications than most blockchains systems today, where usually speed and security parameters for all applications are rigidly fixed by the underlying blockchain infrastructure.

Security Analysis Suppose the number of honest miners in the whole network is $N_h = \rho N$ with honesty ratio ρ, and suppose further that after re-sharding, we require that $N_{i,h} \geq cN_i$ for some honesty threshold c. By the analysis from previous subsections, a miner is chosen by a shard with probability

$$p = 1-(1-s')^{\alpha_m},$$

where the security level is s' and the miner's deposit is α_m deposit units.

For simplicity, we assume that all the miners place the same amount of deposit. Let $B(n,p)$ be the binomial distribution with number of trials n and success probability p, and let $N(\mu,\sigma^2)$ be the normal distribution with mean μ and variance σ^2. We define two random variables

- $X \sim B(N-N_h,p) \approx N(p(N-N_h),(N-N_h)p(1-p))$, the number of malicious miners chosen by the shard;
- $Y \sim B(N_h,p) \approx N(pN_h, N_hp(1-p))$, the number of honest miners chosen by the shard.

We have that

$$P(N_{i,h} \geq cN_i) = P(Y \geq c(X+Y)) = P((1-c)Y - cX \geq 0),$$

and we define the random variable $W = (1-c)Y - cX$. $W \geq 0$ corresponds to a situation in which the number of honest nodes in the shard satisfies the honesty requirement. Because X and Y are independent, the distribution of W can be approximated as follows:

$$W \sim N(\mu,\sigma^2),$$

where

- $\mu = (1-c)pN_h - cp(N-N_h) = pN_h - cpN = p(\rho-c)N$,
- $\sigma^2 = (1-c)^2N_hp(1-p) + c^2(N-N_h)p(1-p) = \left[(1-c)^2\rho + c^2(1-\rho)\right]p(1-p)N$.

Let Z be the standard normal distribution $N(0,1)$. Then

$$\begin{aligned}
& P(W \geq 0) \\
= \;& P(\sigma Z + \mu \geq 0) = P(Z \leq \frac{\mu}{\sigma}) \\
= \;& \Phi\left(\frac{p(\rho-c)N}{\sqrt{\left[(1-c)^2\rho + c^2(1-\rho)\right]p(1-p)N}}\right) \\
= \;& \Phi\left(\frac{p(\rho-c)}{\sqrt{\left[(1-c)^2\rho + c^2(1-\rho)\right]p(1-p)}}\sqrt{N}\right) \\
= \;& \int_{-\infty}^{\frac{p(\rho-c)\sqrt{N}}{\sqrt{\left[(1-c)^2\rho + c^2(1-\rho)\right]p(1-p)}}} \frac{1}{\sqrt{2\pi}} e^{\frac{x^2}{2}}\,dx,
\end{aligned}$$

and hence,

$$\begin{aligned}
P(N_{i,h} \geq cN_i) &= P(W \geq 0) \\
&= 1 - P(W < 0) \\
&= 1 - \int_{-\infty}^{a} \frac{1}{\sqrt{2\pi}} e^{\frac{x^2}{2}}\,dx,
\end{aligned}$$

where $a = \frac{p(\rho-c)\sqrt{N}}{\sqrt{\left[(1-c)^2\rho + c^2(1-\rho)\right]p(1-p)}}$.

Assume all the miners place 10 units of deposit, the security level is $s' = 0.1$, the honesty ratio $\rho = 0.8$, and the honesty threshold is $c = 0.75$. Then we can

compute the probability $P(W \geq 0)$ in terms of N. The plot of $\log(N)$ against $log(1 - P(W \geq 0))$ is shown in Figure 4.

It can be seen that if $N \geq 1500$, the probability that the number of honest miners who are chosen into shard i is less than the threshold cN_i becomes less than 10^{-10}. For comparison, $N = 10424$ in Bitcoin and $N = 14383$ in Ethereum. Thus, the shards would fail to receive an honest threshold of miners with negligible probability. Even if attackers are extremely lucky and a shard does fail to obtain an honest threshold, attackers are still unable to falsify transactions and can only halt the production of blocks in the shard until the shard is re-allocated.

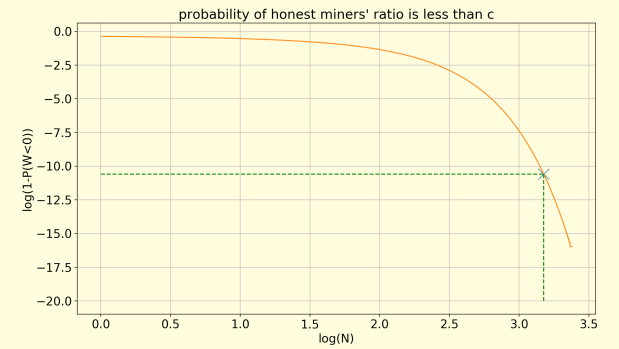


Figure 4: *Plot of* $\log(N)$ *against* $\log(1 - P(N_{i,h} \geq cN_i))$. *At* $N = 1500$ *(where* $\log(N) \approx 3.17$*), the likelihood that less than the threshold of honest miners is allocated into a shard is* 10^{-10}.

