The ZILLIQA Technical Whitepaper

[Version 0.1]

August 10, 2017

The ZILLIQA Team

www.zilliqa.com ✉ enquiry@zilliqa.com @zilliqa

***Abstract*—Existing cryptocurrencies and smart contract platforms are known to have scalability issues, i.e., the number of transactions they are capable of processing per second is limited, usually less than 10. As the number of applications utilizing public cryptocurrencies and smart contract platforms grow, the demand for processing high transaction rates in the order of hundreds and thousands of Tx/s is increasing.**

In this work, we present ZILLIQA— a new blockchain platform that is designed to scale in transaction rates. As the number of miners in ZILLIQA increases, its transaction rates are expected to increase. At Ethereum's present network size of 30,000 miners, ZILLIQA would expect to process about a thousand times the transaction rates of Ethereum. The cornerstone in ZILLIQA's design is the idea of *sharding* — dividing the mining network into smaller shards each capable of processing transactions in parallel.

ZILLIQA further proposes an innovative special-purpose smart contract language and execution environment that leverages the underlying architecture to provide a large scale and highly efficient computation platform. The smart contract language in ZILLIQA follows a *dataflow programming* style which makes it ideal for running large-scale computations that can be easily parallelized. Examples include simple computations such as search, sort and linear algebra computations, to more complex computations such as training neural nets, data mining, financial modeling, scientific computing and in general any MapReduce task.

I. INTRODUCTION

Cryptocurrencies and smart contract platforms are becoming a shared computational resource. One could view these platforms as a new generation of computers that synchronize over thousands of individual computers. However, existing cryptocurrencies and smart contract platforms have widely recognized limitations in scaling. Average transaction rates in Bitcoin [1], Ethereum [2], and related cryptocurrencies have been limited to below 10 (usually about 3-7) transactions per second (Tx/s). As the number of applications utilizing public cryptocurrencies and smart contract platforms grow, the demand for processing high transaction rates in the order of hundreds of Tx/s is increasing. A global payment network would likely require tens of thousands of Tx/s in capacity. Can we build a decentralized and open blockchain platform capable of processing at that scale?

The limitations in scaling up existing protocols are somewhat fundamental — they are rooted in the design of the consensus and network protocols. Therefore, even though re-engineering the parameters of the existing protocols in say Bitcoin or Ethereum (e.g., the block size or the block rate) may show some speedup, to support applications that need processing of thousands of Tx/s however requires rethinking the underlying protocols from scratch.

We present ZILLIQA— a new blockchain platform that is designed to scale in transaction rates. As the number of miners in ZILLIQA increases, its transaction rates are expected to increase as well. Specifically, ZILLIQA's design allows its transaction rates to roughly double with every few hundred nodes added to its network. As of this writing, the Ethereum mining network is over 30,000 nodes. At Ethereum's present capacity, ZILLIQA would expect to process about a thousand times the transaction rates of Ethereum.

ZILLIQA is a redesign from scratch and has been under research and development for over 2 years. The cornerstone in ZILLIQA's design is the idea of *sharding* — dividing the mining network into smaller consensus groups called *shards* each capable of processing transactions in parallel. If the mining network of ZILLIQA is say 8000 miners, ZILLIQA automatically creates 10 sub-networks each of size 800 miners, in a decentralized manner without a trusted co-ordinator. Now, if one sub-network can agree on a set of (say) 100 transactions in one time epoch, then 10 sub-networks can agree on a total of 1000 transactions in aggregate. The key to aggregating securely is to ensure that sub-networks process different transactions (with no overlaps) without double-spending.

The assumptions are similar to existing blockchain-based solutions. We assume that the mining network will have a fraction of malicious nodes/identities with a total computational power that is a fraction ($< 1/4$) of the complete network. It is based on a standard proof-of-work scheme, however, it has a new two-layer blockchain structure. It features a highly optimized consensus algorithm for processing shards.

ZILLIQA further comes with an innovative special-purpose smart contract language and execution environment that leverage the underlying architecture to provide a large scale and highly efficient computation platform. The smart contract language in ZILLIQA follows a dataflow programming style, where the smart contract can be represented as a directed graph. Nodes in the graph are operations or functions, while

an arc between two nodes represent the output of the first and the input to the second. A node gets activated (or operational) as soon as all of its inputs become valid and thus a dataflow contract is inherently parallel and suitable for decentralized systems such as ZILLIQA.

The sharded architecture is ideal for running large-scale computations that can be easily parallelized. Examples include simple computations such as search, sort and linear algebra computations, to more complex computations such as training a neural net, data mining, financial modeling, scientific computing and in general any MapReduce task among others.

This document outlines the technical design of ZILLIQA blockchain protocol. ZILLIQA has a new, conceptually clean and modular design. It has six layers: the cryptographic layer (Section III), data layer (Section IV), the network layer (Section V), the consensus layer (Section VI), the smart contract layer (Section VII) and the incentive layer (Section VIII). Before we present the different layers, we first discuss the system settings, underlying assumptions and threat model in Section II.

II. System Setting and Assumptions

Entities in ZILLIQA. There are two main entities in ZILLIQA: *users* and *miners*. A *user* is an external entity who uses ZILLIQA's infrastructure to transfer funds or run smart contracts. *Miners* are the nodes in the network who run ZILLIQA's consensus protocol and get rewarded for their service. In the rest of this whitepaper, we interchangeably use the terms miner and node.

ZILLIQA's mining network is further divided into several smaller networks referred to as a *shard*. A miner is assigned to a shard by a set of miners called *DS nodes*. This set of DS nodes is also referred to as the *DS committee*. Each shard and the DS committee has a *leader*. The leaders play an important role in the ZILLIQA's consensus protocol and for the overall functioning of the network.

Each user has a public, private key pair for a digital signature scheme and each miner in the network has an associated IP address and a public key that serves as an identity.

Intrinsic token. ZILLIQA has an intrinsic token called *Zillings* or ZILs for short. Zillings give platform usage rights to the users in terms of using it to pay for transaction processing or run smart contracts. Throughout this whitepaper, any reference to amount, value, balance or payment, should be assumed to be counted in ZIL.

Adversarial model. We assume that the mining network *at any point of time* has a fraction of byzantine nodes/identities with a total computational power that is at most $f < \frac{n}{4}$ of the complete network, where $0 \leq f < 1$ and n is the total size of the network. The factor $\frac{1}{4}$ is an arbitrary constant bounded away from $\frac{1}{3}$ selected as such to yield reasonable constant parameters. We further assume that honest nodes are reliable during protocol runs, and failed or disconnected nodes are counted in the byzantine fraction.

Byzantine nodes can deviate from the protocol, drop or modify messages and send different messages to honest nodes. Further, all byzantine nodes can collude together. We assume that the total computation power of the byzantine adversaries is still confined to standard cryptographic assumptions of probabilistic polynomial-time adversaries.

We however assume that messages from honest nodes (in the absence of network partition) can be delivered to honest destinations after a certain bound δ, but δ may be time-varying. The bound δ is used to ensure liveness but not safety [3]. In case such timing and connectivity assumptions are not met, it becomes possible for byzantine nodes to delay the messages significantly (simulating a gain in computation power) or worse "eclipse" the network [4]. In the event of network partition, as dictated by the CAP theorem, one can only choose between consistency and availability [5]. In ZILLIQA, we choose to be consistent and sacrifice availability.

