Ostraka: Secure Blockchain Scaling by Node Sharding

Alex Manuskin
Technion and IC3

Michael Mirkin
Technion and IC3

Ittay Eyal
Technion and IC3

Abstract

Cryptocurrencies, which promise to become a global means of money transactions, are typically implemented with blockchain protocols. Blockchains utilize a variety of consensus algorithms, and their performance is advancing rapidly. However, a bottleneck remains: each node processes all transactions in the system.

We present Ostraka, a blockchain node architecture that scales linearly with the available resources. Ostraka shards (parallelizes) the nodes themselves, embracing the fact that major actors have the resources to deploy multi-server nodes. We show that, in common scenarios, previous sharding solutions have the same property, requiring most node operators resources to process almost all blockchain transactions, while reducing system security.

We prove that replacing a unified node with a sharded Ostraka node does not affect the security of the underlying consensus mechanism and that Ostraka does not expose additional vulnerabilities due to its sharding. We identify a partial denial-of-service attack that is exposed by previous sharding solutions.

We evaluate analytically and experimentally block propagation and processing in various settings. Ostraka achieves linear scaling when the network allows it, unlike previous systems that require costly coordination for transactions that affect multiple shards. In our experiments, Ostraka nodes reach a rate of nearly 400 000 transactions per second with 64 shards, opening the door to truly high-frequency blockchains.

1 Introduction

Payment systems are typically controlled by central entities – central banks control supply by printing money, banks transact among each other with a central national [50, 61] or international [59] mechanism, and credit card companies control their clients' transactions. This centralization of power implies limitations of the users' freedom [57], less resilience, and high fees [6].

Since their introduction with Bitcoin [52], cryptocurrencies brought the promise of a *decentralized* global payment system outside the control of a single entity. Such a system is potentially useful for payment among individuals, but also among banks, machines (IoT) and for micro-payments for social network interactions.

The requirements of such a global payment system are high. Credit card companies can handle over $50k$ transactions per second [63]. And if each of Facebook's 1.5 billion daily active users [34] clicks but 6 Likes a day, translating these to transactions implies about $100k$ per second. The demand is even higher for payments among IoT devices [21].

So far, blockchains systems – the decentralized mechanism on which cryptocurrencies operate – could not reach such rates. Early implementations of blockchains suffered from a consensus bottleneck, limiting them to less than a dozen transactions per second [48, 64, 67] and they cannot scale by tuning their parameters [24].

More recent protocols [2, 18, 33, 38, 43, 44, 56, 58, 65] overcome the consensus bottleneck, but they are limited by the processing capacity of the single node, as each node processes all transactions in the system. We provide relevant background in §2.

Several recent solutions [3, 45, 49, 66] propose to *shard* the blockchain, splitting into into multiple interleaved sub-chains. Each sub-chain is maintained by a distinct set of nodes, and so each node does not have to process all transactions in the system but only those in its sub-chain. This allows the system to process transactions at a high rate while limiting the load on individual machines.

Those solutions target *democratic environments*, where there are many participants with similar resources. In such environments, increasing the number of participants allows to increase the number of shards and thus system capacity. However, we show that when resource distribution is not democratic, as is the case in most active cryptocurrencies, the security and performance of those systems deteriorate.

The sub-chains solutions assume that the number of participating nodes in each shard is sufficiently high (hundreds) [45], as a shard with few nodes could be corrupted by a malicious actor. Sybil-attack [29] protection mechanism, such as *Proof-of-Work* must be in place to prevent an adversary from achieving a majority and corrupting the system. To obtain a right for participation, actors, known as *miners* prove their possession of *mining power*, by solving a cryptographic puzzle. Therefore, it is possible to allot *shares* [44] small amounts of mining power, such that each share is granted to either a small miner or a part of a large miner. So, either the security of the system is reduced due to a small number of actors active in each chain, or, alternatively, non-negligible actors must in fact process most sub-chains.

Meanwhile, users with high stakes in the system (larger players like exchanges and payment processors) wish to validate their transactions. This, in general, implies processing all shards, since the history of each transaction might include transactions from all shards.

Additionally, we show that those solutions are vulnerable to full or partial denial of service attacks: an attacker can generate transactions that all end up in the same sub-chain, affecting the shard and transactions dependent on it. This blocks progress while the system actually has available resources. We discuss sharding solutions and other related work in §3.

In this work, we present Ostraka, a novel architecture that allows for arbitrary scaling of blockchain nodes up to the network capacity limit. Rather than breaking the system's state into different parts that are each managed separately, in Ostraka each node processes all transactions using horizontal scaling. We employ a direct shard-to-shard communication pattern that may be of independent interest. We detail Ostraka's architecture in §4.

The implication is that nodes require more resources, so fewer entities will run such nodes (e.g hundreds instead of thousands currently in Bitcoin [22] and Ethereum [32]). As we show, this is also the case with existing sharding solutions in non-democratic environments. The principals who benefit most from running a node are those with high stakes in the system such as payment processors or exchanges. For such principals, the excess node resources necessary for Ostraka are acceptable. Crucially, the decrease in node count does not mean that the system becomes closed: Anyone with sufficient resources can still join the system and participate in the protocol. Meanwhile, as with current cryptocurrencies, end-users who send and receive payments can still use their mobile devices by directly querying a few nodes. In summary, at the cost of reducing the number of entities that can participate, Ostraka enables a decentralized system with sufficient node capacity to become a global payment infrastructure.

Ostraka can be integrated into existing blockchain protocols, and since only the node architecture is different, security guarantees of the protocol remain (§5). We first show that the Ostraka distributed node is operationally indistinguishable from a unified-node. We define the node validation primitive based on existing state and incoming transactions and show indistinguishability of the results despite order changes and processing parallelization. Our protocol completes all requires stages for block validation, preventing double-spending or illegal minuting. Next, we show that an attacker can only perform a denial-of-service (DoS) attack against a small number of nodes, as the cost of generating transactions that would hit the same shard in many nodes grows exponentially with the number of nodes.

We evaluate Ostraka's block propagation in a variety of configurations of shards and networks. We show that an Ostraka node's processing rate grows linearly with the number of shards, and we provide theoretical analysis for the node performance as a function of inter- and intra-node bandwidth. We achieve performance of 99.5% from optimum speedup with high intra-shard network bandwidth. We measure the performance of Ostraka in transaction processing with as many as 64 shards, achieving a rate of nearly 400,000 transactions per second. We take an example of an existing blockchain protocol and show Ostraka enables an order of magnitude improvement in the blockchain protocol's throughput. Evaluation details are in §6.

In conclusion, we present a system for blockchain scaling, by sharding all aspects of the individual node, without compromises on security.

2 Preliminaries: The UTXO Model

We are interested in cryptocurrency protocols where the system allows users to pay one another with the system's currency, e.g., Bitcoin in the Bitcoin protocol, Ether in Ethereum protocol, etc. The system comprises nodes that operate the system and users whose money is stored and transacted in the system. The system maintains a state, which is the balance of each of the users. Each user can issue transactions that order payments to other users.

In blockchain systems, the nodes agree on the order of all transactions in the system by forming a series of blocks, each containing a list of transactions. Different systems use different protocols to form this agreement, from classical techniques [19, 46, 54] as in Hyperledger Fabric [18], through the Proof-of-Work [30, 40] mechanism introduced by Nakamoto [52], to contemporary Proof of Stake mechanisms [17, 38, 43].

Whichever mechanism is chosen, the system's state at any time is defined by the history of transactions up to that time. An important contribution of Nakamoto's paper is the data structure used to record the state called the UTXO model, which we use in Ostraka. We describe here the elements of the UTXO data structure relevant to this work. The interested reader is referred to [5, 12, 53] for further details.

The UTXO model In systems like bank records, the data structure maintains accounts, a record for each user and the amount held in this account. Instead, Nakamoto proposed to index by coin, so for each coin, the data structure records the owner of this coin. For efficiency, the records are not by whole coins, rather they contain arbitrary amounts and their owners. Thus, the balance of the user is the sum of amounts whose records indicate they belong to that user.

Each transaction signifies the movement of funds from one user to another. Therefore, it contains the information which records to destroy – that of the previous owner, and the new record to add – that of the new owner. The new records formed by the transaction are called its *outputs*, and the references to old records, which should be destroyed, are called its *inputs*. Obviously, the sum of coins in records destroyed should equal the sum of coins in the records generated; when a transaction references a previous transaction output to use its coins it is said to *spend* this output. The system's state is thus defined by the set of all transaction outputs that were not yet spent or the *Unspent Transaction Outputs* or the *UTXO Set*. The total amount of coin a user has is the sum of amounts recorded with her ownership. Each user keeps track of the coins in her possession, which she can use to create new transactions. We proceed to describe the structure of a transaction.

Transaction A transaction in the UTXO model comprises two lists – inputs and outputs. The outputs are the new records indicating ownership of amounts. The inputs are references to outputs of previous transactions, i.e., the records to be destroyed. Each transaction is uniquely identified by a cryptographic hash of its contents, called `TxHash`. An input of a transaction references the output it spends by the `TxHash` of the transaction that contains the output and its index in that transaction's output list.

Authorization In a banking system, the system is protected through an authentication mechanism and only an authenticated user can order a transaction out of her account. In an open system, since any node can add a transaction, the authorization is implemented with cryptographic primitives.

Transaction outputs contain, beyond the transferred amount, a *spending condition*, phrased as a script called `ScriptPubKey`. An output spending this input holds a *claiming condition* called `ScriptSig`. The most common example of an output spending condition is providing a signature. If Alice is to pay Bob, Bob first generates a private-public key pair and sends Alice his public key. Alice generates a transaction with an output whose script requires Bob's signature. Once this transaction is placed on the Blockchain, only Bob can move the money out of this output, therefore the payment is complete. In order for Bob to spend the output and make a payment (say, to Carol), he creates a new transaction referencing the output Alice had generated, and places in the corresponding input a signature with his private key.