3.3 Shard Splitting

MultiVAC dynamically achieves scalability by using *shard-splitting* to increase the number of shards. If the network realizes that a particular shard consistently faces high transaction flow, the shard is split in two shards with each half serving half of the original shard's accounts.

Suppose the original shard is

$$S_{i,r} = (\mathbb{C}_{i,r}, (B_{i,h}), (H_{i,h}), MT_{i,r}, \mathscr{M}_{i,r}, \mathscr{C}_{i,r}, \mathscr{S}_{i,r}),$$

where $1 \leq h \leq r$. After splitting, the shard is separated into

$$split(S_{i,r}) = (S_{2i,r}, S_{2i+1,r}),$$

where

- $S_{2i,r} = (\mathbb{C}_{i,r}, (B_{i,h}), (H_{i,h}), MT_{i,r}, \mathscr{M}_{2i,r}, \mathscr{C}_{2i,r}, \mathscr{S}_{i,r})$;
- $S_{2i+1,r} = (\mathbb{C}_{i,r}, (B_{i,h}), (H_{i,h}), MT_{i,r}, \mathscr{M}_{2i+1,r}, \mathscr{C}_{2i+1,r}, \mathscr{S}_{i,r})$;

and

$$\mathscr{C}_{i,r} = \mathscr{C}_{2i,r} \sqcup \mathscr{C}_{2i+1,r}.$$

Shard splitting does not affect the in-shard consensus, the shard's blocks, block headers, Merkle Trees or storage nodes. In particular, storage nodes are required to work for both newly created shards for a short period to guarantee a smooth transition. Inside the storage node, the blockchain of the shard fork is able to serve both shards while avoiding duplicated storage. Provided that each new shard has enough storage nodes, storage nodes are allowed after a certain period to stop supporting one of the shards so as to not overburden them by serving multiple shards.

After shard splitting, miners $\mathscr{M}_{2i,r}$ and $\mathscr{M}_{2i+1,r}$ are re-allocated to both shards using re-sharding. Since accounts are assigned to shards based on their address, the original shard addresses will starte with 00. After splitting, the new shard will be responsible for addresses starting with 000 and 001 respectively. This is depicted in Figure 5.

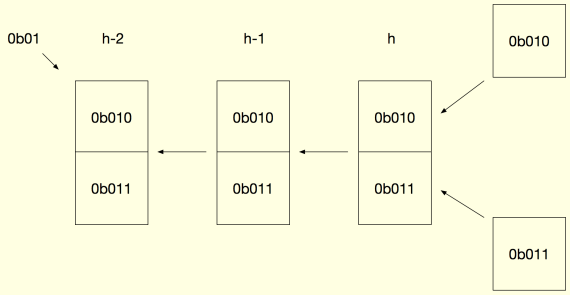


Figure 5: *Account Address-Based Shard-Splitting*

3.4 In-Shard Consensus

In MultiVAC's definition of a shard

$$S_{i,r} = (\mathbb{C}_{i,r}, (B_{i,h}), (H_{i,h}), MT_{i,r}, \mathscr{M}_{i,r}, \mathscr{C}_{i,r}, \mathscr{S}_{i,r}),$$

with $1 \leq h \leq r$, the consensus algorithm $\mathbb{C}_{i,r}$ is replaceable from shard to shard. Theoretically, any shard could run any consensus algorithm, including PoW, although this would not be practical. MultiVAC instead runs consensus algorithms from the Byzantine consensus family, which assume a quorum of 2/3 honest nodes to operate. Byzantine consensus algorithms have full finality, meaning every shard's blockchain is unique and cannot be forked. Examples of Byzantine consensus algorithms are pBFT [22], Tendermint [23], and BA* [19]. Research and development is underway to explore the Byzantine Consensus Algorithm(s) most suited to our speed and flexibility requirements.

Once allocated to a shard, miners are delegated via VRF to tasks comprising of block generation and consensus. The benefit of using a VRF is that not all miners need to participate in every consensus task, significantly reducing network traffic and speeding up consensus. It is precisely because the randomness used to assign miners cannot be manipulated and is verifiable, a high level of security is reached even without all miners participating in every step.

The in-shard miner selection process for consensus is as follows. We call the length of time between block verifications as *blocktime*. Every blocktime is broken up into a number of *tasks*, which are usually voting procedures which depend on the specific algorithm used. Suppose the current blocktime is r and the current task number is t. There exists a random seed Q^{r-1} in the block header of the previous blocktime, and each miner in the shard generates a random number with their private key

$$h = VRF(SK, hash(Q^{r-1}, r, t)).$$

A miner is selected as a voter if h is less than a predetermined threshold $p_{r,s}$.

An important difference exists between the above algorithm and the VRF used in shard-selection, namely, in the above algorithm, miners are not weighted. In other words, so long as a miner is allocated into a shard, he or she has an equal say in the vote, making it impossible for miners with high stake deposits to also dominate a shard with high voting power.

The first task for every blocktime is block proposal. At the beginning of the consensus, each miner selected for the block proposal task constructs a new block B_u and broadcasts the proposed block to the whole shard. In the block proposal task, the random number h generated during miner selection doubles as that miner's priority value, thus producing a total order over priority values for proposed blocks. Receivers of potential blocks cache all blocks and proceed to future tasks based on the priorities of the blocks.