III. Cryptographic Layer

The *cryptographic layer* defines the cryptographic primitives used in ZILLIQA. Similar to several other blockchain platforms, ZILLIQA relies on elliptic curve cryptography for digital signatures and a memory-hard hash function for proof-of-work (PoW).

Throughout this whitepaper, we extensively use `SHA3` [6] hash function to present our design. `SHA3` is originally based on `Keccak` [7] which is widely used in different blockchain platforms in particular Ethereum. In the near future, we may switch to `Keccak` to enable better interoperability with other platforms.

A. Schnorr Signature

ZILLIQA employs Elliptic Curve Based Schnorr Signature Algorithm (EC-Schnorr) [8] as the base signing algorithm. We instantiate the scheme with `secp256k1` curve [9]. The same curve is currently used in Bitcoin and Ethereum but for a different signing algorithm called ECDSA. Choosing EC-Schnorr over ECDSA has several benefits that we discuss below:

1) Non-malleability: Informally put, the *non-malleability* property means that given a set of signatures generated on a message using a private key, it should be hard for an adversary to produce a new signature for the same message that is valid for the corresponding public key. Unlike ECDSA which is malleable, EC-Schnorr has been proven to be non-malleable [10].

2) Multisignature: A multisignature scheme allows multiple signers to "aggregate" their signatures on a given message into a single signature which can be authenticated against a single public key that "aggregates" the keys of all the authorized parties. While, EC-Schnorr is natively a multisignature scheme (see [11]), ECDSA allows creating multisignatures but in a less flexible way.

ZILLIQA uses EC-Schnorr based multisignatures to reduce the signature size when multiple signatures are required on a message. Smaller signatures are particularly important in our consensus protocol where multiple parties need to agree on a data by signing it.

3) Speed: EC-Schnorr is faster than ECDSA since the latter requires computing an inverse modulo a large number. No inversion is required in EC-Schnorr.

The exact EC-Schnorr key generation, signing and verification procedures are given in Appendix A. In the Appendix, we also present how EC-Schnorr can be used as a multisignature scheme.

B. Proof of Work

ZILLIQA uses PoW only to prevent Sybil attacks and generate node identities. This is in contrast to many existing blockchain platforms (in particular Bitcoin [1] and Ethereum [12]), where PoW is used to reach distributed consensus. ZILLIQA employs *Ethash* [13], the PoW algorithm used in Ethereum 1.0.

Ethash is a *memory hard* hash function designed to make it easy to mine with GPUs but hard with specialized computing hardware such as ASICs. To achieve this, Ethash computation requires a considerable amount of memory (in GBs) and I/O bandwidth such that the function cannot be invoked in parallel on specialized computing hardware.

Roughly speaking, Ethash takes a data (for instance a block header) and a `nonce` of 64-bits as inputs and generates a 256-bits digest. The algorithm consists of four subroutines which are run in the given order:

1) **Seed generation:** Seed is a `SHA3-256` digest which is updated after every 30000 blocks called an epoch. For the first epoch it is the `SHA3-256` hash of a series of 32 bytes of zeros. For every other epoch it is always the `SHA3-256` hash of the previous seed.
2) **Cache generation:** The seed is used to generate a pseudorandom cache using `SHA3-512`. The size of the cache linearly increases with epoch. The initial size of the cache is 16 MB.
3) **Dataset generation:** The cache is then used to generate a dataset, where each "item" in the dataset depends on only a small number of items in the cache. The dataset is updated once every epoch so that the miners do not have to make changes to it very frequently. The size of the dataset also increases linearly with epoch. The initial size of the dataset is 1 GB.
4) **Mining and Verification:** Mining involves grabbing random slices of the dataset and hashing them together. Verification uses the cache to regenerate the specific pieces of the dataset needed to compute the hash.

IV. DATA LAYER

Broadly speaking, the *data layer* defines the data that constitutes the global state of ZILLIQA. By extension, it also defines the data needed by the different entities in ZILLIQA to update its global state.

A. Accounts, Addresses and State

ZILLIQA is an *account-based* system (as Ethereum). There are two types of accounts: *normal* account and *contract* account. A normal account is created by generating an EC-Schnorr private key. A contract account is created by another account.

Each account is identified by an *address* derived differently depending on its type. The address for a normal account is derived from the account's private key. For a given private key sk, the address $\mathcal{A}_{\text{normal}}$ is a 160-bit value computed as:

$$\mathcal{A}_{\text{normal}} = \mathsf{LSB}_{160}(\texttt{SHA3-256}(\mathsf{PubKey}(sk))),$$

where, $\mathsf{LSB}_{160}(\cdot)$ returns the rightmost 160 bits of the input and $\mathsf{PubKey}(\cdot)$ returns the public key corresponding to the input secret key. The address for a contract account is computed from the address of its creator and how many transactions the creator account has sent, aka *account nonce* (described below).

$$\mathcal{A}_{\text{contract}} = \mathsf{LSB}_{160}(\texttt{SHA3-256}(\texttt{address}||\texttt{nonce})),$$

where, `address` is the address of the creator account, and `nonce` is the creator's nonce value.

Each account (whether normal or contract) is associated with an *account state*. The account state is a key, value store and comprises of the following keys:

1) `account nonce`: (64 bits) A counter (initialized to 0) that counts the number of transactions sent from a normal account. In case of a contract account, it counts the number of contract creations made by the account.
2) `balance`: (128 bits) A non-negative value. Whenever an account receives tokens from another account, the received amount is added to the account's `balance`. When an account sends tokens to another account, the `balance` is reduced by the appropriate amount.
3) `code hash`: (256 bits) This stores `SHA3-256` digest of the contract code. For a normal account it is the `SHA3-256` digest of the empty string.
4) `storage root`: (256 bits) Each account has a storage which is again a key, value store with 256-bit keys and 256-bit values. `storage root` is a `SHA3-256` digest that represents this storage. For instance, if the storage is a trie, then `storage root` is the digest of the root of the trie.

The global state (`state`) of ZILLIQA is a mapping between account addresses and account states. It is implemented using a trie like data structure.

B. Transactions

A transaction is always sent from a `normal` account address and it updates the global `state` of ZILLIQA. A transaction has the following fields:

1) `version` (32 bits): Current version.
2) `nonce` (64 bits): A counter equal to the number of transactions sent by the sender of this transaction.
3) `to` (160 bits): Destination account address. In case the transaction creates a new contract account, this field is the rightmost 160 bits of `SHA3-256` of the empty string.
4) `amount` (128 bits): The transaction amount to be transferred to the destination address.

5) `gas price` (128 bits): *Gas* is defined as the smallest unit of computation. `gas price` is the amount that the sender is willing to pay per unit of *gas* for computations incurred in the transaction processing.
6) `gas limit` (128 bits): The maximum amount of gas that should be used while processing the transaction.
7) `code` (unlimited): An expandable byte array that specifies the contract code. It is present only when the transaction creates a new contract account.
8) `data` (unlimited): An expandable byte array that specifies the data that should be used to process the transaction. It is present only when the transaction invokes a call to a contract at the destination address.
9) `pubkey` (264 bits): An EC-Schnorr public key that should be used to verify the signature. The `pubkey` field also determines the sending address of the transaction.
10) `signature` (512 bits): An EC-Schnorr signature on the entire data.