An input is only valid if the `ScriptPubKey` in its referenced output evaluates to true with the input's `ScriptSig`. A transaction is only valid if all of its inputs are valid. A user's balance is, therefore, the sum of UTXOs for which the user can supply `ScriptSig` that would evaluate to true.

Data structures To operate a blockchain system, each of the nodes maintains three data structures. First, a set of transactions that were not yet placed in the blockchain called the mempool; secondly, the blockchain itself, with the series of all transactions that took place so far; and finally, the current system state comprising all unspent transaction outputs.

Users of the blockchain create transactions by broadcasting them to the system. These transactions are first aggregated in the nodes' mempools. Once a node constructs block from transactions in the mempool, it publishes the block to the peers, and the nodes store it.

Once a new block is created and broadcast, all the nodes participating in the protocol update their state accordingly. When the node receives a new block, it removes all outputs spent by transactions in the last block from the UTXO set and adds the newly created outputs. The state of the UTXO set is thus the state of the blockchain following each block.

Light clients The nodes are in charge of validating the blockchain's blocks, making sure their structure is legal and the transactions they contain are valid (e.g., spend existing outputs, do not create money, have correct signatures). Nodes do so by processing the entire blocks, however, users often interface with the system using mobile devices that do not have the capacity to process the entire blockchain. Such client therefore use a lightweight protocol called *Simplified Payment Verification* [52] (*SPV*). An SPV client stores only the header of each block, which contains metadata that allows it to perform basic validity checks. The header also contains a Merkle tree [51] digest of all transactions in the block. The Merkle tree allows a node to efficiently prove to a client that a certain transaction is present in the block. Thus, a user communicates with a few nodes and asks them for transactions relevant to it. If at least one of them is honest, the user can deduce which transactions she cares about are in the blockchain.

Security in *Proof-of-Work* Blockchains Unlike classical consensus systems, where the number of participants is known in advance and a majority is required to reach consensus, many blockchain systems are open for participants to leave and join at will. In this setup, it is difficult to determine a majority, as the number of participants is unknown. Instead, a lottery system is put in place to make sure participant cannot fake votes and mount a Sybil-Attack [29] by pretending to be many separate entities.

In *Proof-of-Work* (*PoW*) systems, participant compete for the right to participate in the consensus protocol by solving a cryptographic puzzle. To increase the chance of winning the game, one must invest resource in hardware, specialized for solving the puzzle. The chance to find the solution can only increase by accumulating resources and not identities. The effort a miner invests into solving the puzzle is referred to as *mining power*. Similarly, in *Proof-of-Stake* (*PoS*), right to participate is determined by the amount of funds (e.g. coins) one poses.

In a *PoW* blockchain, block propagation time has a direct effect on system security. The Nakamoto consensus states that the most up-to-date state of the blockchain is the chain with the most work, sometimes called the longest chain.

When a miner finds a block solving the puzzle, she broadcasts it to the system. If block propagating is slow, miners who are not yet aware of the new block keep mining on top of the old block. This can lead to two (or more) miners finding several valid children to the same block, without one hearing about the other. The system becomes split between groups who first heard of each block. This split is called a *fork*.

Nodes can change the chain they follow by performing a *ReOrg*. They revert the state created by one chain and update it according to the longest chain. Forks cause miners to waste mining power, trying to extend a chain that is eventually discarded.

To prevent forks, block propagation time must be much shorter than the interval between blocks. However, if nodes do not validate blocks before propagating them further, the system could be susceptible to DoS attacks, by flooding the system with invalid blocks. The significance of block processing time to Nakamoto-based blockchains has been demonstrated by Decker et al. [27] and Croman et al. [24] who measured the properties of the Bitcoin network, as well as by Gencer et al. [36] who also measured Ethereum's network.

Note that mining power is determined by the resources used specifically for solving the cryptographic puzzle. This hardware is not related to the resources required from the node to serialize and process transactions.

Improving scaling in Proof-of-Work blockchains To overcome this dependency on network properties, various protocols improve on Nakamoto's approach to better utilize the available bandwidth. In Bitcoin-NG [33], leader election is separated from transaction serialization. The leader publishes a small key-block that quickly propagates through the system and then publishes many micro-blocks at network speed. Byzcoin [44], Algorand [38] and Hybrid Consensus [56] further improve latency by electing committees and running a classic consensus protocol between members of the committees to finalize blocks. These solutions rely on BFT algorithms to achieve finality on blocks and advance the blockchain. As such, they are resilient to an adversary of up to 1/3 of the network, compared to 1/2 in Nakamoto consensus.

3 Related Work

We overview here the various approaches to scaling blockchain performance, focusing on proposals considering splitting the blockchain into sub-chains as well as other endeavors to alleviate the load on the single blockchain node.

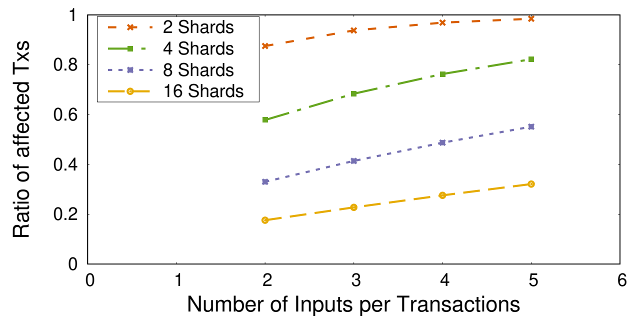

Figure 1: Ratio of effected transactions by attacking a single shard

3.1 Concurrent sub-chains (Sharding)

Several solutions suggest to shard the blockchain, splitting it into sub-chains. Each sub-chain is maintained by a committee running a Byzantine Fault Tolerant (BFT) protocol, tolerant of an adversary of up to 1/3 of the network. Elastico [49] was one of the first proposals of this scheme. OmniLedger [45] builds on top of ByzCoin [44] while dividing the system into shards. RapidChain [66] closely follows the ideas presented in Elastico and OmniLedger but proposes different solutions to committee election and cross-shard transactions, not dependent on the user. Both OmniLedger and RapidChain use the UTXO model to distribute transactions across shards. Chainspace [3] suggests a similar sharding solution for a smart contracts system [64].

DoS vulnerability Unlike Ostraka, sub-chain systems are vulnerable to DoS attacks. In these systems, transactions are usually assigned to shards according to their `TxHash`. An attacker can overwhelm only a single shard with many transactions much more cheaply than by saturating the entire system. Moreover, since each transaction may affect multiple shards, transactions in other shards can also be affected.

We evaluate the effectiveness of this attack. Consider a system of n shards and transactions with a single output and k inputs. The probability of a transaction to either have its output or one of its inputs in the shard undergoing the attack is the complement of the probability that the output and all inputs not in the attacked shard, $1-\left(\frac{n-1}{n}\right)^{k+1}$. Note that this is a lower bound: The number of transactions not able to execute is even higher when considering transaction dependant on pending transactions. Figure 1 illustrates this result. It shows, for example, that even with 2-3 inputs, in a 2-sharded system about 90% of the transactions are affected, but even with 16 shards about 20% of the transactions are affected.

Performance Sub-chain systems use 2-phase commit [45, 66] to atomically commit transactions that affect multiple shards. Since most transactions affect multiple shards, the additional overhead is considerable. The integrated node architecture of Ostraka avoids this limitation and it achieves linear scaling in the number of shards.

Node load Ostraka nodes shard the data internally, such that every node maintains the entire state. This is seemingly a difference from sub-chain systems where each node only maintains a subset of the state. However, as we show below, this difference only holds in what we call *democratic environments*, where there are many miners (stakeholders for PoS) with similar mining power [45, 49, 66]. In non-democratic environments, where the number of miners is small, sub-chain systems are also require nodes to store the entire state. Prominent systems relying either on *PoW* or *PoS* are typically non-democratic [26]. As of today, in Bitcoin, for example, close to 85% of the mining power is controlled by 10 mining entities [13].

Recall that sub-chain systems rely on BFT protocols to maintain each of the sub-chains. The underlying BFT protocol assumes a bound on the number of Byzantine nodes, where a node is a unit of mining power (stake for PoS). Even if the bound holds for the entire system (e.g., less than a third of the mining power belongs to malicious principals), in order to guarantee the bound holds, there should be sufficiently many nodes in each shard [45]. For example, with a 25% adversary, each shard should have at least 600 nodes [45]. Since the number of principals is not necessarily this large, participants instead split their mining power such that every mining power unit allows its owner one virtual node, or *share* in the system. To prevent an attacker from targeting a specific shard, each share is assigned to a shard uniformly at random.

We estimate the minimal miner size above which it will have shares in most of the shards. Denote by S the number of nodes per shard and by n the number of shards. Let r be the target ratio of shards that should be occupied in expectation and denote the miner size to achieve the target by α. The number of shares belonging to the miner is αnS. The exptected number of shards occupied by the miner is therefore $n\left(1-\left(\frac{n-1}{n}\right)^{\alpha nS}\right)$. This expression should equal the target number of shards, rn, and solving for α we get $\alpha = \frac{\log(1-r)}{Sn\log(1-1/n)}$.

To obtain concrete numbers, we take S to be either 300 or 600 as required by Kogias et al. [45] and vary the number of shards, n, from 1 to 32. We target an expected 90% of the shards and obtain the results shown in Figure 2. Indeed, even with only 300 nodes per shard and 32 shards, a miner would participate in 90% of the shards in expectation if its size is larger than 0.8%

The implication is that in sub-chain solutions in non-democratic scenarios, which are common in active blockchains, the miners must participate in all shards. Note that the same holds for users with high stakes in the system (e.g., exchanges, payment processors, merchants) who wish to validate their transactions: Since transactions depend on transactions from other shards, full validation requires the processing of all sub-chains. Therefore, node performance remains a challenge, even when sharding to sub-chains.