4 MultiVAC Storage and Transmission Sharding

In this section we describe MultiVAC's sharded storage and transmission solution.

MultiVAC performs full sharding over the blockchain network, meaning that we not only split up the network computational load amongst miners but also divide up network storage amongst nodes. We design this in such a way that miners need only communicate with other miners in the same shard to obtain or commit a transaction, significantly reducing transmission costs and realizing sharded transmission. MultiVAC is the first major blockchain to create a fully-sharded, three-pronged computation, storage, and transmission solution, essential to scale blockchains to industrial capacity.

Indeed, storage load is a major concern on modern blockchains. A blockchain achieving high TPS (transactions per second) also creates a high storage load. If a blockchain achieves 2,000 TPS with a transaction size of 400 bytes, the monthly storage pressure is 1,931GB. It thus becomes impractical for a common node to store all the blocks. MultiVAC solves this issue through storage sharding. In MultiVAC, there is a division of labor between miner nodes and storage nodes. Miner nodes are responsible for generating blocks and have voting rights in the system, whereas storage nodes are solely responsible for storing and serving data and act as service providers to the network. MultiVAC inclines towards reducing the local computational load on miners as much as possible so that many ordinary computers can join the mining network, allowing decision-making power to remain with ordinary users and promoting the network's health and efficiency.

Our storage solution has the following features:

No full ledger in miner nodes. Suppose we have a blockchain with a Visa-average throughput of 2000 tps. If miners stored the full ledger, this would amount to 23T of data a year linearly increasing over time, a large amount for ordinary computers. MultiVAC has most of the data held by storage nodes so that miners need only hold less than 1G of data at any time.

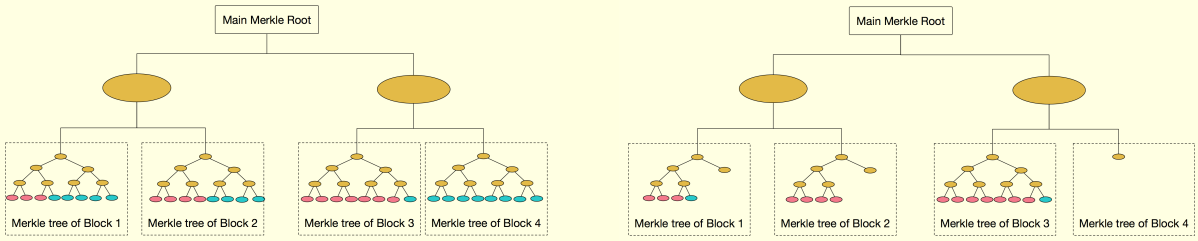
In-shard gossip protocol. In traditional blockchains like Bitcoin, messages are broadcast via gossip protocol to all the users in the network. In MultiVAC we implement a gossip protocol that makes it possible to broadcast a message only to all users within particular shards, allowing nodes to only receive messages in which they are interested.

Verifiable and trustworthy service by storage nodes. Though most of the data in MultiVAC is stored by storage nodes to which miners send requests, the miners must also store enough information to verify that the data returned is accurate. The MultiVAC protocol is designed in such a way that all the information provided by storage nodes is verifiable by the miner and all unverified messages are discarded.

No extra network transmission for miners. To perform verification, miner nodes need to locally store the Merkle root of the main Merkle tree. In MultiVAC's design, the miners are capable of updating the Merkle root with only newly generated blocks, thus updating the main Merkle root without additional transmission pressure to the storage node, giving miners necessary information for in-shard block generation and re-sharding.

4.1 Storage Methodology

A Merkle Tree of Transaction Outputs MultiVAC's storage nodes store transaction output states in a Merkle Tree data structure. These are packaged into blocks, which contain outputs in a Merkle tree called the **block Merkle tree**. Block Merkle trees are compiled together into a large Merkle tree, the **main Merkle tree**, where all historical outputs are stored. Each storage node is responsible for maintaining the section of the main Merkle tree that contains the outputs covered by his shard, determined based on the receiver's account id. Merkle trees provide an efficient storage and communication solution to store transaction outputs and to allow other nodes to verify outputs by tracing Merkle Paths.



(a) *The Full Main Merkle Tree*

(b) *The Pruned Main Merkle Tree Stored by the Storage Nodes*

Figure 6: *A depiction of the Main Merkle Tree stored in storage nodes, where red nodes are outputs to shard 1; blue nodes are outputs to shard 2; and orange nodes are intermediate nodes in the Merkle tree. Figure (a) is the complete Merkle Tree; and Figure (b) is the main Merkle tree maintained by shard 1's storage nodes. Outputs that shard 1 are not responsible for are pruned along with unnecessary intermediate nodes.*

Division of Storage Labor In MultiVAC, miners are required to keep track of

- All block headers;
- Roots of all shards' main Merkle trees;
- Merkle Paths as required to perform updates to the Merkle Roots of all shards.

Storage nodes are required to be aware of all blocks, comprising all historical transactions. However, they only maintain and update transaction outputs whose accounts are in the shard they are responsible for. For this reason, storage nodes of different shards will have different main Merkle Trees because storage nodes are only responsible for updating the state of transactions in their shard. Once the state is updated, the block hash changes and is propagated up to the Merkle root, resulting in each storage node having recorded the same transactions but updating only the transactions that it is responsible for serving up. Storage nodes may then optionally choose to prune the records of transactions that it is not responsible for.