Each transaction is uniquely identified by a `transaction ID` — a `SHA3-256` digest of the transaction data that excludes the `signature` field.

C. Blocks

The ZILLIQA protocol introduces two types of blocks (and thereby two blockchains): *transaction blocks* (TX-Block) and *directory service blocks* (DS-Block). TX-Block contains the transactions sent by users, while DS-Block contains metadata about the miners who participate in the consensus protocol.

1) DS Blocks: A DS-Block has two parts: the `header` and the `signature`. The `header` part of DS-Block has the following fields:

1) `version` (32 bits): Current version.
2) `previous hash` (256 bits): The `SHA3-256` digest of its parent block header.
3) `pubkey` (264 bits): The public key of the miner who did PoW on this block header.
4) `difficulty` (64 bits): This can be calculated from the previous block's difficulty and the block number. It stores the difficulty of the PoW puzzle.
5) `number` (256 bits): The number of ancestor blocks. The *genesis block* has a block number of 0.
6) `timestamp` (64 bits): Unix's time() at the time of creation of this block.
7) `mixHash` (256 bits): A digest calculated from `nonce` which allows detecting DoS attacks.
8) `nonce` (64 bits): A solution to the PoW.

The `signature` part of DS-Block contains the following two fields:

1) `signature` (512 bits): The signature is an EC-Schnorr based multisignature on the DS-Block header signed by DS nodes.
2) `bitmap` (1024 bits): It records which DS nodes participated in the multisignature. We denote the bitmap by a bit vector B, where, $B[i] = 1$ if the i-th node signed the header else $B[i] = 0$.

DS-Blocks form a *DS blockchain*.

2) Transaction Blocks: As discussed earlier, a DS-Block contains information on the nodes who reach consensus on transactions. TX-Block stores information on which transactions were agreed upon by the nodes in a DS-Block. Every DS-Block is linked to multiple TX-Blocks. A TX-Block has three parts: `header`, `data` and `signature`. The `header` consists of the following fields:

1) `type` (8 bits): A TX-Block is of two types, *micro block* (`0x00`) and *final block* (`0x01`). More on these in Section V-D.
2) `version` (32 bits): Current version.
3) `previous hash` (256 bits): The `SHA3-256` digest of its parent block header.
4) `gas limit` (128 bits): Current limit for gas expenditure per block.
5) `gas used` (128 bits): Total gas used by transactions in this block.
6) `number` (256 bits): The number of ancestor blocks. The *genesis block* has a block number of 0.
7) `timestamp` (64 bits): Unix's time() at the time of creation of this block.
8) `state root` (256 bits): It is a `SHA3-256` digest that represents the global state after all transactions are executed and finalized. If the global state is stored as a trie, then `state root` is the digest of the root of the trie.
9) `transaction root` (256 bits): It is a `SHA3-256` digest that represents the root of the Merkle tree that stores all transactions that are present in this block.
10) `tx hashes` (each 256 bits): A list of `SHA3-256` digests of the transactions. The `signature` part of the transaction is also hashed.
11) `pubkey` (264 bits): It is the EC-Schnorr public key of the *leader* who proposed the block.
12) `pubkey micro blocks` (unlimited): It is a list of EC-Schnorr public keys (each 264 bits in length). The list contains the public keys of the leaders who proposed transactions. The field is present only if it is a final block.
13) `parent block hash` (256 bits): It is the `SHA3-256` digest of the previous final block header.
14) `parent ds hash` (256 bits): It is the `SHA3-256` digest of its parent DS-Block header.
15) `parent ds block number` (256 bits): It is the parent DS-Block number.

The `data` part of a TX-Block contains the set of transactions. It has the following fields:

1) `tx count` (32 bits): The number of transactions in this block.
2) `tx list` (unlimited): A list of transactions.

The `signature` part of a TX-Block contains an EC-Schnorr based multisignature. It has the following two fields:

1) `signature` (512 bits): The signature is an EC-Schnorr based multisignature on the TX-Block header signed by a set of nodes. The signature is produced by a different set of nodes depending on whether it is a micro block or

a final block. Further details on the signatories is given in Section V-D.

2) `bitmap` (1024 bits): It records which nodes participated in the multisignature. We denote the bitmap by a bit vector B, where, $B[i] = 1$ if the i-th node signed the header else $B[i] = 0$.

The final blocks form the *transaction blockchain*. The transaction blockchain does not include micro blocks.

V. Network Layer

Zilliqa has been designed to scale in transaction rates. The main idea is that of *sharding*, i.e., dividing the mining network into small shards, each capable of processing transactions in parallel. In this section, we present the idea of *network* and *transaction sharding*.

A. Network Sharding

Network sharding, i.e., dividing the mining network into smaller shards is a two-step process. First, a dedicated set of nodes called the directory service committee (or DS committee) are elected which then shard the network and assign nodes to their shard. We present these processes below in further detail.

1) Directory Service Committee: To facilitate sharding of the network, we first elect a group of nodes, called *directory service* (DS) nodes. The DS nodes form a *DS committee*. The election of DS nodes is based on a proof-of-work puzzle that we refer to as PoW$_1$. The algorithm for PoW$_1$ is given in Algorithm 1.

Algorithm 1: PoW$_1$ for DS committee election.

Input: i: Current `DS-epoch`, DS_{i-1}: Prev. DS committee composition.
Output: `header`: DS-Block header.

1 **On each competing node:**
// get epoch randomness from the DS blockchain
// DB_{i-1}: Most recent DS-Block before start of i-th epoch
2 $r_1 \leftarrow \text{GetEpochRand}(\text{DB}_{i-1})$
// get epoch randomness from the transaction blockchain
// TB_j: Most recent TX-Block before start of i-th epoch
3 $r_2 \leftarrow \text{GetEpochRand}(\text{TB}_j)$
// pk: node's public key, `IP` = node's IP address
4 $\texttt{nonce}, \texttt{mixHash} \leftarrow \text{Ethash-PoW}(pk, \texttt{IP}, r_1, r_2)$
5 $\texttt{header} \leftarrow \text{BuildHeader}(\texttt{nonce}, \texttt{mixHash}, pk)$
// `header` includes pk and `nonce` among other fields
// `IP`, `header` is multicast to members in the DS committee
6 $\text{MulticastToDS}_{i-1}(\texttt{IP}, \texttt{header})$
7 **return** `header`

Each node that has successfully produced a valid `nonce` for PoW$_1$ earlier than other nodes proposes a `header` for a new DS-Block. Recall that a DS-Block has a `header` and a `signature` part. When a node does a PoW$_1$, it only generates a DS-Block `header`. The `header` is then multicast to the nodes in the DS committee. The DS committee then runs a consensus on the proposed DS-Block `header` and then builds a signature part. Once, $2f$ DS nodes have signed the DS-Block `header`, it is committed to the *DS blockchain*.