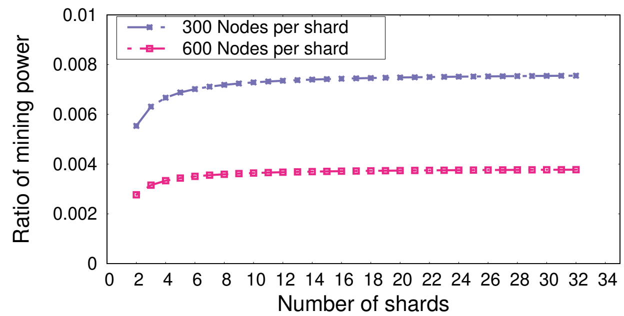

Figure 2: Ratio of mining power requiring participation in all shards

We proceed to review other related work.

3.2 Other Related Work

Classical systems sharding Sharding databases has been applied to many systems operating today [20, 23]. These systems distribute the database across several machines to allow for scale-out performance. These systems usually target databases, replying to queries from users, data can be intelligently partitioned, such that the number of shards involved in each transaction is minimal. This is an imperfect fit for a blockchain system, where each transaction requires random access to obtain address information. Unlike databases, our goal is to also distribute block validation, consisting of primarily CPU intensive operations. During the implementation of this work, we have tried using off-the-shelve database solutions, but the latency introduced by transactional databases, required for atomic transactions, was found to be unacceptable.

Client partitioning In Aspen [37] and SplitScale [55], a leader is chosen using *PoW* like in Bitcoin. The elected leader creates blocks on several concurrent chains. Clients can track a portion of the state, one sub-chain, but overall performance is not affected, in contrast to Ostraka.

Storage sharding Dietcoin [35] shards the UTXO set of a blockchain and uses Merkle trees of UTXO-shards for SPV. Clients can efficiently obtain proofs of inclusion for their transactions by obtaining only the relevant UTXO-shard. Abe et al. [1] reduce node storage complexity by using lower-degree replication of the blockchain itself, storing each block in a small number of nodes and locating blocks with consistent hashing. In contrast, Ostraka improves on the block validation process. In both solutions, miners keep the full state and perform full validation of each block, therefore these techniques can be used with Ostraka nodes.

Node scaling proposals Dickerson et al. [28], implement concurrent execution in smart contracts in a virtual machine model as used in Ethereum [64]. They add a concurrent schedule to each block, enabling consistent parallel execution across different nodes. However, the nature of the VM model limits the level of their measured speedup to 2x. Read-after-Write dependencies of instructions in the VM model enforce a sequential execution to avoid an inconsistent state. Ostraka achieves arbitrary scaling; we observed an improvement of two orders of magnitude, limited only by the network and the capacity of our experimental testbed.

Scaling by using cryptographic primitives Current efforts suggest using cryptographic techniques such as zk-SNARKs [10], zk-STAKRS [9] and Mimblewimble [41] to achieve a combination of scalability and privacy. While they offer asymptotic advantages, the exact numbers are yet to be seen, and the privacy guarantees present a tradeoff, as they prevent validation of the system's state (cf. [62]).

Chancellor [60], proposes the concept of node sharding to improve Bitcoin Cash's performance. In this work, we specify a sharding architecture, evaluate it both analytically and experimentally, overcome the DoS vulnerability of a distributed node architecture and analyze its security.

Vermorel et al. [42] discuss the benefits of canonical block ordering, where all nodes use the same order of transactions to improve Bitcoin Cash's block transmission. Ostraka avoids the topological order for parallelization efficiency, but a canonical order is not applicable since transaction order is unique per node. We also formally prove the security of order-agnostic block validation.

Facebook recently presented Libra [47], a global payment system where an order of 100 validators are responsible for validating transactions [8] using a variant of the HotStuff [65] BFT algorithm. Similar to Ostraka, Libra aims to achieve high frequency of transaction without splitting the system into sub-chains. Although all Libra nodes process the entire blockchain serially, the paper states that data structures in Libra are chosen with parallelism in mind. Unlike Ostraka, Libra is using the VM model for transactions execution, so parallelization would call for different techniques than Ostraka's.

4 Architecture

In this section, we cover the structure and operation of Ostraka. Data structures used in Ostraka are similar to the Nakamoto client. Similarly to Nakamoto, Ostraka uses a mempool to store incoming transactions, a UTXO set to hold the current state of the blockchain, and stores blocks in a local database. The key difference is that in Ostraka, the mempool, UTXO set, and blockchain storage are distributed among several machines, called *shards*, to allow for horizontal scaling.

We go into further details on the various components of Ostraka in (§4.1). Explain the changes required from a standard blockchain client in (§4.2) and how these components interact to perform all tasks of a blockchain node in (§4.3).

4.1 System Components

In Ostraka a machine, called the *coordinator*, keeps track of the blockchain and orchestrates communication within the node (intra-shard) and with other nodes (inter-node). We refer to the shards of the same node as *sibling shards*. Sharding in Ostraka is local on each node and does not mandate the same number of shards per node. Transactions are distributed among shards and only the shard responsible for a given transaction, stores and validates it. Transaction outputs are stored in the *UTXO-shard* of the shard they are assigned to.

Coordinator The coordinator orchestrates the node operation. When the node is first initialized, all shards connect to the coordinator. The coordinator is responsible to connect them. It orchestrates the p2p communication with other peers in the system.

The coordinator tracks the state of the blockchain. It stores the header of the block and determines the main chain (e.g. with most *PoW* in Bitcoin and Ethereum). It instructs the shards of which block-shards to request, process and validate. In case of a reorg, the coordinator commands the shards to roll back their state to the block of the divergence and start following the new chain.

In addition, the coordinator takes part in the process of block validation. It validates the block header, making sure it meets the consensus requirement. In general, all tasks assigned to the coordinator are of $O(1)$ complexity, with respect to the number of transactions in a block, except for Merkle tree validation. The Coordinator stores only block headers, occupying $O(1)$ with respect to transactions in a block.

Shards Each shard connects to the coordinator who assigns it a local *ShardID*. The *ShardID* determines which transactions the shard is responsible for. Once all shards are connected, the coordinator sends each shard the *ShardID*, IP and port of all sibling shards. Next, shards connect to each other to form a clique. Via the *ShardID*, transactions are mapped to the shard responsible for storing it and its outputs. It performs the mapping by calculating a hash of the transaction, with an added *salt*, and taking the most significant bits of the result.

Shards maintain the data structures of the blockchain – the mempool, UTXO set and the blockchain itself. Each shard stores transactions as partial blocks or *block-shard*, containing only transactions relevant to its *ShardID*, indexed by the block hash. It stores unspent outputs in a UTXO-shard, and pending transactions in a local mempool. Shard perform transactions routing and validation.

4.2 Sharding Design

As explained, in the UTXO model, transactions have inputs and outputs. Each transaction has a unique `TxHash`, created by applying the SHA-256 function to the transaction content. Inputs reference previous transaction outputs by `TxHash` and the `index` of the output in the transaction. Therefore, `TxHash` is the natural index for transactions. Transactions can be stored, indexed by their `TxHash` as the result of SHA-256 is unique to each transaction, under the assumption it is a secure cryptographic function. Transaction outputs can be indexed by the `TxHash` and `index` in each transaction input. Indeed, `TxHash` is used as index by other sharding proposals, such as OmniLedger [45], SplitScale [55], RSCoin [25], Elastico [49].

No transaction order In unified blockchain clients storage and processing are all performed on a single machine(e.g., [52, 64]), transactions in a block are ordered as a list and a transaction can only spend outputs created in a previous block, or earlier in the current block's list. We call this common approach, the *topological-order* algorithm. Using a topological order in Ostraka would have implied unnecessary dependencies, as each node has to verify for each input both that the referenced output exists and that it is earlier in the block order. The order of transactions within a block as used in unified-node protocols does not fit Ostraka's distributed architecture. Instead, Ostraka treats each block as an unordered set of transactions. Dependencies of a transaction are satisfied if the referenced output is in an earlier block or in the current block.

Merkle tree validation As in unified-node systems, the header of each Ostraka block contains the root of the Merkle tree of the block's transactions. This is necessary both for the blockchain's consensus protocol and for SPV validation. For all nodes should calculate the same root, all Ostraka nodes sort the transactions lexicographically by their `TxHash`. Shards calculate the `TxHash` of their transactions, while the coordinator is responsible to build and validate the Merkle tree.

Adding salt If Ostraka had sharded based on the lexicographic order, Merkle root calculation would be naturally distributed among the shards, each calculating a large subtree independently of the others. However, this would allow an attacker to cheaply form transactions that would all be assigned to the same shard on all nodes. The processing speed of all nodes would then be $1/\ell$ (for ℓ shards), possibly leading to a DoS.

To prevent such attacks, each node generates a random salt on initialization. It assigns its transaction based on its salt, so the assignment is unique per node. To each `TxHash`, we apply an additional hash as follows:

$$\texttt{NewTxHash} = \text{SHA-256}(\texttt{TxHash}||\mathit{salt})$$

Outputs in the UTXO set are stored with the original `TxHash`, but the responsibility of the shard storing the information is determined by `NewTxHash`. For a shard to figure out which sibling shard is responsible for a specific output, it must first calculate the `NewTxHash`, then take the most significant bits of the result. The number of shards does not have to be a power of two, other deterministic functions can also be used, e.g. the modulo function.

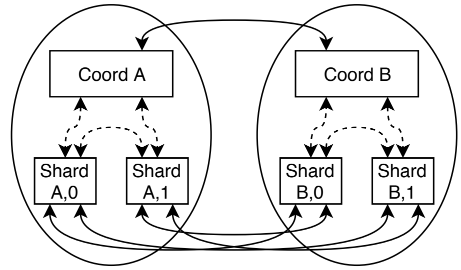

Figure 3: Two connected Ostraka nodes, with intra-node connections

The coordinator publishes its chosen salt to its shards and remote nodes. Shards can thus calculate the assignment of transactions to both sibling and remote shards. When a shard requests transactions from a peer-shard, it sends its *ShardID*. The peer-shard calculates the `NewTxHash` for the specific salt of the peer, and only sends transactions relevant to the requesting shard's *ShardID*. This way network bandwidth is distributed among the shards and each shard receives only transactions it requires. We refer the reader to (§5.3) for a detailed analysis of the attack and its mitigation.