The Block Merkle Tree Every newly confirmed block contains a group of newly confirmed transactions, stored in a Merkle tree data structure called the block Merkle tree. The outputs of these transactions need to be recorded because they may be used as inputs to future transactions. For every block that goes through consensus, the miner that proposed it constructs the block Merkle tree and adds its Merkle root to the block's header. Inside of the block Merkle tree, each output is marked with a 0/1 state marking whether or not the output is spent. These outputs are sorted by the recipient's address and the transaction's hash such that transactions within the same shard are adjacent. Because of this strict ordering, the block Merkle tree can be deterministically generated given a list of transactions. The idea is summarized in Figure 7.

The Main Merkle Tree Storage nodes store all historical outputs in a Merkle tree called its main Merkle tree. The main Merkle tree consists of two parts, the

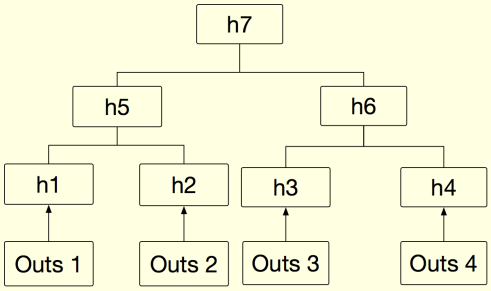
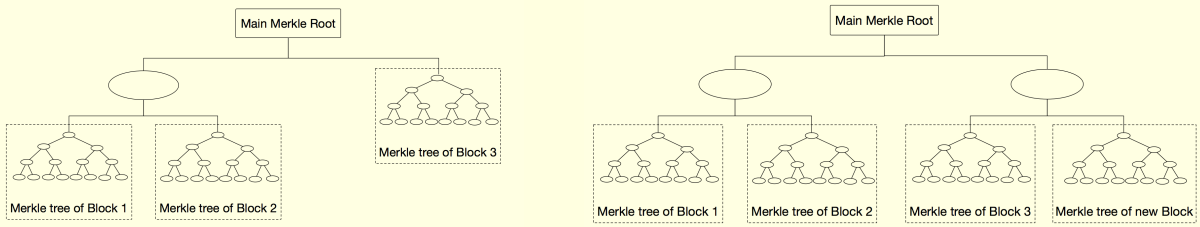


Figure 7: *A Block Merkle Tree. New outputs are labelled Outs 1-4.* $h_i = hash(Outs\ i, 0)$ *for* $i = 1, 2, 3, 4$*, and* $h5, h6$ *and* $h7$ *are calculated from their child hashes.*

top main Merkle tree and constituent block Merkle trees, such that the leaves of the top main Merkle tree are the roots of the block Merkle trees. Storage nodes append new block Merkle trees to the top main Merkle tree when new blocks are added. We illustrate this structure in Figure 6(a), assuming only two shards for simplicity.

Recall that an output consists of a value and a receiver's address. Storage nodes in Shard 1 are only interested in output addresses that are also in Shard 1. To improve storage efficiency the storage node is allowed to discard sub-trees containing transactions outputted to other shards so long as it still maintains those outputs' Merkle roots. This still allows the node to have access to every output that it is responsible for. This is illustrated in Figure 6(b).

When clients propose transactions, storage nodes forward to miners the Merkle paths of inputs to be spent, which are all outputs of previously confirmed transactions in a particular user's account. Miners maintain their own copy of the shard's Merkle root and verify the veracity of the forwarded transaction by checking its Merkle root against their local copy. Afterwards, miners follow the Merkle path and check whether or not the input has already been spent. If a malicious storage node provides false information to the miners, miners will reject the false information upon receiving a Merkle root that is different from their local copies.



(a) *Original main Merkle tree* **(b)** *New main Merkle tree*

Figure 8: *Here, we depict the effect of appending a block Merkle Tree to a main Merkle Tree which is not full. Figure (a) is the original main Merkle tree, which becomes Figure (b) after appending a the block Merkle tree.*

Thus, the only harm that a malicious storage node can do is to refuse to provide services, but since storage nodes are greatly duplicated to provide reliability of service, miners can always easily switch to another storage node.

4.2 The Block Confirmation Message

Miners need to keep track of the Merkle roots of all shards. Thus, miners need to be notified when new blocks in other shards are confirmed so as to update their local versions of that shard's Merkle root. This, however, is not sufficient information to perform the update; miners also need a way to prove that the information about the consensus transmitted to it is true. MultiVAC achieves this by broadcasting the votes of the final consensus task along with the newly confirmed block. Upon completion of the final consensus task, the miner generates a confirmation message containing all of the votes together with block header and broadcasts this message to the whole network. This message is called the **block confirmation message**. This confirmation message contains sufficient information for any miner to verify that the shard has indeed reached consensus, and the miner will update its own Merkle root with the obtained block header.

Negligible forgery risk. The possibility of forging a confirmation message by a malicious user is negligible, since votes included in the message must contain signatures and credentials from voting miners. A malicious user has to corrupt a majority of voting miners in the shard in order to create an artificial confirmation message.