After a successful bootstrapping phase, at any time, the composition of the DS nodes is stipulated by a predefined window of size n_0. The most recent n_0 nodes who have successfully mined a DS-Block form the DS committee.

The average time between mining two consecutive DS-Blocks is referred to as the `DS-epoch`. The value of `DS-epoch` is set in a way to minimize the chances of two competing blocks. At the start of a `DS-epoch`, a new DS node joins the DS committee and the oldest member of the DS committee is churned out. This fixes the size of the DS committee to n_0 during any `DS-epoch`. The newest member of the DS committee then becomes the *leader* and leads the consensus protocol for the epoch (see Section VI for the consensus protocol). This further induces a strict ordering on the members of the DS committee.

One can show that if the DS committee size is sufficiently large (say above 800), then among the n_0 members of the committee at most $\frac{1}{3}$ are byzantine with high probability.

2) Resolving Conflicts: Our consensus protocol (to be presented in Section VI) does not permit forks in the DS blockchain. The forks may occur when multiple nodes solve the puzzle at roughly the same time. In order to resolve the conflict, each DS node retrieves the `nonce` field from the received headers and sorts them in the increasing order. Let us suppose that the largest `nonce` for the i-th DS node is n^i_{max}.

The leader of the DS committee then proposes his own `header` (that corresponds to the largest `nonce` he has seen) and runs a consensus protocol to agree on the DS-Block `header`. The i-th DS node then agrees to accept the proposed header only if the corresponding nonce is larger than or equal to n^i_{max}. Once the consensus is reached, the `signature` part of the DS-Block is built and the agreed upon winner then becomes the leader.

3) Generating Shards: Once the DS committee is elected, the actual sharding of the network can start. In order for a node to participate in the underlying consensus protocol, it has to perform a proof-of-work (PoW$_2$). The sharding protocol is repeated at the start of every `DS-epoch`. The algorithm for PoW$_2$ is given in Algorithm 2.

Algorithm 2: PoW$_2$ for shard membership.

Input: i: Current `DS-epoch`, DS_i: Current DS committee composition.
Output: `nonce`, `mixHash`: outputs of Ethash-PoW

1 **On each competing node:**
// get epoch randomness from the DS blockchain
// DB_{i-1}: Most recent DS-Block before start of i-th epoch
2 $r \leftarrow \text{GetEpochRand}(\text{DB}_i)$
// pk: node's public key, `IP` = node's IP address
3 $\texttt{nonce}, \texttt{mixHash} \leftarrow \text{Ethash-PoW}(pk, \texttt{IP}, r)$
// `IP`, `header` is multicast to members in the DS committee
4 $\text{MulticastToDS}_i(\texttt{nonce}, \texttt{mixHash}, pk, \texttt{IP})$
5 **return** `nonce`, `mixHash`

The computed valid `nonce` (and `mixhash`) for PoW$_2$ is then multicast to the DS committee. The DS nodes will

collectively accept just enough PoW solutions to be sharded into l *consensus committees* or *shards*, each with n_0 nodes to run consensus. Once enough number of PoW_2 solutions have been received by the leader of the DS committee, he initiates a consensus protocol to agree on the set of valid PoW_2 solutions. At the end of the consensus protocol, the leader generates an EC-Schnorr multisignature signed by the DS nodes. In order to proceed further, more than $2/3$ of the DS nodes must have agreed on the set of acceptable PoW_2 solutions.

Sharding leverages a deterministic function to assign a node to a shard. Let us assume that we need ℓ shards each having n_0 nodes. The `nonce` values are then sorted in the increasing order and the first n_0 nodes are assigned to the first shard, the next n_0 to the next shard and so on. The identity of the miner who proposed the largest `nonce` within a shard is declared its leader. This further induces a strict ordering on the members of the shard.

One can also show that if n_0 is sufficiently large (say above 800), then within each shard at most $\frac{1}{3}$ are byzantine with high probability.

B. Public Channel

The DS nodes publish certain information on the public channel, including the identities and connection information of the DS nodes, the list of nodes in each shard, as well as the sharding logic for transactions (explained in Section V-D). The public channel is untrusted and is assumed to be accessible by all nodes. In our implementation, our broadcast primitive implements such a public channel.

A user of our blockchain who would like to submit a transaction for acceptance can then check the information on sharding to get the shard responsible for processing her transaction. The information published on the public channel is expected to be signed by more than $2/3$ of the DS nodes that can be verified by any node or user.

C. New Nodes Joining ZILLIQA

For a new node to join the network, it can attempt to solve PoW_1 to become a DS node or a PoW_2 to become a member of a shard. To this end, it would need to obtain information on the randomness required for a PoW_1 or a PoW_2 from the blockchains. Once it obtains the randomness information, the new node can submit its solution to the DS committee.

D. Transaction Sharding and Processing

As presented in Section V-A, network sharding creates shards each capable of processing transactions in parallel. In this section, we present how a particular transaction gets assigned to a shard and how the transactions get processed. For this purpose, we use the following abstraction: $A \xrightarrow{n} B$ to indicate a transaction of n ZIL from the sender's account A to the receiver's account B.

1) Transaction Assignment: Any transaction say $A \xrightarrow{n} B$ gets processed by a single shard. Assuming that there are ℓ shards numbered from 0 to $\ell - 1$, a transaction is assigned to a shard identified by the $\lfloor \log_2 \ell \rfloor + 1$ rightmost bits of the sender's address, i.e., the address of the account A in the example. As the account address is a 160-bit integer, ℓ is bounded above as:

$$\lfloor \log_2 \ell \rfloor + 1 \leq 160.$$

In practice though, ℓ will be smaller than 100.

Once the assigned shard is identified, the transaction is then multicast to some nodes within the shard who then broadcast it further. Once the transaction reaches the leader of the assigned shard, it includes it in a TX-Block and runs the consensus protocol.

Double spend (or replay attacks) can be easily detected using the `nonce` present in every transaction. Recall that each transaction has a `nonce` that counts the number of transactions sent from the sender's account. Once a transaction gets into the transaction blockchain, the `nonce` is updated in the account's state and thereby in the global `state`. A transaction with a `nonce` value smaller than or equal to the current value in the global `state` gets rejected by the miners.

Sharding transactions based on the sender's account address natively allows shard members to detect double spend as every transaction from a sender will be processed within the same shard.

2) Transaction Processing: All the nodes within a committee can propose transactions. These transactions are sent to the leader to run a consensus protocol on which set of transactions forms the next TX-Block. Blocks proposed by each shard is called a *micro block* (identified by the type marker `0x00`). A micro block contains an EC-Schnorr multisignature by more than $\frac{2}{3}$ nodes from the shard. The leader also builds a bitmap B that identifies the public keys of the signers. $B[i] = 1$ if the i-th member of the shard has signed the TX-Block header. When a shard reaches a consensus on a TX-Block, its leader multicasts the block `header` and the `signature` to some of the DS nodes. The DS nodes then broadcast it within the DS committee so that the block reaches its leader. The `data` part of the block can be asynchronously sent to the nodes.