4.3 Operation

We proceed to describe networking (§4.3), and Ostraka's sharded implementation of block validation (§4.3).

Establishing connections Nodes in the system connect to each other according to the topology rules (e.g. 8-12 random peers in Bitcoin). We focus on establishing connections between two peers.

When a new Ostraka node *A* joins the system, its coordinator requests a list of available peers. Assuming node *A* is trying to connect to node *B*. Coordinator of node *A* first sends a connection request to the coordinator of node *B*. Node *B* responds with the ID table of its shards. Coordinator *A* sends the information to its shards, who in turn connect to the shards of node *B*. An example is presented in figure 3

Block validation When a node receives a new block, it sends a INVENTORY message to its peers, informing them that a new block has been produced. A coordinator receiving an INVENTORY message a new, informs its shards of the new block hash, and the peer who has it. Each shard requests the relevant block-shard from the shards of the peer. It sends the peer shards its local *ShardID*, the total number of shards in the node, the salt used by the node and the block hash it is interested in. Each shard of the sending node iterates over the transactions associated with the block hash and calculates the `NewTxHash` using the salt. The sending shard sends the requester only transactions relevant to it.

Block validation comprises block header validation, where most operations are performed in $O(1)$, and transaction validation, where operations are $O(n)$ with respect to transactions in a block. Block header validation is performed by the coordinator. We go over the important steps necessary to approve a block header.

Duplicates: The coordinator checks for duplicate blocks. If the coordinator has already heard of the block before, it does not request it.

Validating *PoW*: Applying a hash function to the block header must be a valid solution to the cryptographic puzzle.

Coinbase: There must be a single Coinbase transaction. The Coinbase transaction is the reward paid to the miner for finding the block. Shards send the coordinator the number of Coinbase transactions they have seen. If the sum is not exactly one, the block is invalid.

Previous block hash: The coordinator checks that the block header contains a reference to the hash of the previous block.

Merkle validation: The coordinator is responsible for validating the Merkle tree. First, shards calculate the hashes of their transactions, the leaves of the Merkle tree, and send them to the coordinator. On a node with ℓ shards and a block with n transactions, each shard is performing $\frac{n}{\ell}$ hash operations. The coordinator sorts all transaction hashes according to the lexicographic order and then calculates the remaining n hashes to build the root. Given all leaves, calculating the Merkle tree is a simple process to parallelize. We implement a parallel computation of the Merkle tree, such that the coordinator uses several threads to process each level of the Merkle tree in parallel. Even though this calculation is linear in the number of transactions in the block, our measurements show it is much quicker than the other stages of block validation at the shards, and it happens in parallel. On a 4-core `c4.xlarge` machine, a Merkle root of $1,000,000$ transactions can be calculated in half a second, thus not a bottleneck of block validation.

Calculating the Merkle tree is currently the only operation the coordinator performs that is linear in the number of transactions in the block. We could delegate this process as well by switching from a Merkle tree to a different accumulator that allows each shard to accumulate its transactions and then the coordinator to aggregate the results of the shards. A possible candidate is the RSA accumulator [11]. Unfortunately, RSA accumulators as proposed by [7], require a trusted setup, reducing the decentralization properties of the system. Moreover, the RSA accumulator operations are so much more computationally expensive than calculating a hash function that although they achieve an asymptotic improvement, in practice they currently reduce system performance. In future implementations, accumulators as proposed by Boneh et al. [15] can be used as a drop-in replacement for Merkle trees.

Shards perform their part of block validation in parallel with the coordinator. The coordinator awaits approval from all shards prior to adding the current block to the chain. If at any point one of the shards comes across an illegal transaction, it sends a BADBLOCK message to the coordinator, who then discards the block.

To validate a transaction, a shard must first obtain all the required information. A shard only has its share of UTXOs in the UTXO set. Each shard is aware of the *ShardID* of all other shards. It constructs a list of missing outputs to request from each shard by using the `TxHash`, the salt, and the *ShardID*. Next, it requests the missing UTXOs from the appropriate sibling shards. This is done once, in a batch for each shard, to minimize communication overhead. Upon request, the shard sends the information, i.e. the `ScriptPubKey`, and value of the outputs to the requester.

Transaction inputs can reference outputs from either a previous or the current block. To make outputs available for other shards, each shard begins the validation process by adding all the outputs created by transactions in the current block to a data structure called a *view*. The view is used as a scratchpad for outputs that are either created or spent during the validation of the current block. When a shard receives a request for output information, it can search for the output either in its UTXO-shard or the view. After an output is requested by one of the sibling shards, it is marked as *spent* in the view, two requests for the same UTXO indicate a double-spend attempt, and will thus trigger a BADBLOCK message to the coordinator. Once all the information on transaction outputs is received, the shard proceeds to validate the scripts.

For each transaction, each shard performs the following validity checks:

Equal sum : For each transaction, the collective sum of the inputs must be at least as high as the outputs. Any remainder is the fee paid to the miner. The shard responsible for the transactions performs the check after it receives output information from sibling shards.

No double spends : Each input must reference an output that was previously produced, and has not yet been spent. A shard declares a block as invalid if some output is consumed twice, either by a request from a sibling or by a transaction in the block-shard.

No illegal minting : Each input must reference a previous output. If a shard receives a request for an output that is not in its view or UTXO-shard, the block is invalid.

Authorization : Each input must successfully comply with the spending condition of the output it consumes. Scripts can contain computationally expensive cryptographic functions e.g. signature validation and are usually the most costly stage of block validation. After receiving output information from sibling shards, all input scripts are validated.

Once all shards finish validation, they inform the coordinator. Once validation is complete, they each update their local UTXO-shard according to all spent and created outputs in the view. Each shard then stores the block-shard and removes its transactions from the local mempool.

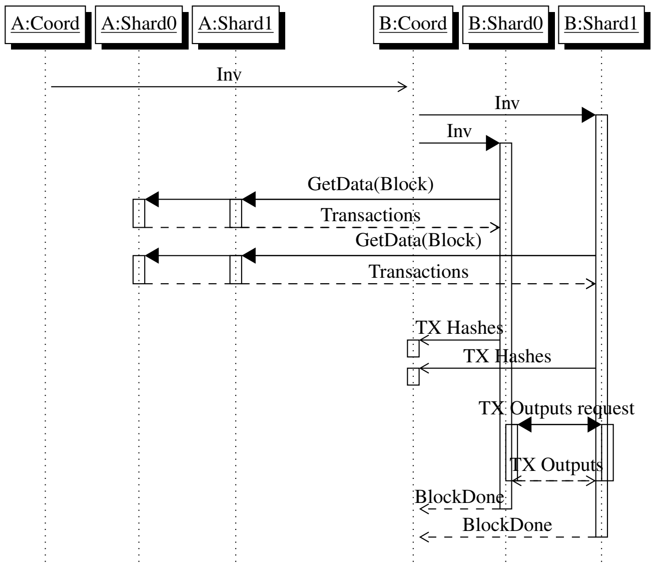

Figure 4: Ostraka block transmission and validation messages.

Figure 4 summarizes the steps of block propagation and validation. Transactions are first transmitted from node A to node B, then shards of node B interchange messages to obtain missing UTXO information.

Other operations Other inter-node operations a node performs can be easily implemented in Ostraka. Peers can request transaction directly from shards, and block headers from the coordinator.

One exception is that of a reorg. A reorg happens when a node needs to switch the main chain it follows. The coordinator and all shards must roll back to a state of a previous block, then update it according to the longest chain. To make a reorg possible, each shard maintains a record of the spent outputs in each block-shard. It records outputs as spent if they have been consumed by the shard, or if they have been requested by any of the sibling shards.

A reorg is performed as follows: First, the coordinator sends the hashes of blocks to roll back, as well as the hashes of the new blocks. Each shard fetches the spent outputs records of the rolled back block from the database. Each shard adds the spent outputs of each transaction in the block to the UTXO-shard. Next, it removes all outputs created by transactions in blocks-shard from the UTXO-shard. No communication is required between the shard during the rollback, and it can thus be quickly resolved.

5 Security

Ostraka can be integrated into a system using any blockchain algorithm (e.g., [2, 33, 38, 43, 44, 56, 58, 65]) with the UTXO model. Therefore, we prove that Ostraka is as secure as a system running the same blockchain protocol with a unified node. We show that the fact Ostraka is oblivious to the transaction order within the block does not affect the functionality and show that the sharding does not affect the functionality (§5.1).

Beyond functionality, blockchain protocols rely on propagation time. Therefore we also show that Ostraka's sharding does not enable attacks on node processing time. We show how individual-node shuffling prevents an adversary from affecting macro system performance (§5.3).

5.1 Functional Indistinguishability

Blockchain systems typically [33, 44, 52, 64] form blocks that are ordered such that each transaction follows transactions it depends on. We call this a *topological-order System*.

In contrast, Ostraka is agnostic to the blocks' order, and only requires a transactions' dependencies to be present anywhere in the block. We call such a system an *unordered system*. We show that an unordered system is functionally indistinguishable from a Topological-order system.

From topological to unordered systems Most stages of block validation (cf. §4.3) like signature and *PoW* verification are agnostic to transaction order. Two exceptions are the Merkle tree calculation and transaction dependency checks.

Merkle tree calculation depends on the transaction order, but the choice of order does not matter as long as all nodes calculate it consistently. For this purpose, Ostraka orders transactions lexicographically, according to their `TxHash`.