Minimizing unnecessary network transmissions. A significant communication cost is incurred if all miners broadcast a confirmation message to the entire network. In MultiVAC's protocol miners only gossip one confirmation message and discard other messages if they have already gossiped.

4.3 Miners Update Requirements

We give a survey of miners' update requirements to maintain the block ledger.

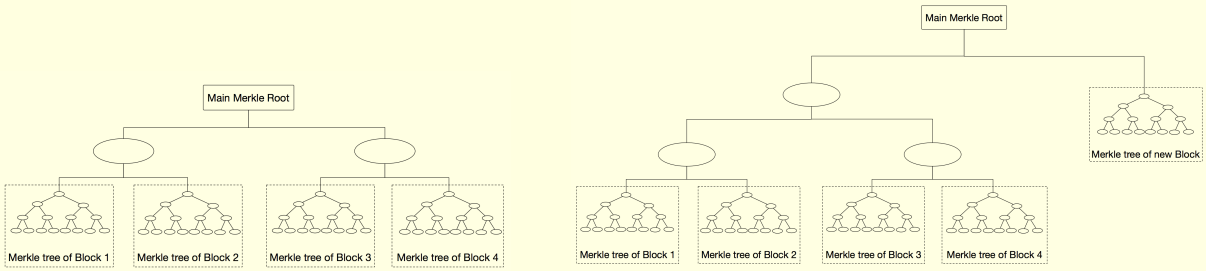
Recall that miners store not only all block headers but also the Merkle root from all shards and some Merkle paths from all shards as required for bookkeeping. When a miner is a block proposer, he is responsible for producing a block whose header contains two Merkle roots: the root of the attached block Merkle tree and the shard's new main Merkle root upon addition of the block. Miners must update state changes from newly spent transactions when calculating the new main Merkle root. In addition, Merkle roots of blocks received from other shards must be appended to miners' local versions of that shard's Merkle root and Merkle paths in order to stay consistent with the other shards. Finally, the miner's pending transaction pool consists of Merkle paths generated according to the current Merkle tree. After a new block is confirmed, these Merkle paths need to be updated in order for the system to stay consistent. These comprise a miner's update tasks in MultiVAC.

MultiVAC designs a process allowing miners to update their local state without knowledge of the whole Merkle tree. To do this, the miner requires the following information:

- The Merkle paths of all inputs spent in a miner's newly generated block.
- All roots of block Merkle trees included in block headers received from other shards since the miner's last update.
- The right-most Merkle path of each shard's top main Merkle tree, that is, the Merkle path marking the locations where new blocks should be added in each shard. These provided by the shard's storage node when the miner first joins the shard.

We present in more detail the different update tasks that a miner is required to perform below.

Updating Merkle Roots for a Newly Spent Transaction First, a block proposing miner must take all transactions involved in the block and accordingly update all their input states. This requires obtaining the hash value of a Merkle path at index i, a process summarized in Pseudocode 1.



(a) *Original main Merkle tree* **(b)** *New main Merkle tree*

Figure 9: *Here, we depict the effect of appending a block Merkle Tree to a full main Merkle Tree. Figure (a) is the original main Merkle tree, which becomes Figure (b) after appending a the block Merkle tree.*

Pseudocode 1: *getHash*

```
INPUT:
    P_t = ((H_0, S_0), (H_1, S_1), ..., (H_n, S_n)): Merkle
        path of a transaction t
    i: The location of the required hash
OUTPUT:
    H: A hash value in the merkle path
    S ∈ {L, R}: The side of the hash value
PROCEDURE:
    H ← H_i
    S ← S_i
```

The miner needs to change the states of spent inputs from 0 to 1 and calculate the new resultant Merkle root. It may update the Merkle root without knowledge of the entire Merkle tree as the miners will be provided with the corresponding Merkle paths for each input is changed. In Pseudocode 2 we provide the logic of updating the main Merkle root if only one input was changed. The procedure for updating the Merkle root if more than one input was changed can be easily derived.

Pseudocode 2: *updateRoot*

```
INPUT:
    t: Transaction
    P_t: Merkle path of the transaction t
OUTPUT:
    R: New Merkle Root
PROCEDURE:
    R ← hash(input, 1)
    for i in P_t; do
        H, S ← getHash(P_t, i)
        if S = L; do
            R ← hash(H, R)
        else; do
            R ← hash(R, H)
    Output R
```

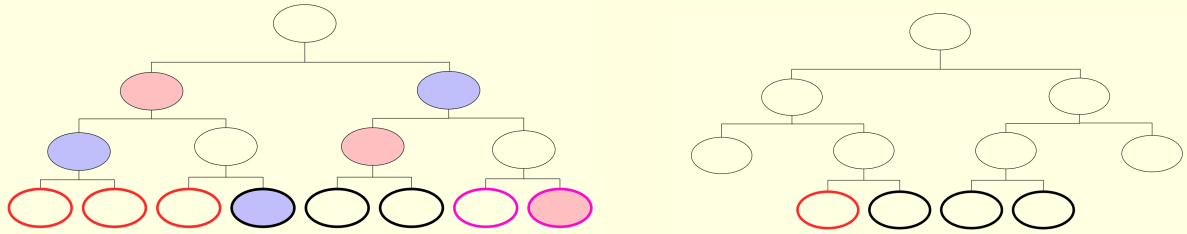
Incorporating Merkle Roots from New Blocks of Other Shards The miner has received a number of new blocks from other shards since its last update. When a miner is responsible for creating the next main Merkle root (i.e. the miner is one of the block proposer), he is responsible for including these blocks into the main Merkle tree so as to remain consistent with other shards. This is performed by appending the new blocks to the main Merkle tree before computing the Merkle root.