The DS committee then aggregates all blocks sent from the shards, and runs another round of the consensus protocol among themselves to agree on the *final block*. A final block is a TX-Block identified by the type marker `0x01`. A final block contains an EC-Schnorr multisignature by more than $\frac{2}{3}n_0$ nodes from the DS committee. The leader in the DS committee also builds a bitmap B that identifies the public keys of the signers. $B[i] = 1$ if the i-th member of the DS committee has signed the TX-Block header. The final block `header` and the `signature`, is then multicast to some nodes within each shard. The actual TX-Block `data` is not sent by the DS nodes.

Within each shard the following steps are taken to process the final block:

1) Each node in the shard verifies the EC-Schnorr multisignature using the public keys of the DS nodes. If the signatures are valid against more than $\frac{2}{3}n_0$ public keys represented by the bitmap, then the nodes perform the next checks.

2) For each transaction hash included in the final block `header`, the node checks whether its corresponding transaction content is available. If the corresponding transaction was proposed by the shard to which the node belongs, then the hash of the transaction data is compared with the hash contained in the final block `header`. If the transaction was proposed by another shard, the transaction data is shared asynchronously across shards.
3) Once the transaction data is available, the `data` part of the final block is reconstructed and the TX-Block is appended to the local transaction blockchain. The account state and the global `state` are accordingly updated.
4) If the transaction content is not available, the node temporarily invalidates the sending account of that transaction in its local view of accounts so that any other pending transactions for this account are rejected until the local transaction content can be brought in sync with the global `state`. Such rejected transactions will have to be retried by the sending node.

VI. Consensus Layer

As mentioned earlier, each shard and the DS committee have to run a consensus protocol on the micro blocks and the final blocks respectively. In this section, we present the *consensus layer* which defines the consensus protocol to run within each shard and the DS committee. In the rest of the discussion, we refer to shards and the DS committees collectively as a *consensus group*.

A. Practical Byzantine Fault Tolerance

The core of Zilliqa's consensus protocol relies on *practical byzantine fault tolerance* (PBFT) protocol proposed by Castro and Liskov [3]. We however improve its efficiency by using the idea of employing EC-Schnorr multisignature in the PBFT protocol as developed in [14], [15]. Use of EC-Schnorr multisignature lowers the normal case communication latency from $O(n^2)$ to $O(n)$ and reduces the signature size from $O(n)$ to $O(1)$, where n is the size of the consensus group. In this section, we present an overview of PBFT.

In PBFT, all the nodes within a consensus group are ordered in a sequence, and it has one *primary* node (or leader) and the others are referred to as *backup* nodes. Every round of PBFT has three phases as discussed below:

1) **Pre-prepare phase:** In this phase, the leader announces the next record (a TX-Block in our case) that the group should agree on.
2) **Prepare phase:** Upon receiving the pre-prepare message, every node validates its correctness and multicasts a *prepare* message to all the other nodes.
3) **Commit phase:** Upon receiving more than $\frac{2}{3}n$ prepare messages, a node multicasts a *commit* message to the group Finally, a node waits for more than $\frac{2}{3}$ commit messages to ensure that a sufficient number of nodes have made the same decision. Therefore, all honest nodes accept the same valid record.

PBFT relies upon a correct leader to begin each phase and proceed when the sufficient majority exists. In case the leader is byzantine it can stall the entire consensus protocol. To address this challenge, PBFT offers a *view change* protocol to replace the byzantine leader with another one. If the nodes do not see any progress for a bounded time, they can independently announce the desire to change the leader. If a quorum of more than $\frac{2}{3}n$ nodes decides that the leader is faulty, then the next leader in a well-known schedule takes over.

Owing to the multicast of every node in the prepare/commit phase, the communication complexity for PBFT in the normal case is $O(n^2)$.

B. Improving Efficiency

Classical PBFT uses message authentication code (MAC) for authenticated communication between nodes. As MAC requires a secret key shared between every two nodes, the nodes in one consensus group can agree on the same record with a communication complexity of $O(n^2)$ per node. Due to the quadratic complexity, PBFT becomes impractical when the committee has over 20 nodes.

To improve the efficiency, we use the ideas inspired from ByzCoin [15]:

1) We replace MAC with digital signatures to effectively reduce the communication overhead to $O(n)$.
2) In the meantime, to allow the other nodes to verify the agreement, one typical way is to collect the signatures from the honest majority and append them to the agreement, thereby resulting in the agreement size linear in the size of the consensus group. To improve on this, we employ EC-Schnorr multisignatures to aggregate several signatures into an $O(1)$-size multisignature.

We however cannot directly use the classical EC-Schnorr multisignature scheme in the PBFT setting. This is because in the classical setting all the signers agree on signing a given message and the signature is valid only when all the signers have signed the message. In the PBFT setting, we only require that the message be signed by over $\frac{2}{3}n$ nodes in the consensus group. One of the main modification required is to maintain a bitmap B for the signers who participate in the signing process. If the i-th node participated in the process, $B[i] = 1$, else it is 0. The bitmap is build by the leader. The bitmap can then be used by any verifier to validate the signature. The resulting protocol is left in Appendix B.

C. Zilliqa Consensus

In Zilliqa, we use PBFT as the basis consensus protocol and employ two rounds EC-Schnorr multisignatures to replace the prepare and commit phases in PBFT. The different modifications to the PBFT phases are explained below.

1) **Pre-prepare phase:** As in standard PBFT, the leader distributes a TX-Block or a statement (signed by the leader) to all the nodes in the consensus group.
2) **Prepare phase:** All honest nodes check the validity of the TX-Block and the leader collects responses from more than $\frac{2n}{3}$ nodes. This guarantees that the statement

proposed by the leader is safe and consistent with all previous histories. The signature is generated using EC-Schnorr multisignature. The leader also builds the bitmap of nodes who signed the TX-Block.

3) **Commit phase:** To ensure that more than $\frac{2n}{3}$ nodes know the fact that more than $\frac{2n}{3}$ nodes have verified the TX-Block, we perform a second round of EC-Schnorr multisignature. The statement being signed is the multisignature generated from the last round.

At the end of the three phases, the consensus is reached on the TX-Block proposed by the leader.

D. Leader Change

In our consensus protocol, if the leader is honest, it can drive the nodes in the consensus group to reach agreements on new sets of transactions continuously. However, if the leader is byzantine, it can intentionally delay or drop messages from honest nodes, and slow down the protocol. To penalize such malicious leaders, our protocol changes the leader of each shard and the DS committee periodically. This prevents the byzantine leader to stall the consensus protocol for an infinite time. Since all the nodes are ordered, the next leader will been chosen in a round robin manner.

In fact, the leader of a shard is changed after every micro block and the leader of the DS committee is changed after every final block. Let us assume that the size of the consensus group is n, then within a `DS-epoch`, we allow a maximum of n final blocks, each final block aggregating a maximum of 1 micro block per shard.

VII. Smart Contract Layer

ZILLIQA comes with an innovative special-purpose smart contract language and execution environment that leverages the underlying architecture to provide a large scale and highly efficient computation platform. In this section, we present the *smart contract layer* that employs a *dataflow programming* architecture.

A. Computational Sharding using Dataflow Paradigm

ZILLIQA's smart contract language and its execution platform is designed to leverage the underlying network and transaction sharding architecture. The sharded architecture is ideal for running computation-intensive tasks in an efficient manner. The key idea is the following: only a subset of the network (such as a shard) would perform the computation. We refer to this approach as *computational sharding*.