Of course, topologically ordered blocks may be invalid in a system that expects lexicographically ordered blocks, and vice versa. Moreover, transactions can be placed in a non-topological order and be rejected by the topologically ordered system, or in non-lexicographical order and be rejected by a lexicographic system. We thus need to show that any *set* of transactions can be processed as a legal block in a topological system if and only if they can be processed as a legal block in the unordered system.

To formally state this claim we first specify the two block-validation algorithms. Both algorithms take a UTXO set and a set of transactions and return either the updated UTXO set or $\perp$ if the transactions cannot form a legal block.

The topological-order system algorithm denoted $\mathcal{A}_T$, is shown in Algorithm 1. The algorithm receives a `UTXO_set` and the set of transactions T. The algorithm first sorts T topologically and then processes the transactions in order. It is important to note that the set T can always be ordered topologically: cycles between transactions cannot form since the references are by transaction hash. Assuming SHA-256 is a secure cryptographic hash function, and can be modeled as a random oracle, Harris [39] shows that the for a space of 2^κ, the expected number of steps to find a cycle is $\frac{1}{4}\sqrt{2\pi 2^\kappa}$, therefore an adversary cannot find such a cycle except with negligible probability in the hash output size. Therefore the

Algorithm 1: Topological $\mathcal{A}_T$

```
Input  : Set of transactions T, UTXO_set
Output : New UTXO_set or ⊥
1  (t_1, t_2, ..., t_k) ← T sorted in topological order
2  foreach TX ← t_1, t_2, ..., t_k do
3      foreach Ref_Out in TX do
4          if Ref_Out ∉ UTXO_set then
5              return ⊥
6          else
7              UTXO_set ← UTXO_set \ {Ref_Out}
8      foreach TX_out in TX do
9          UTXO_set ← UTXO_set ∪ {TX_out}
10     Validate TX
11 return UTXO_set
```

sorting in Algorithm 1, line 1 is possible and the algorithm is well defined.

The algorithm returns $\perp$ if at some point it encounters a transaction whose input is in not in the `UTXO_set`. For each transactions, the operation *Valitates TX* (line 10), covers all additional transaction validation steps after the referenced outputs were found, e.g., script validation, correct sum etc. If *Validate TX* is unsuccessful for any transaction, the algorithm returns $\perp$.

The unordered system algorithm denoted $\mathcal{A}_U$, is shown in Algorithm 2. Unlike Algorithm $\mathcal{A}_T$, in $\mathcal{A}_U$ transactions are not ordered. All outputs are first added to the `UTXO_set`, then the algorithm processes the transactions sequentially in an arbitrary order. Other than that, the operation is similar to $\mathcal{A}_T$: The block is rejected due to transaction validation failure or if a transaction's input uses a non-existent output.

Algorithm 2: Unordered ($\mathcal{A}_U$)

```
Input  : Set of transactions T, UTXO_set
Output : New UTXO_set or ⊥
1  foreach TX ∈ T do
2      foreach TX_out in TX do
3          UTXO_set ← UTXO_set ∪ {TX_out}
4  foreach TX ∈ T do
5      foreach Ref_Out by TX do
6          if Ref_Out ∉ UTXO_set then
7              return ⊥
8          else
9              UTXO_set ← UTXO_set \ {Ref_Out}
10     Validate TX
11 return UTXO_set
```

We show that Algorithms $\mathcal{A}_T$ and $\mathcal{A}_U$ are functionally indistinguishable.

Lamma 1. *Given the same* `UTXO_set` *and set of transactions T, $\mathcal{A}_T$ and $\mathcal{A}_U$ return the same value.*

We prove the lemma by showing that $\mathcal{A}_T$ returns $\perp$ if and only if $\mathcal{A}_U$ returns $\perp$.

$\mathcal{A}_T \leftrightarrow \mathcal{A}_U$

Proof. We prove in each direction separately.

$\mathcal{A}_T \rightarrow \mathcal{A}_U$ We first show that if $\mathcal{A}_T$ rejects an input tuple $(T, \texttt{UTXO\_set})$ then $\mathcal{A}_U$ also rejects this input.

If $\mathcal{A}_T$ returns $\perp$, there exists some transaction $TX_i \in T$, at index i, and an input in TX_i, referencing an UTXO *Ref_Out* $\notin$ `UTXO_set`, produced by an output *TX_out*. This can be a result of three cases, and we show that $\mathcal{A}_U$ rejects as well.

1. The UTXO is not in the `UTXO_set` and is not produced by any transaction preceding TX_i: There can be no transaction succeeding TX_i which produces *TX_out* as it contradicts the topological order, thus, there is not *any* transaction producing output *TX_out* in T. Upon processing TX_i, $\mathcal{A}_U$ return $\perp$ (Algorithm 2 line 7).

2. The UTXO is not in the `UTXO_set` and is produced by some transaction TX_k, preceding TX_i but consumed by some transaction TX_j, succeeding TX_k but preceding TX_i: In $\mathcal{A}_U$, *TX_out* is added to the `UTXO_set` in line 3, $\mathcal{A}_U$ processes either TX_j or TX_i first. Without loss of generality, TX_i is processed first. Upon processing TX_j, $\mathcal{A}_U$ would not find UTXO *Ref_Out* in the `UTXO_set` and return $\perp$ (Algorithm 2 line 7).

3. The UTXO is in the `UTXO_set` and it is consumed by some transaction TX_j, preceding TX_i: This is similar to the previous case, except now the UTXO is in the `UTXO_set` already.

$\mathcal{A}_U \rightarrow \mathcal{A}_T$ If $\mathcal{A}_U$ returns $\perp$, there exists some transaction $TX_i \in T$, and an input *Ref_Out* in TX_i, referencing an UTXO *Ref_Out* $\notin$ `UTXO_set` produced by an output *TX_out*. This can also be a result of three cases, and we show that $\mathcal{A}_T$ rejects as well.

1. The UTXO is not in the `UTXO_set` and is not produced by *any* transaction in T: In $\mathcal{A}_T$, the UTXO is not produced by *any* transaction in T and in part, by any transaction preceding TX_i in the topological order of T in $\mathcal{A}_T$. Thus $\mathcal{A}_T$ return $\perp$ upon processing transaction TX_i (Algorithm 1 line 5).

2. The UTXO is not in the `UTXO_set` and is produced by some transaction TX_k, but is also consumed by some transaction TX_j : In the topological order of T, both TX_j and TX_i must succeed TX_k. Transaction TX_k is processed first and adds *Ref_Out* to the `UTXO_set`. Without loss of generality, TX_i is processed first and removes *Ref_Out* from the `UTXO_set`. Upon processing TX_j, output *Ref_Out* is not in the `UTXO_set` and system T return $\perp$ (Algorithm 1 line 5).

3. The output is in the `UTXO_set` and it is also consumed by some transaction TX_j: Similar to the second case, but *Ref_Out* is in the `UTXO_set`.

Therefore, given the same `UTXO_set`, and transactions T, $\mathcal{A}_T$ return $\perp$ if and only if $\mathcal{A}_U$ returns $\perp$. Both algorithms add the same set of *outputs* to the `UTXO_set`, and remove the same set of *utxos*. No output can be added to the `UTXO_set` twice, and any attempt to remove an output twice results in $\perp$. Thus the new `UTXO_set` returned by each algorithm is identical. $\square$

Algorithm 3: Ostraka, $\mathcal{A}_O$

Input : Set of transactions T, `UTXO_set`
Output : New `UTXO_set` or $\perp$

Distribute T into $T_1 \dots T_\ell$ and `UTXO_set` into $\texttt{UTXO\_set}_1 \dots \texttt{UTXO\_set}_\ell$
for all *Shards,* $S_i \leftarrow S_1 \dots S_\ell$ **do in parallel**
  Send (T_i, $\texttt{UTXO\_set}_i$) to shard S_i
for *Shards,* $S_i \leftarrow S_1 \dots S_\ell$ **do**
  Receive $\texttt{UTXO\_set}_i$
  if *Received* $\perp$ **then**
    Return $\perp$
  else
    $\texttt{UTXO\_set} \leftarrow \texttt{UTXO\_set} \cup \{\texttt{UTXO\_set}_i\}$
Return `UTXO_set`

Algorithm 4: Ostraka shard algorithm

Input : Set of transactions T_i, `UTXO_shard`
Output : New `UTXO_set` or $\perp$

1 Add all produced outputs in T_i to the `UTXO_shard`
/* Request TX outputs */
2 Request all referenced outputs in T_i from siblings
3 Receive requests for missing outputs
4 **foreach** *Requested_Out* **do**
5   **if** *Requested_Out* $\notin$ *UTXO_shard* **then**
6   Send $\perp$ to Coordinator
7   **else**
8     $\texttt{UTXO\_shard} \leftarrow \texttt{UTXO\_shard} \setminus \{$*Requested_Out*$\}$
/* Receive TX outputs */
9 Send requested outputs to requesters
10 Receive requested outputs and add to `UTXO_shard`
/* Transaction validation */
11 **foreach** $TX \leftarrow t_1, t_2, \dots, t_k \in T_i$ **do**
12   **foreach** *Ref_Out by TX* **do**
13   **if** *Ref_Out* $\notin$ *UTXO_set* **then**
14   Send $\perp$ to Coordinator
15   **else**
16       $\texttt{UTXO\_shard} \leftarrow \texttt{UTXO\_shard} \setminus \{$*Ref_Out*$\}$
17   *Validate TX*
18 return `UTXO_shard`

5.2 From unified node to distributed node

We now show that Ostraka is functionally indistinguishable from $\mathcal{A}_U$. The distributed algorithm of Ostraka, denoted $\mathcal{A}_O$, is shown in Algorithms 3–4. A Coordinator algorithm receives the `UTXO_set` and transaction set T as input, and should return the updated `UTXO_set` if all transactions are valid, or $\perp$ otherwise. The Coordinator algorithm first distributes the `UTXO_set` and transactions among ℓ shards. Shards begin validation by adding all produced outputs in their set of transactions T_i, to their local `UTXO_shard`. They proceed by requesting missing output information from the relevant siblings. Once they receive the information, each shard proceeds to validate its set of transactions. If some transaction is invalid or if the UTXO it references is missing from the `UTXO_set`, the shard sends $\perp$ to the coordinator. Otherwise, the shard sends to the coordinator its updated `UTXO_shard`. If all shards returned updated UTXO sets, the coordinator returns their union. Otherwise, it returns $\perp$.