For all the pending blocks from shard i with height h_i, the miner sorted them as (i, h_i) lexicographically and append the blocks onto the main Merkle tree in such order. New heights of each shards will be recorded in the block and broadcasted within the shard. It is worth mentioning that the new height of each shard is greater than or equal to the height of the previous round, which is equal if the miner has not received a new block from that shard.

Different block proposers may see different pending blocks due to network asynchronization and thus generate different main Merkle root and different heights for other shards. However, only one of the new Merkle root will be confirmed by the consensus algorithm. Given that the main Merkle tree is completely determined by the new heights of the other shards, it gives sufficient information for other miners to confirm the computation of new main Merkle root and storage nodes to update the main Merkle tree.

We design a mechanism such that miners need only store the block merkle roots instead of the entire blocks in order to perform this operation. This solves a common problem in sharding solutions: as the network expands, cross-shard communication requirements become burdensome and hamper scalability. Our solution realizes transmission sharding by keeping the amount of cross-shard communication low, so that the transmission volume required for a single shard is close to fixed no matter how much the network expands. To do this, miners are required to maintain each shard's main Merkle Tree's rightmost Merkle path through which new blocks are added.

There are two cases for adding a new block. In MultiVAC, new block Merkle trees are always appended to the right-most branch of the main Merkle tree. If the Merkle tree is incomplete, the miner simply appends the new block Merkle tree to the next available node. This scenario is represented in Figure 8. Alternatively, if the main Merkle tree is already complete, the miner can increase the tree height by one and append the block. The second scenario is presented in Figure 9.



(a) *Full Merkle Tree of Output Messages*

(b) *Pruned Merkle Tree of Output Messages*

Figure 10: *In the figures above, red circles are outputs to shard 1, black circles are outputs to shard 2, and pink circles are outputs to shard 3. Suppose we have a storage node for shard 2 that is only interested in outputs to shard 2 only. Figure (a) is the full Merkle tree of output messages that the storage node receives, with the left-most path for shard 2 denoted by the blue nodes and the right-most path for shard 2 denoted by the red nodes. Figure (b) is the resulting Merkle Tree of output messages that shard 2 stores after pruning the outputs to other shards based on the left-most and right-most paths.*

These updates are made possible because the miner keeps track of each shard's right-most Merkle paths. This allows the miner responsible for proposing the next main Merkle root in its shard to also incorporate new blocks from the other shards.

Updating Internal Storage After In-Shard Block Confirmation Miners who receive a new confirmed block in their shard is able to update their main Merkle root and rightmost Merkle path with the information contained in the new block. In addition, they must update the Merkle paths of the pending transactions in their pending transaction pools, as they will have changed upon the update of the Merkle root. The logic for doing so is presented in Pseudocode 3:

Pseudocode 3: *updatePath*

```
INPUT:
    P_t^old: Merkle path of pending transaction
    P: Merkle path of a confirmed transaction
    𝒪: An output

OUTPUT:
    P_t^new: New Merkle path of the pending
        transaction
PROCEDURE:
    ℋ_old ← hash(𝒪, 0)
    ℋ_new ← hash(𝒪, 1)
    if ℋ_old in P_t^old; do
        P_t^new ← replace ℋ_old by ℋ_new in P_t^old
    for i in P; do
        ℋ_tmp, 𝕊_tmp ← getHash(P, i)
        if 𝕊_tmp = 𝕃; do
            ℋ_old ← hash(ℋ_tmp, ℋ_old)
            ℋ_new ← hash(ℋ_tmp, ℋ_new)
        else; do
            ℋ_old ← hash(ℋ_old, ℋ_tmp)
            ℋ_new ← hash(ℋ_new, ℋ_tmp)
        if ℋ_old in P_t^old; do
            P_t^new ← replace ℋ_old by ℋ_new in P_t^old
    Output P_t^new
```

4.4 Storage Node Update Requirements

We now give a survey of storage nodes' update requirements to maintain the states of their assigned shards.

Recall that storage nodes are required to maintain the states of all historical transactions sent to accounts in their shard. Since these transactions may have been generated by any shard, storage nodes have different procedures for updating in-shard and out-of-shard blocks.

When new blocks are confirmed in any shard, the miner that produced them creates and transmits an output message directed to all shards that must receive its outputs. Storage node listen to output messages from other shards in order to learn about new confirmed blocks and receive the outputs that they are responsible for. When a storage node receives a new block in its own shard, it processes all output messages and adds partial block Merkle trees holding the outputs from other shards to its main Merkle tree. It then finally appends the new block Merkle tree from its own shard to its main Merkle tree.

We give descriptions of these processes below.