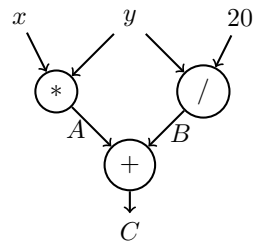
In contrast with existing smart contract architectures (such as Ethereum), computational sharding in ZILLIQA takes a very different approach towards how to process contracts. In Ethereum, every full node is required to perform the same computation to validate the outcome of the computation and update the global `state`. Albeit being secure, such a fully redundant programming model is prohibitively expensive for running large-scale computations that can be easily parallelized. Examples include simple computations such as search, sort and linear algebra computations, to more complex computations such as training a neural net, data mining, financial modeling, etc.

ZILLIQA's computational sharding approach relies on a new smart contract language that is not Turing-complete but scales much better for a multitude of applications. The smart contract language in ZILLIQA follows a *dataflow programming* style [16], [17]. In the dataflow execution model, a contract is represented by a directed graph. Nodes in the graph are primitive instructions or operations. Directed arcs between two nodes represent the data dependencies between the operations, i.e., output of the first and the input to the second. A node gets *activated* (or operational) as soon as all of its inputs are available. This stands in contrast to the classical von Neumann execution model (as employed in Ethereum), in which an instruction is only executed when the program counter reaches it, regardless of whether or not it can be executed earlier.

The key advantage of employing a dataflow approach is that more than one instruction can be executed at once. Thus, if several nodes in the graph become activated at the same time, they can be executed in parallel. This simple principle provides the potential for massive parallel execution. To see this, we present a simple sequential program in Figure 1a with three instructions and in Figure 1b, we present its dataflow variant. Under the von Neumann execution model, the program would run in three time units: first computing A, then B and finally C. The model does not capture the fact that A and B can be independently computed. The dataflow program on the other hand can compute these two values in parallel. The node that performs addition gets activated as soon as A and B are available.

$$A = x * y$$
$$B = y/20$$
$$C = A + B$$

(a) A simple program.



(b) Dataflow program.

Fig. 1: (a): A simple sequential program with three instructions (b): Its dataflow variant.

When run on the ZILLIQA's sharded network, each node in the dataflow program can be eventually attributed to a single shard or even a small subset of nodes within a shard. Hence the architecture is ideal for any MapReduce style computational tasks, where some node perform the mapping task while another node can work as a reducer to aggregate the work done by each mapper.

In order to facilitate the execution of a dataflow program,, ZILLIQA's smart contract language has the following features:

1) Operating over a virtual memory space shared globally across the entire blockchain.

2) Locking of intermediate cells in the virtual shared memory space during execution.
3) Checkpointing intermediate results during execution committed to blockchain.

B. Smart Security Budgeting

Apart form the benefits of parallelization offered from the dataflow computation model, ZILLIQA further provides a flexible security budgeting mechanism for computational sharding. This feature is enabled by sharding the computational resources in the blockchain network via an overlay above the consensus process. Computational sharding allows users of ZILLIQA and applications running on ZILLIQA to specify the sizes of consensus groups to compute for each of the subtasks. Each consensus group will then be tasked to compute the same subtask, and produce the results. The user specifies the condition on acceptance of the results, e.g., all in the consensus group must produce the same results, or $3/4$ of them must produce the same results, etc.

A user of the application running on ZILLIQA can budget how much she wants to spend on computing and security, respectively. In particular, a user running a particular deep learning application may spend more gas fee on running more of different neural network tasks than letting too many nodes repeating the same computation. In this case, she can specify a smaller consensus group for running each neural network computation. On the other hand, a sophisticated financial modeling algorithm that requires greater precision may task consensus groups of larger number of nodes to compute the critical portions of the algorithm to be more resilient against potential tampering and manipulation of the results.

C. Scalable Applications: Examples

ZILLIQA aims to provide a platform to run highly scalable computations in a multitude of fields such as data mining, machine learning and financial modeling to name a few. Since supporting efficient sharding of Turing-complete programs is very challenging, and there exist public blockchains that support Turing-complete smart contracts (e.g., Ethereum), ZILLIQA focuses on specific applications with requirements not met today.

1) **Computation with parallelizable computation load:** Scientific computing over large data is a typical example where one requires a large amount of distributed computing power. Moreover, most of these computations are highly parallelizable, examples include linear algebra operations on large matrices, search in the sea of huge amount of data and simulation on a large dataset among others. ZILLIQA provides such computing tasks an inexpensive and short turnaround alternative. Moreover, with the right incentive in place with computational sharding, and security budgeting ZILLIQA can be leveraged as a readily available and highly reliable resource for such heavy computation load.
2) **Train neural nets:** With the ever growing popularity and use cases of machine learning (in particular deep learning), it is imperative to have an infrastructure that allows deep learning models to train on large datasets. It is well known that training on large datasets is crucial to a model's accuracy. To this end, ZILLIQA's computational sharding and dataflow language will be particularly useful to build machine learning applications. It will serve as an infrastructure that may run tools like TensorFlow[1] by tasking groups of ZILLIQA nodes to independently perform different computations such as computing gradients, apply activation function, compute training loss, etc.
3) **Application with high complexity and high precision algorithms:** Different from the applications mentioned above, some applications, such as computations over financial models, may require high precision. Any minor deviation in one part of the computation may incur heavy losses in investments. Such applications can task consensus groups of larger number of nodes in ZILLIQA to allow them to cross-check the computational results of each other. The key challenge in offloading the computational tasks of such financial modeling algorithms to a public platform, such as ZILLIQA, is the concern for data privacy and intellectual property of the algorithms. To begin with, we envision certain well known portion of such computation can be placed to ZILLIQA for efficient and secure computation first, while the future research and development of ZILLIQA will further strengthen the protection of data privacy and intellectual property for such applications.

VIII. INCENTIVE LAYER

A. Token Supply

ZILLIQA has a finite supply of 21 billion ZILs. The smallest unit being 10^{-12} part of a ZIL. Each final TX-Block comes with a block reward that generates new tokens. The block reward will be spread over a period of 10 years decreasing over time. We aim to mine roughly 80% of the tokens in the first 4 years and the remaining 20% in the next 6 years. The token emission will be "smooth" in the sense that the block reward does not reduce drastically after a certain number of blocks. The smooth reduction in the block reward means that the network hashrate can be expected to be stable as the reward reduces over gradually over time.

After 10 years, we expect to have reached significant scale both in terms of the number of nodes in the network and users executing transactions. By then, we expect the market to have stabilized upon certain rates of transaction fees to fully sustain the running of the network without a need for new tokens entering the system as rewards.

B. Incentivizing Miners

Miners reach consensus on transactions, process them, perform computations as per the smart contract and update the global `state`. Miners are hence incentivized by requiring the sender of each transaction to pay some gas upfront.