The pseudo-code makes a slight simplification of Ostraka that nonetheless represents its logical behavior. For performance, in the actual system each shard stores its `UTXO_shard` locally after it obtained it directly from neighbor shards, the coordinator does not distribute it on each block. The shards also obtain the transactions directly, rather than through the coordinator.

We show that an unordered-system algorithm and Ostraka are functionally indistinguishable.

Lamma 2. *Given the same `UTXO_set` and set of transactions T, $\mathcal{A}_U$ and $\mathcal{A}_O$ return the same value.*

As before, we prove the lemma by showing that $\mathcal{A}_U$ returns $\perp$ if and only if $\mathcal{A}_O$ returns $\perp$. Similarly to the previous proof, if the set T contains an illegal transaction it is detected by one of the shards. There are several cases for this, depending on the distribution of transactions among shards.

$\mathcal{A}_U \leftrightarrow \mathcal{A}_O$

Proof. We prove in both directions.

$\mathcal{A}_U \rightarrow \mathcal{A}_O$ We first show that if $\mathcal{A}_U$ rejects an input $(T, \texttt{UTXO\_set})$ then $\mathcal{A}_O$ also rejects this input.

If $\mathcal{A}_U$ returns $\perp$, there exists some transaction $TX_i \in T$, and an input in TX_i, referencing an UTXO *Ref_Out* $\notin$ `UTXO_set`, produced by an output *TX_out*. This can be a result of three cases, and we show that $\mathcal{A}_O$ rejects as well.

1. The UTXO is not in the `UTXO_set` and is not produced by *any* transaction in T: The UTXO is not produced by any transaction in T, thus no shard receives a transaction producing *Ref_Out* in $\mathcal{A}_O$. WLOG, some shard S_i receives TX_i. If *Ref_Out* references an output assigned to a shard S_j, when receiving request for *Ref_Out*, S_j does not have *Ref_Out* in its `UTXO_shard` and send $\perp$ (Algorithm 4 line 6). If *Ref_Out* references a transaction assigned to S_i, when processing transactions, *Ref_Out* is not in `UTXO_shard` of S_i, and S_i sends $\perp$ (Algorithm 4 line 14)

2. The UTXO is not in the `UTXO_set` and is produced by some transaction TX_k, but is also consumed by some transaction TX_j: WLOG, *TX_out* is produced by a transactions assigned to some shard S_k, and is added to its `UTXO_shard` (Algorithm 4 line 1). If both consuming transactions TX_j and TX_i, are assigned to sibling shards, then upon receiving all requests from sibling shards, S_k processes one of the requests first and remove *Ref_Out* from it `UTXO_shard`. Upon processing the second request, *Ref_Out* is not in `UTXO_shard`, and S_k sends $\perp$ (Algorithm 4 line 6). If TX_j is assigned to a sibling shard, and TX_i is assigned to S_k, *Ref_Out* is removed from `UTXO_shard` after all outputs are requested. Upon processing TX_i, S_k does not have *Ref_Out* in `UTXO_shard`, and send $\perp$ (Algorithm 4 line 14). If both TX_i and TX_j are assigned to S_k, one of them is be processed first and removes *Ref_Out* from the `UTXO_shard`, and upon processing the second one, *Ref_Out* is not be in `UTXO_shard` and S_k sends $\perp$ (Algorithm 4 line 14).

3. The UTXO is in the `UTXO_set` and it is also consumed by some transaction TX_j: This is similar to the second case, but *TX_out* is already in the `UTXO_shard` of S_k.

$\mathcal{A}_O \rightarrow \mathcal{A}_U$ If $\mathcal{A}_O$ returns $\perp$ and rejects, then some shard S_i sends $\perp$ to the Coordinator. This can be a result of several cases, and we show that $\mathcal{A}_U$ rejects as well.

First, notice that the `UTXO_set` and transactions T are distributed deterministically. There can be no two shards producing the same *TX_out*, and similarly, *Ref_Out* cannot reside in two UTXO-shards.

1. When processing requests for outputs, there is some UTXO *Ref_Out* $\notin$ `UTXO_shard`

   (a) The UTXO is not in the `UTXO_shard`, is not produced by any transaction in T_i, and is requested by some sibling shard S_j. Then there exists some transaction $TX_j \in T$ consuming UTXO. As `UTXO_set` is distributed deterministically, there can be no other `UTXO_shard` other than $\texttt{UTXO\_set}_i$ where the UTXO might reside. Therefore, it is not in the `UTXO_set` of $\mathcal{A}_U$, and upon processing TX_j, $\mathcal{A}_U$ returns $\perp$ (Algorithm 2 line 7).

   (b) The UTXO is not in the `UTXO_shard`, is produced by some transactions TX_k, and is requested by some two sibling shard S_j, S_l. Then there exist some two transaction TX_j and $TX_j \in T$, consuming the same UTXO. WLOG, in $\mathcal{A}_U$, TX_j is processed first and consumes the UTXO. Upon processing TX_l, $\mathcal{A}_U$ returns $\perp$ (Algorithm 2 line 7).

2. When processing transactions, there exists some transaction $TX_i \in T_i$, and an input in TX_i, referencing an UTXO *Ref_Out* $\notin$ `UTXO_shard`.

   (a) The UTXO is in the `UTXO_shard` and is requested by some sibling shard S_j: There is some other transaction $TX_j \in T$ consuming the UTXO. WLOG, in $\mathcal{A}_U$ TX_j is processed first, and upon processing TX_i, $\mathcal{A}_U$ returns $\perp$ (Algorithm 2 line 7).

   (b) The UTXO is not in the `UTXO_shard`, is produced by some transaction TX_k, and is requested by some sibling shard S_j. The case is similar to the previous after all outputs are added to the `UTXO_set` in Algorithm $\mathcal{A}_U$ (Algorithm 2 line 3).

   (c) The cases enumerated for $\mathcal{A}_U$ are same if all transactions are assigned to a single shard, and are not requested by any sibling.

Therefore, given the same `UTXO_set`, and transactions T, $\mathcal{A}_U$ return $\perp$ if and only if $\mathcal{A}_O$ returns $\perp$. Both algorithms add the same set of *outputs* to the `UTXO_set`, and remove the same set of *utxos*. No output can be added to the `UTXO_set` twice, and any attempt to remove an output twice results in $\perp$. Thus the new `UTXO_set` returned by each algorithm is identical. □

We can now show that a classical topological-order system with a unified node is functionally indistinguishable from an Ostraka node.

Theorem 1. *Given the same* `UTXO_set` *and set of transactions T, a topologically ordered system and Ostraka, are functionally indistinguishable.*

The proof follows directly from Lemmas 1 and 2.

5.3 Processing Time

We have shown the systems are functionally indistinguishable; it remains to show that an attacker cannot affect block processing time in Ostraka more than it could in a unified node. Limiting processing time is important to prevent an attacker from strangling the system, creating major slowdown and loss of throughput. This is part of the reason for a limited block size in Bitcoin and Gas limit in Ethereum [31].

Targeting a single shard . In a unified node, processing time is (roughly) the sum of processing times of all transactions. In Ostraka it is the maximum time it takes any shard to process its transactions.

If transaction distribution is identical for all nodes, then an attacker can cheaply generate transactions that are all placed in the same shard. For example, an attacker can create a block where all transactions start with four zeros in the MSB. This is cheap to perform, taking only 16 attempts on average per transaction. In general, to create a transaction starting with ℓ zeros, the attacker needs 2^ℓ attempts. For a block of L transactions, an expected total of $L \cdot 2^\ell$ attempts are required. Note that this analysis applies to any index function.

We denote by *Df* the distribution function applied to `TxHash` to determine the shard responsible for it. For ℓ shards, the *Df* results in a number from 1 to ℓ.

In Ostraka, we use salt to redistributed transactions. Each node is aware of the salt values of its neighbors. As connections are performed randomly, it is unlikely a node will obtain the values of all nodes in the system. Yet, even if one obtains all *salts*, attack difficulty rises exponentially with the number of targeted nodes.

We analyze the difficulty of the attack. Assume the attacker knows the salt values of k peers, $S_1..S_k$. We define the distribution function *Df* for a value x and a salt value S_i to be $Df(x, S_i) = \text{SHA-256}(x||S_i)$. To create a transaction resulting in v for all k peers, the attacker needs to find a `TxHash` such that $\forall 1 \le i \le k : Df(\texttt{TxHash}, S_i) = v$. The probability of finding such value, if the attacker tries to target one of ℓ shards is $(1/\ell)^k$. To cause an ℓ times slowdown for k peers, requires ℓ^k attempts per transaction.

It quickly becomes infeasible to affect even a small percentage of the system. For example, creating one transaction affecting the validation time of 100 nodes, requires the hashing rate of Bitcoin for 800,000 years [14], affecting only 10% of a system comprised of a thousand nodes. Of course, a single transaction is not enough, many must be aggregated into a malicious block for the attack to be effective.

An attack of this sort can be executed by either a miner, participating in the protocol, or a user, generating transactions. The attack has additional costs, on top of the computational power required to generate these transactions.

Creating a single large transaction . Creating an abnormally large single transaction occupying a significant percentile of a block can also cause a significant slowdown.

In Ostraka, each transaction is processed by a single shard. Creating an overwhelming large transaction, occupying the entire block, would annul the benefits of the distributed architecture. Adding salt does not help mitigating this attack vector, as it only changes *which* shard processes the transaction. In Ostraka, we mitigate this attack by adding a transaction size limit. Limiting transaction size allows for even distribution of transactions among shards.