The Output Message An output message from shard i to shard j is produced and sent every time a new block is produced in shard i that has outputs in shard j. The message consists of:

- The total number of outputs in the block;
- The content, index and Merkle path of the output prior (left of) the outputs to shard j;
- The content, index and Merkle path of the output subsequent to (right of) outputs to shard j;
- The list of outputs from shard i to shard j.

Output messages indicate to a storage node of shard j that a new block has been confirmed in shard i. With the list of outputs from shard i to shard j as well as the left and right Merkle paths, shard j is able to reconstruct the section of the block's Merkle Tree containing only the outputs to shard j.

Updating Out-of-Shard Confirmed Blocks When a storage node for shard j receives an output message from another shard i, it buffers the corresponding output message and does not immediately update its main Merkle tree.

The storage node only processes all output messages when a new block is confirmed in shard j, as described in the section below. When it does so, for each output message from i to j it constructs a partial block Merkle tree with the same Merkle root as shard i's block Merkle tree but only containing the outputs associated with shard j. The partial block Merkle tree is represented in Figure 10.

Updating In-Shard Confirmed Blocks New confirmed blocks in a storage node's shard contain the shard's new main Merkle root and information incorporating the other shards' blockchain heights determined by the block proposer.

After a new in-shard block is received, the current heights of other shards h_i are included. The storage node processes all buffered output messages from other shards at once, up to the height of the corresponding shard h_i, and reconstructs their partial block Merkle trees, appending each partial Merkle tree to the right end of its own main Merkle tree in lexicographic order. It then appends the block Merkle tree of the new block from its shard to its main Merkle tree.

Upon receiving a confirmed block in its own shard, the storage node updates its main Merkle tree, ensuring that the resultant Merkle root is the same as the one as included in the block's header. Since the network is asynchronous, it is possible that a storage node may be missing blocks from other shards during this process. Under those circumstances, the storage node requests the missing block and confirmation message directly from the corresponding shard after a short time delay of 5 seconds.

These sum up a storage node's update requirements to keep their respective shards up to date.

4.5 Storage and Transmission Summary

Here, we summarize our storage and transmission solution. MultiVAC realizes storage and transmission sharding by utilizing a division of labor between miners and storage nodes. Miners are responsible for achieving consensus on blocks and processing transactions and storage nodes are responsible for storing all historical data and serving up transactions. To perform secure and verifiable information sharing, miners must store all block headers, the roots of the main Merkle trees in all shards, as well as the right-most Merkle paths of all shards. This allows them to keep summaries of the blockchain state in all shards and process transactions consistent with such summaries, while keeping their hardware storage requirements at an incredibly low level. Storage nodes serve particular shards and store the blockchain's state of the shard allowing for duplication and availability. The design of this storage solution allows the vast majority of communications in the blockchain to be conducted within the shard, which we call transmission sharding. We further design mechanisms to relieve the network pressure on any parts of the network at any point in time, allowing for extremely fast and scalable protocol operation.

5 Summary

Traditional blockchain systems suffer from scalability issues that hamper their usability in the real-world context. Among the various architectures proposed to scale blockchains, the most promising thus far is, blockchain sharding, which still carries inherent limitations. Current sharding solutions have produced progress in parallelizing processing but are still limited in speeding up other aspects of computation, namely transmission and storage. To design a blockchain that is able to serve the real economy, MultiVAC is the world's first architecture that designs sharding for all aspects of blockchain computation: processing, transmission and storage.

MultiVAC uses Verifiable Random Functions to dynamically parallelize the network of miners into network fragments called shards. Miners are re-allocated into shards in an equitable and publicly verifiable way that is also safe against Sybil attacks. MultiVAC's elegant engineering solution involves a division of labor that realizes sharded transmission and storage through, ensuring that no part of the computational process is limited by a single bottleneck and that ordinary miners always have decision-making power in the blockchain. MultiVAC's architecture allows for linearly scalable throughput while staying close to blockchain's core values of decentralization and security. With our elegant all-dimensional shard architecture, MultiVAC is the world's first blockchain able to practically and robustly handle industrial-level demand.

References

[1] Nakamoto, Satoshi, *Bitcoin: A Peer-To-Peer Electronic Cash System.* (2008)

[2] Wood, Gavin. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* (2014).

[3] Larimer, Daniel et al. *EOS.IO Technical White Paper v2.* (2017)

[4] The Internet of Services Foundation. *Iost: The Next-Generation, Secure, Highly Scalable Ecosystem For Online Services.* (2017). https://iost.io/iost-whitepaper/

[5] The QuarkChain Foundation. *QuarkChain: A High-Capacity Peer-to-Peer Transactional System* (2018)

[6] Kwon, Jae and Ethan Buchman. *Cosmos: A Network of Distributed Ledgers* (2018) https://cosmos.network/cosmos-whitepaper.pdf

[7] The AElf Foundation *AElf - A Multi-Chain Parallel Computeing Blockchain Framework* (2017)

[8] Poon, Joseph, and Vitalik Buterin. *Plasma: Scalable Autonomous Smart Contracts.* (2017)

[9] Poon, Joseph, and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.* (2016)

[10] Popov, Serguei *The Tangle.* (2018). https://iota.readme.io/docs/whitepaper

[11] Baird, Leemon, Mance Harmon, and Paul Madsen. *Hedera: A Governing Council & Public Hashgraph Network.* (2018)