[1] https://www.tensorflow.org/

Recall that each final TX-Block aggregates at most one micro block from each shard. Each micro TX-Block contains a `gas used` field that stores the total gas used by transactions in the block. Each final TX-Block also has a `gas used` field that is a sum of all the `gas used` field of each micro TX-Block. Once a TX-Block is proposed, the corresponding `gas used` and the block reward is almost equitably distributed among 1) the leaders of the shards who proposed one micro block and 2) the leader of the DS committee who proposed the final. In case the equitable distribution is not possible, the distribution is slightly biased towards the leader of the DS committee. Hence, if the reward is m and the total number of stake holders for a reward is n, then the leader of each shard gets $\lfloor \frac{m}{n} \rfloor$, while, the leader of the DS committee gets $m - n \cdot \lfloor \frac{m}{n} \rfloor$.

As the leader of each shard is changed once a new micro block is proposed, every member of the shard gets rewarded. Similarly, as the leader of the DS committee is changed after every final block, every member of the DS committee is also rewarded.

IX. Related Work

ZILLIQA is developed upon the ideas of Bitcoin-NG [18], collective signing (CoSi) [14], ByzCoin [15], Elastico [19] and OmniLedger [20].

Bitcoin-NG first proposed the idea to decouple leader election and his block proposals within Bitcoin. First, a leader is elected by mining a *keyblock* who can then mint many *microblocks* within the 10 minute block interval. The idea was further used in ByzCoin [15].

The idea of network and transaction sharding for a Bitcoin like system was first proposed in [19]. However, network/transaction sharding alone cannot solve the scalability issues as each shard needs to sign a TX-Block which makes the total number of signatures linear in the number of signers. This eventually results in a large block size and becomes a bottleneck during the broadcast/multi-cast.

Multisignatures [11] provides a solution to the above problem. CoSi [14] uses an EC-Schnorr multisignature scheme to design a protocol for collective signing. CoSi was proposed to work in a much less hostile environment than that of a public blockchain with byzantine nodes. With several significant enhancements we develop for the CoSi scheme, we derive a secure scheme and apply it to ZILLIQA.

Several other proposals have surfaced to sidestep the inherent scalability limitation of existing blockchain protocols, for instance, re-parameterizing the original Bitcoin protocol (e.g. increasing block sizes), moving as much computation off-chain (e.g. micropayment channels and lightening networks), creating hierarchy of blockchains (e.g. sidechains). None of these protocols directly make the blockchain protocol itself more scalable. ZILLIQA targets the heart of the scalability problem – its blockchain.

ZILLIQA can be seen as an extension of ByzCoin and OmniLedger with several security and performance optimizations. ZILLIQA also proposes a smart contract platform not available in ByzCoin/OmniLedger.

ZILLIQA's smart contract platform takes a different approach when compared with Ethereum. ZILLIQA's smart contract platform leverages the underlying sharding architecture and is based on dataflow programming. The advantages of dataflow programs are many: inherent concurrency and parallelism, easy to reason about their correctness, natural composability of functions and programs, etc.

X. Future Research Directions

Below, we discuss some ongoing and future directions of research to improve ZILLIQA.

State sharding. With increase in ZILLIQA's user base and its high transaction throughput would come the following challenge: How to efficiently handle the continuous influx of blocks that modify the global `state`. This is also referred to as *state sharding* in the literature. In essence, state sharding will alleviate full nodes from storing and receiving all blocks and transactions. This way it can further reduce the storage and communication load for nodes, and thus constitute another scaling-up factor to the throughput. However, it is non-trivial to design a secure and efficient state sharding scheme, as cross-shard communications arising from state sharding may outweigh the performance gains. More research needs to be done to address such additional complexities.

Secure Proof-of-Stake (SPoS). To the best of our knowledge, there has been no literature that proposes a secure PoS scheme, and thus we base ZILLIQA's building blocks on a PoW scheme. However, given the significant performance gain from PoS for consensus algorithms, it is worthwhile investigating further into the PoS paradigm, in search for a secure and efficient PoS scheme for ZILLIQA.

Storage pruning. We are currently exploring ways to securely prune the dated blocks stored on the blockchain to reduce the storage requirements and ease the joining process for new nodes. We may consider multi-grade storage, compression of blocks and transactions as possible solutions.

Cross-Chain support. ZILLIQA has every intention to complement other public blockchains and build a healthy ecosystem to provide end users a broad spectrum of platforms of choice for their applications. To this end, ZILLIQA will seek technical solutions to support gradual cross-chain communication and potentially enable cross-chain applications.

Privacy-preserving computation. It is desirable for several applications in particular (financial modeling applications) to have strong privacy and intellectual property protection when running on ZILLIQA. Solutions based on Oblivious RAM to hide access pattern on an encrypted data [21], ZK-SNARK [22] to hide the input to a program, and private function evaluation [23] to hide the contract's business logic are also being investigated.

XI. Conclusion

In this whitepaper, we have presented ZILLIQA's sharding architecture that allows the mining network to process transactions in parallel and reach high throughput. ZILLIQA also

comes with a unique smart contract platform that leverages the underlying sharing architecture and follows a dataflow programming paradigm. The new smart contract language is ideal for running computation-intensive task in an efficient manner.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf," 2008.

[2] E. Foundation, "Ethereum's white paper," https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[3] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186.

[4] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse Attacks on Bitcoin's Peer-to-Peer Network," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 129–144.

[5] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services," in *In ACM SIGACT News*, 2002, p. 2002.

[6] NIST, "Sha-3 standard: Permutation-based hash and extendable-output functions," 2015.

[7] B. Guido, D. Joan, P. Michaël, and V. A. Gilles, "The Keccak Reference ," 2011.

[8] C. Schnorr, "Efficient signature generation by smart cards," *J. Cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[9] C. Research, "SEC 2: Recommended Elliptic Curve Domain Parameters," 2000. [Online]. Available: http://www.secg.org/download/aid-386/sec2-final.pdf

[10] A. Poelstra, "Schnorr Signatures are Non-Malleable in the Random Oracle Model," 2014.

[11] S. Micali, K. Ohta, and L. Reyzin, "Accountable-subgroup Multisignatures: Extended Abstract," in *Proceedings of the 8th ACM Conference on Computer and Communications Security*, ser. CCS '01. New York, NY, USA: ACM, 2001, pp. 245–254.

[12] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," http://gavwood.com/paper.pdf, 2014.

[13] "Ethash," https://github.com/ethereum/wiki/wiki/Ethash, Accessed on June 27, 2017., version 23.

[14] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities "honest or bust" with decentralized witness cosigning," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 526–545.

[15] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 279–296.

[16] Arvind and D. E. Culler, "Annual review of computer science vol. 1, 1986," J. F. Traub, B. J. Grosz, B. W. Lampson, and N. J. Nilsson, Eds. Palo Alto, CA, USA: Annual Reviews Inc., 1986, ch. Dataflow Architectures, pp. 225–253.

[17] A. L. Davis and R. M. Keller, "Data flow program graphs," *Computer*, vol. 15, no. 2, pp. 26–41, Feb. 1982.

[18] I. Eyal, A. E. Gencer, E. G. Sirer, and R. van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, 2016, pp. 45–59.

[19] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 17–30.

[20] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger," *IACR Cryptology ePrint Archive*, vol. 2017, p. 406, 2017.

[21] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 299–310.

[22] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 781–796.