Limiting transactions size might introduce an inconvenience to honest users wishing to aggregate an abnormally high number of outputs into a single transaction. Yet, making any payment is possible. A user can first issue aggregating transactions, merging several outputs into a new single output. Then, creating a new transaction, using the aggregated outputs. In general, a user can make any payment with only two types of transactions. A *join* transaction of two outputs into one, or a *fork* transaction of a single output into two.

6 Evaluation

Throughput of a blockchain system depends on the time it takes all nodes to receive a block. This depends on t_{hop}: the time it takes for a single node to receive and process the block. t_{hop} depends on both transmission time and processing time. Each node validates the block before it propagates it further, to prevent DoS attacks on the system [24, 27].

In our evaluation, we focus on the theoretical analysis and measurements of t_{hop}. Block propagation time across the system depends on the topology, consensus protocol and size of the system, which are not in the scope of this paper.

We begin by describing our evaluation methodology (§ 6.1). We proceed to measure block validation time, and how it is affected by adding shards to a node (§ 6.2). We present a theoretical analysis and measurements of t_{hop}, as a function of both network bandwidth and the number of shards (§ 6.3). Similarly, we analyze and present the effect of intra-shard bandwidth (§ 6.4) and finally conclude with scenarios and applications for various configurations of Ostraka.

6.1 Methodology

We evaluate performance of an Ostraka node under various circumstances. We give a theoretical analysis and measurements of how node capacity is affected by the number of shards assigned to a node, the network bandwidth available to the node, and bandwidth between sibling shards.

We implement the necessary elements for performance evaluation based on the *btcd* Bitcoin client [16]. We implement the intra-node and inter-node networking, as well as block validation protocol. We added or modified approximately 10*K* lines of code.

We measure the effects of network bandwidth and configuration of shards to estimate node performance. We measure block sending time between two Ostraka nodes, as well as block processing time on the receiving node. Unless stated differently, we used a `c4.large` EC2 instance (2 vCPUs 4GB RAM) for each shard, and a `c4.xlarge` instance (4 vCPUs 8GB RAM) for the coordinator of each node. A block is constructed of transactions with a single input, and 2 outputs and is on average 200 *Bytes* in size. Processing capacity of a single shard is approximately 1700 Tx/sec. Each block undergoes full validation, i.e. we do not assume some of the transactions have been previously known to the receiving node, and have already been validated. Transactions in each block are distributed among shards according to `TxHash` and salt. Transaction dependency within the block does not affect processing time, as transaction validation is performed in parallel, achievable by our block validation algorithm. All measurements are done three times. Each image with error bars presents the minimum, maximum and average measurements. Theoretical analysis is presented where applicable.

6.2 Node processing capacity

First we evaluate how block processing time affects t_{hop}. To isolate processing time, we measure block validation times of various configurations, with high intra-shard network bandwidth of 250 *Mbit/s* per shard.

The number of transactions in a block is the result of the block size, denoted *BSize*, divided by the mean transaction size, denoted *TxSize*. S_{tps} is the number of transactions per second a shard can process. By multiplying S_{tps} by the number of shards, *#Shards*, we get the processing rate of the node. Dividing number of transactions in a block by the node processing rate gives block processing time: $\frac{BSize}{TxSize} \cdot \frac{1}{S_{tps} \cdot \#Shards}$. Therefore, block processing time increases linearly with the size of the block.

We measure the processing time of blocks constructed with an increasing number of transactions. As expected, processing time in our measurements increases linearly as the size of the block increases. We repeat this experiment varying the number of shards from 1 to 32.

We estimate the number of transactions each configuration processes in ten seconds from the linear regression. Ten seconds was chosen arbitrarily to be an upper limit on the processing time of each node. We find the maximum block size a node is capable of processing within the limit, maximizing throughput without compromising security.

A node's processing rate can be increased by adding additional shards or by increasing the CPU core count of each

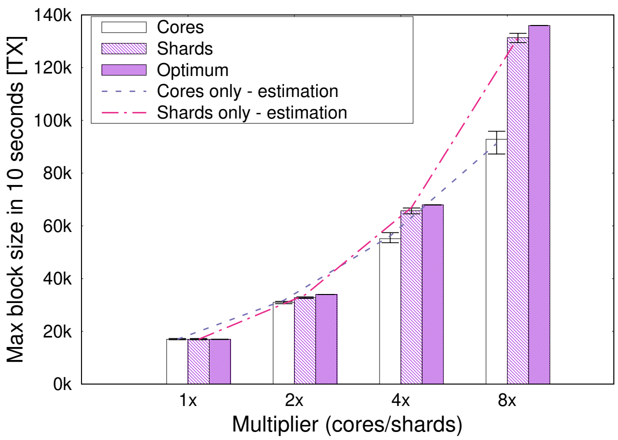

Figure 5: Node processing capacity. Adding shards compared to adding cores

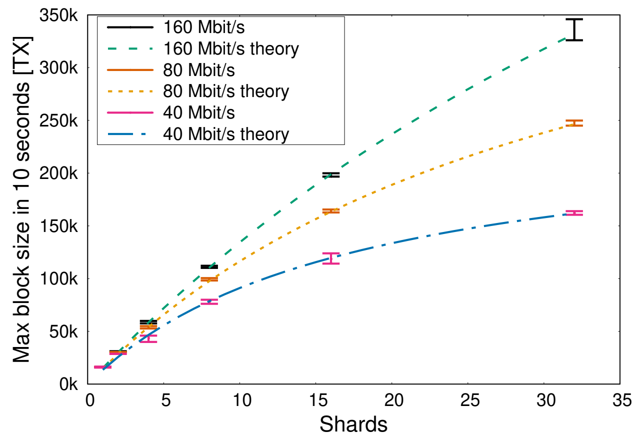

Figure 6: Theoretical and measured capacity, with increasing number of shards and available network bandwidth

shard. Amdahl's law [4] states that in a given workload, only the parallelizable part of the execution can be accelerated by additional cores or machines. For a workload S, a parallelizable percentage P, and a multiplier of K, the best achievable speedup is $\frac{1}{(1-P)+\frac{P}{k}}$.

We evaluate the benefits of additional shards compared to additional cores. We take a machine with 2 cores as a baseline. To measure core count effect, we test the processing rate with $4,8$ and 16 cores. To measure the effect of added shards, we test with $2,4$ and 8 shards. These represent multiplying the baseline by $2x, 4x, 8x$ cores/shards respectively.

We estimate the percentage of parallelism achieved with each configuration by comparing our measurements with the theoretical speedup. Measurements show that by using 8 times as many cores, we can achieve 5.5x improvement, while by using 8 times as many, we achieve 7.7x improvement, parallelizing 99.5% of the workload. These estimations and measurements are shown in figure 5. The optimum is the multiplication of the baseline.

While additional cores only accelerate script validation, sharding is beneficial for faster data lookup and I/O operations as well. For large UTXO sets, it allows keeping more entries in system memory, enabling faster access.

Importantly, while the number of cores on each machine is limited, the number of shards is not. Of course, we can combine the benefits of both.

6.3 Bandwidth

Next, we explore the effect of block delivery time on overall block time. Block delivery time is determined by network latency and bandwidth. We focus on network bandwidth as it is more significant for large blocks [24]. This time is proportional to the block size. Given a block size, we divide it by the bandwidth to obtain transmission time. As block size and the number of transactions per block are linearly dependent, overall block time increases linearly with block size. $t_{hop}[sec] = \frac{BSize}{BW} + \frac{BSize}{TxSize}\left(\frac{1}{S_{tps}\cdot\#Shards}\right)$

We now analyze node *capacity* in terms of transactions per seconds, as a function of both network bandwidth and the number of shards. The theoretical formula for node capacity in terms of transactions per second: $Capacity\left[\frac{TX}{sec}\right] = \frac{BS/TxSize}{BlockTime}$

To evaluate the capacity, we divide the number of transactions in a block by the time it takes to transmit and process it. Plugging-in overall block time time yields:

$$Capacity\left[\frac{TX}{sec}\right] = \frac{1}{\frac{TxSize}{BW} + \left(\frac{1}{S_{tps}\cdot\#Shards}\right)} \quad (1)$$

By adding more shards we expect t_{hop} to approach block transmission time, as block processing time approaches 0, in accordance with Amdahl's law.

We measure block processing time when both the network bandwidth and the number of shards varies. We measure with configurations of *1,2,4,8,16* and 32 shards, as well as bandwidth of $40, 80, 160$*Mbit/s* of overall incoming bandwidth. As before, we estimate capacity by evaluating the linear regression at ten seconds.

In figure 6 shows capacity measurements as well as theoretical analysis. Error bars represent our measurements for each configuration of bandwidth and shards. The dashed lines represent our theoretical analysis as shown in equation 1. We observe linear scaling without bandwidth restrictions, and an asymptotic upper bound of available bandwidth.

6.4 Intra-shard network

Part of the Ostraka protocol is requesting and sending outputs information between shards. When intra-shard bandwidth is high, time to send requests and later to send information to requesting shards is not significant. The transmission time of requests increases as intra-shard bandwidth becomes limited.

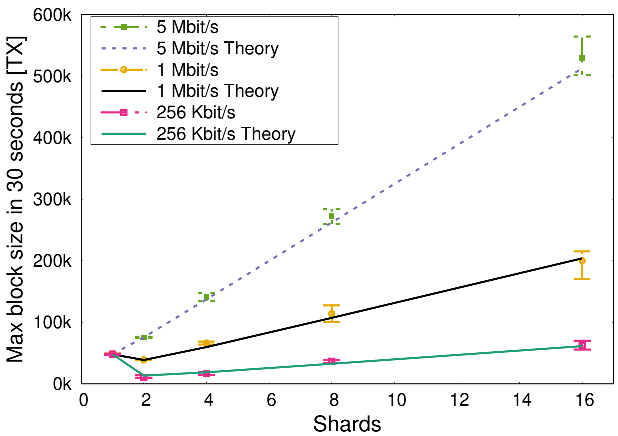

Figure 7: Transactions processed in 30 seconds with limited intra-shard bandwidth

For n transactions per block and ℓ shards, each shard has in expectation $\frac{n}{\ell}\frac{\ell-1}{\ell}$ outputs it needs to request from sibling shards. This is also the number of outputs it needs to send others.