[12] Liu, Chunming, Daniel Wang, and Ming Wu. *Vite: A High Performance Asynchronous Decentralized Application Platform* (2018). https://www.vite.org/whitepaper/vite_en.pdf

[13] Li, Chenxing, Peilun Li, Wei Xu, Fan Long, and Andrew Chi-chih Yao. *Scaling Nakamoto Consensus to Thousands of Transactions per Second.* (2018)

[14] The Zilliqa Team. *The Zilliqa Technical Whitepaper.* (2017), https://docs.zilliqa.com/whitepaper.pdf

[15] The Ethereum Foundation. *Sharding Roadmap.* https://github.com/ethereum/wiki/wiki/Sharding-roadmap

[16] Micali, Silvio, Michael Rabin, and Salil Vadhan. *Verifiable Random Functions.* In Foundations of Computer Science, 1999. 40th Annual Symposium on, pp. 120-130. IEEE, 1999.

[17] Dodis, Yevgeniy, and Aleksandr Yampolskiy. *A Verifiable Random Function with Short Proofs and Keys.* In International Workshop on Public Key Cryptography, pp. 416-431. Springer, Berlin, Heidelberg, 2005.

[18] Hanke, Timo, Mahnush Movahedi, and Dominic Williams. *DFINITY Technology Overview Series, Consensus System.* (2018)

[19] Gilad, Yossi, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. *Algorand: Scaling Byzantine Agreements For Cryptocurrencies.* (2017)

[20] David, Bernardo Machado, Peter Gazi, Aggelos Kiayias, and Alexander Russell. *Ouroboros Praos: An Adaptively-Secure, Semi-Synchronous Proof-Of-Stake Protocol.* IACR Cryptology ePrint Archive 2017 (2017): 573

[21] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang *High-speed high-security signatures* Journal of Cryptographic Engineering 2 (2012), 77–89. https://cr.yp.to/papers.htmled25519

[22] Castro, Miguel, and Barbara Liskov. *Practical Byzantine Fault Tolerance.* In OSDI, vol. 99, pp. 173-186. 1999.

[23] Buchman, Ethan, Jae Kwon, and Zarko Milosevic. *The Latest Gossip on BFT Consensus.* (2018)

[24] de Vries, Alex. *Bitcoin's Growing Energy Problem.* Joule 2, no. 5 (2018): 801-805.

[25] Back, Adam. *Hashcash-A Denial of Service Counter-Measure.* (2002)

Appendix 1: Proof of Work and Proof of Stake

Proof of Work and Proof of Stake are the two best competing systems for achieving consensus in blockchains. Bitcoin famously uses the **Proof of Work** algorithm popularized by Satoshi Nakamoto, requiring miners to solve random computational puzzles requiring on average fixed amounts of time before they can contribute to consensus. PoW secures the network but requires the nonstop operation of tens of thousands of computations equivalent to continuously guessing random numbers. This requirement is extremely energy-inefficient and wasteful, causing the Bitcoin network to consume as much electricity as the entire country of Ireland in 2017. [24] Because of this huge expenditure of computational power, Proof of Work is not able to scale to the level that MultiVAC desires.

In reality, the purpose of the Proof of Work algorithm is not in itself able to produce consensus, and neither is its use essential to doing so. The function of Proof of Work is to certify identities, thus protecting the network against Sybil attacks or fake-identity attacks, when malicious actors pass off as multiple users so as to overwhelm the network. If node selection for bookkeeping is simply randomly chosen based on accounts, malicious users can easily simulate thousands of accounts in a virtual machine and overwhelm the network. PoW forces miners to provide proof of a costly limited resource, namely computational power, before being eligible to engaging in consensus, rendering it basically useless to create fake identities in exchange for mining power. In this sense the use of Proof of Work in Bitcoin is similar to earlier uses of Proof of Work in HashCash to certify an email against spam [25].

PoW provides one other important functionality: verifiable randomness. It both establishes identities and also selects miners in a random fashion whose fairness can be validated by all the other miners. When a Bitcoin miner succeeds in landing a PoW solution, what he or she performs is both a proof of his or her identity though the expenditure of computational power and also a proof of winning the node selection lottery by providing the random puzzle solution. This mechanism is based on the assumption that it very hard for dishonest users to own a controlling stake in the majority of the network's processing power, which is required to solve the computational puzzles.

For this reason, MultiVAC uses a **Proof of Stake** (PoS) selection system which avoids PoW's wasteful energy requirements but ensures an equal level of security. An alternative to Proof of Work, PoS is a system where where miners are randomly selected according to their amount of money (stake) in the system. As it is very hard to accumulate a controlling stake of the money in a network, providing proof of ownership of a certain amount of money also performs the same function of identity establishment as Proof of Work. In MultiVAC, we prove miners' identity by forcing them to lock up a stake deposit before they are able to start mining and earn mining rewards. Proof of Stake is secure because money ownership is required to be selected for bookkeeping, and it is very hard for dishonest users to own a controlling stake in the total amount of money in the network. After Proof of Stake identities are established, we use the mechanism of VRF to select bookkeepers in a random and verifiable way, fulfilling both functions of Proof of Work.