In is important to note that although the number of siblings each shard must contact increases with the number of shards, the overall number of outputs requested/sent decreases. Thus, intra-shard communication decreases also linearly as we add additional shards.

We donate by BW_s the intra-shard bandwidth and by TX_{out}, the expected size required for sending and receiving outputs information per transaction. Processing time for a block-shard, when accounting for intra-shard communication is then:

$$\frac{BSize}{TxSize} \cdot \frac{1}{\#Shards} \cdot \left(\frac{1}{S_{tps}} + \frac{\#Shards - 1}{\#Shards} \cdot \frac{TX_{out}}{BW_s} \right) \quad (2)$$

We measure block processing time while restraining intra-shard bandwidth to $5 Mbit/s, 1 Mbit/s, 256 Kbit/s$ per shard.

Similarly to previous experiments, we estimate processing capacity by measuring block processing time for 20,000 up to 100,000 transactions. Since processing time is higher, we look at 30 seconds as a time target.

For $5 Mbit/s$, despite the limited bandwidth, the benefits of parallelizing transaction validation and storage outweigh the limits of cross-shard communication.

For $256 Kbit/s$ per shard, we observe a significant dip in performance when moving from a single shard to two shards. This is the result of sending information cross-shard becoming dominant. Figure 7 shows these effects. Our theoretical evaluation falls within the margin of error. When intra-shard bandwidth is extremely limited, it is more beneficial to operate a single shard.

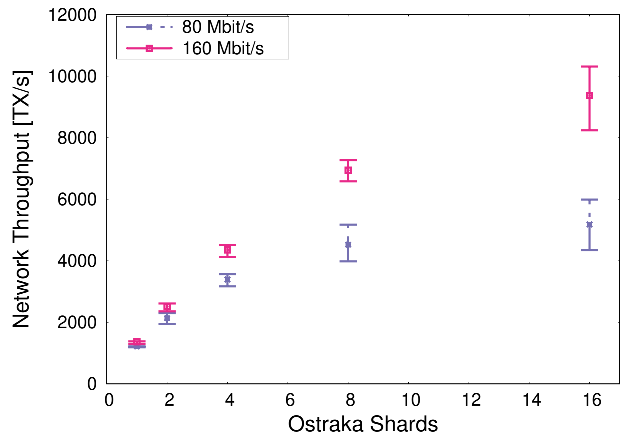

Figure 8: Bitcoin-NG simulation

6.5 Examples

To make the results of the analysis more concrete, we studied two particular points on the configuration space.

For a system run by large-scale players, we consider nodes with 64 shards, each an 8-core system, with an intra-shard bandwidth of 1 $Gbit/s$ per shard. We measured a processing rate of nearly 400,000 transactions per second in this setting.

In contrast, we consider nodes run by a small number of collaborators who trust each other. Here, we take 8 shards, each a 2 core system, with an intra-shard bandwidth of 20 $Mbit/s$ per shard. Even in this case, we measured processing rate of 10,000 transactions per second.

Finally, to assess the benefits of increasing block validation rate on a blockchain system, we take as an example the Bitcoin-NG [33] protocol. We simulate 1,000 sharded nodes implemented with Ostraka instead of unified nodes as in the original paper. We produce an event-driven simulation, with identical node configurations, a network topology similar to Bitcoin and fixed bandwidth per node. Figure 8 shows the effect of adding additional shards to each node. Network topology and properties strongly affect the protocol's performance, but the trend is clear. Ostraka allows the system to achieve an order of magnitude improvement in throughput.

7 Conclusion

We present Ostraka, an architecture for scalable and distributed blockchain nodes. Ostraka overcomes security and performance limitations of previous solutions and achieves linear scaling in the number of shards. We show how Ostraka fits the non-democratic environment of current blockchain ecology and allows for scaling without compromise on security. Our experiments achieved 400k tx/sec at the capacity of our experimental testbed. Together with a high-performance consensus protocol, Ostraka can saturate network capacity.

References

[1] Ryosuke Abe, Shigeya Suzuki, and Jun Murai. Mitigating bitcoin node storage size by dht. In *Proceedings of the Asian Internet Engineering Conference*, pages 17–23. ACM, 2018.

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. *CoRR*, abs/1612.02916, 2016.

[3] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[4] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, nj, apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, 2007.

[5] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. " O'Reilly Media, Inc.", 2014.

[6] Lawrence M Ausubel. The failure of competition in the credit card market. *The American Economic Review*, pages 50–81, 1991.

[7] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 480–494. Springer, 1997.

[8] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain.

[9] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46, 2018.

[10] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*, pages 90–108. Springer, 2013.

[11] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 274–285. Springer, 1993.

[12] Bitcoin wiki. Bitcoin wiki. `en.bitcoin.it/wiki/Main_Page`.

[13] Blockchain.com. Hashrate distribution. `https://www.blockchain.com/en/pools`.

[14] blockchain.com. Hash rate, 2019. `https://www.blockchain.com/charts/hash-rate`. Accessed March 2019.

[15] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. Technical report, Cryptology ePrint Archive, Report 2018/1188, Tech. Rep, 2018.

[16] btcsuite. Open Source Bitcoin Client Software. `https://github.com/btcsuite/btcd`.

[17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[18] Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, 2016.

[19] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[21] VNI Cisco. Cisco visual networking index: Forecast and trends, 2017–2022. *White Paper*, 2018.

[22] Coin.dance. Coin dance, 2019. `https://coin.dance/nodes`. Accessed March 2019.

[23] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[24] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün Sirer. On Scaling Decentralized Blockchains. In *Proc. 3rd Workshop on Bitcoin and Blockchain Research (BITCOIN 2016)*, 2016.

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

[26] Daniel Goldman. are we decentralized yet. http://web.archive.org/web/20190613115644/https://arewedecentralizedyet.com/.

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

[28] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 303–312. ACM, 2017.

[29] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[30] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[31] Ethereum.org. The ethereum network is currently undergoing a dos attack, 2016. https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/. Accessed March 2019.

[32] ethernodes.org. The ethereum node explorer, 2019. https://www.ethernodes.org. Accessed March 2019.

[33] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2016)*, 2016.

[34] Facebook Inc. Facebook company info. https://newsroom.fb.com/company-info/, December 2018.

[35] Davide Frey, Marc X Makkes, Pierre-Louis Roman, François Taïani, and Spyros Voulgaris. Dietcoin: shortcutting the bitcoin verification process for your smartphone. *arXiv preprint arXiv:1803.10494*, 2018.

[36] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. Decentralization in Bitcoin and Ethereum Networks. *arXiv preprint arXiv:1801.03998*, 2018.

[37] Adem Efe Gencer, Robbert van Renesse, and Emin Gün Sirer. Service-oriented sharding with aspen. *arXiv preprint arXiv:1611.06816*, 2016.

[38] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[39] Bernard Harris. Probability distributions related to random mappings. *The Annals of Mathematical Statistics*, pages 1045–1062, 1960.

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

[41] Tom Elvis Jedusor. Mimblewimble, 2016.

[42] Joannes Vermorel. Canonical Transaction Ordering for Bitcoin. https://blog.vermorel.com/pdf/canonical-tx-ordering-2018-06-12.pdf.

[43] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[44] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[45] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger. *IACR Cryptology ePrint Archive*, 2017:406, 2017.

[46] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[47] Libra.org. Libra, 2019. https://libra.org. Accessed July 2019.

[48] Litecoin wiki. Litecoin wiki. https://litecoin.info/index.php/Main_Page.

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

[50] Masav. Masav. https://www.masav.co.il/, Accessed March 2017.

[51] Ralph C Merkle. Protocols for public key cryptosystems. In *Security and Privacy, 1980 IEEE Symposium on*, pages 122–122. IEEE, 1980.

[52] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. http://www.bitcoin.org/bitcoin.pdf, 2008.

[53] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. Bitcoin and cryptocurrency technologies, 2016.

[54] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[55] Kazım Rıfat Özyılmaz, Harsh Patel, and Ankit Malik. Split-scale: Scaling bitcoin by partitioning the utxo space. *arXiv preprint arXiv:1809.08473*, 2018.

[56] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[57] Sveriges Riksbank. The Riksbank's e-krona project, report 1. Accessed March 2019, https://www.riksbank.se/globalassets/media/rapporter/e-krona/2017/rapport_ekrona_uppdaterad_170920_eng.pdf, 2017.

[58] Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies, 2018.

[59] Susan V Scott and Markos Zachariadis. Origins and development of swift, 1973–2009. *Business History*, 54(3):462–482, 2012.

[60] Shammah Chancellor. Sharding Bitcoin Cash. https://medium.com/@Bitcoin_ABC/sharding-bitcoin-cash-35d46b55ecfb.

[61] SPFS. SPFS. https://www.cbr.ru/psystem/mes/, Accessed March 2017.

[62] Josh Swihart, Benjamin Winston, and Sean Bowe. Zcash counterfeiting vulnerability successfully remediated. Accessed March 2019, https://z.cash/blog/zcash-counterfeiting-vulnerability-successfully-remediated/, 2019.

[63] Visa Inc. Visa Inc. at a Glance. https://usa.visa.com/dam/VCOM/download/corporate/media/visa-fact-sheet-Jun2015.pdf. Accessed May 2017, June 2015.

[64] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

[65] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

[66] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: A fast blockchain protocol via full sharding. *IACR Cryptology ePrint Archive*, 2018:460, 2018.

[67] ZCash. ZCash FAQ. https://z.cash/support/faq/.