[23] P. Mohassel, S. S. Sadeghian, and N. P. Smart, "Actively Secure Private Function Evaluation," in *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, 2014, pp. 486–505.

[24] BSI, "Technical Guideline TR-03111 Elliptic Curve Cryptography," Federal Office for Information Security, Tech. Rep., 01 2012.

[25] D. J. Bernstein, "Multi-user Schnorr Security, Revisited," *IACR Cryptology ePrint Archive*, vol. 2015, p. 996, 2015. [Online]. Available: http://eprint.iacr.org/2015/996

[26] M. Michels and P. Horster, "On the Risk of Disruption in Several Multiparty Signature Schemes," in *Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology*, ser. ASIACRYPT '96. London, UK, UK: Springer-Verlag, 1996, pp. 334–345.

APPENDIX A
SCHNORR DIGITAL SIGNATURE ALGORITHM

A. EC-Schnorr (Single Signer) Scheme

EC-Schnorr works on a group where the discrete logarithm is hard [8], [24], [25]. ZILLIQA uses the elliptic curve group defined over the popular `secp256k1` curve. We denote by $C := (p, G, n)$ the set of parameters that define the group, where p is a prime number that specifies the underlying field $\mathbb{F}_p$, G is the base point on the curve and n (a prime) is the order of G. EC-Schnorr also requires a cryptographic hash function H that we instantiate with `SHA3-256` [6].

EC-Schnorr is a set of three algorithms KeyGen, Sign and Verify that we present in this section. In the algorithms below, for any scalar x and a point Q, we denote the scalar multiplication by $[x]Q$.

1) KeyGen(C): The algorithm takes the curve parameters C and returns a pair of public (pk) and private (sk) keys.

KeyGen($C = (p, G, n)$)

1. Choose $sk \xleftarrow{\$} [1, n-1]$,
2. Set $pk \leftarrow [sk]G$,
3. **return** (pk, sk).

2) Sign(C, pk, sk, m): This algorithm is run by the signer. It takes the curve parameters C, a public key and a private key pair (pk, sk) and a message to sign $m \in \{0,1\}^*$. It returns a signature σ.

Sign(C, pk, sk, m)

1. Choose $k \xleftarrow{\$} [1, n-1]$,
2. Set $Q \leftarrow [k]G$,
3. Set $r \leftarrow H(Q||pk||m) \bmod n$,
4. **If** $r = 0$ **Goto** 1.
5. Set $s \leftarrow k - r \cdot sk \bmod n$,
6. **If** $s = 0$ **Goto** 1.
7. Set $\sigma \leftarrow (r, s)$,
8. **return** σ.

3) Verify(C, σ, pk, m): This algorithm is run by a verifier who wishes to check the validity of a signature. It takes the curve parameters C, a signature σ, a public key pk and a message m. It returns 1 if the signature is valid for m under pk, or else returns 0.

Verify(C, σ, pk, m)

1. Parse $(r, s) \leftarrow \sigma$,
2. **If** $r, s \notin [1, n-1]$ **return** 0.
3. Set $Q \leftarrow [s]G + [r]pk$,
4. **If** $Q = \mathcal{O}$(neutral point) **return** 0.
5. Set $v \leftarrow H(Q||pk||m) \bmod n$,
6. **If** $v = r$ **return** 1, else **return** 0.

B. EC-Schnorr Multisignature Scheme

1) Setting & Assumptions: EC-Schnorr can also be used as a multisignature scheme [11]. In a multisignature scheme, we have T signers: $P_1, \ldots, P_T$, an aggregator and a verifier. The signers wish to jointly sign a message m. The aggregator plays the role of a facilitator and aggregates the signatures sent by each individual signer. The verifier verifies the aggregated signature. The role of aggregator and the verifier can be played by the same entity.

Each signer P_i has her own public private key pair (pk_i, sk_i) for EC-Schnorr single signer scheme. We denote by $P = \{pk_1, \ldots, pk_T\}$ the set of all public keys. We also assume a public message m_p known to every entity. The message m_p may be specific to the application scenario and make take the following form: `I know the private key for my public key for the session id: XXXX`. The purpose of this message is to defeat certain known attacks on the scheme [26].

2) Multisignature Protocol: Multisignature is an interactive protocol between signers, the aggregator and the verifier (see Figure 2 for a schematic representation). The protocol has six steps as described below.

1) **(One-Time) Identity Setup**: This step is run between each participant and the verifier. At the start of the protocol, each signer P_i if not currently involved in another signing protocol generates an EC-Schnorr signature σ_i on the message m_p. P_i then sends (σ_i, pk_i) to the verifier. The verifier then performs the following checks:
   a) Check if $pk_i \in P$. If the check fails, the verifier aborts.
   b) Check if each σ_i is a valid EC-Schnorr signature on m_p for pk_i, by invoking Verify(C, σ_i, pk_i, m_p). Verifier aborts if any of these signature verifications returns 0. If all the signatures are valid, then the protocol proceeds to the next step.

   If the verifier does not receive σ_i for every pk_i in P, she also aborts. To record whether/or not she received a signature from P_i, she uses a bitmap $Z[1, \ldots, |P|]$. Identity Setup is a one-time process followed by any number of the next steps. Only if the set up successfully terminates, the next steps of the protocol can start.
2) **Commitment Generation**: Each signer P_i then choses a random $k_i \xleftarrow{\$} [1, n-1]$ and computes $Q_i = [k_i]G$. Recall that G is the base point on the elliptic curve and n is the order of G. P_i then sends Q_i to the aggregator.
3) **Challenge Generation**: The aggregator first computes the aggregated keys: $pk = \sum_{pk_i \in P} pk_i$ for keys in P. She also computes $Q = \sum_i Q_i$ for Q_i's received in the previous step. She then computes $r \leftarrow H(Q||pk||m) \bmod n$ and sends (r, Q, pk) to each P_i.
4) **Response Generation**: Each signer P_i first checks the integrity of r received previously. This is done by recomputing $H(Q||pk||m)$ and checking if it is equal to the received r. If the check fails then P_i aborts the protocol or else generates $s_i \leftarrow (k_i - r \cdot sk_i) \bmod n$ and sends s_i to the aggregator.
5) **Response Aggregation**: Aggregator computes the aggregated response $s = \sum_i s_i \bmod n$ and builds an aggregated signature $\sigma = (r, s)$. She then sends (m, σ) to the verifier.
6) **Signature Verification**: Verifier now checks whether the signature is valid. She performs the following steps:
   a) Aggregate the public keys in P as pk'.
   b) Check if σ is a valid EC-Schnorr signature on m for the public key pk' by invoking Verify(C, σ, pk', m). Returns the output of Verify.

APPENDIX B
MULTISIGNATURE FOR PBFT

Classical EC-Schnorr multisignature protocol as described in Appendix A requires the participation of all the participants. Hence, we cannot directly use it in the PBFT setting, where, we only require that the message be signed by at least $\frac{2}{3}n + 1$ nodes in the committee. In this section, we present a tweak to this protocol inspired from [14]. The tweak consists in maintaining two bitmaps that record the participation in the protocol. The modified protocol is given in Figure 3. Below, we briefly present the protocol.

1) **(One-Time) Identity Setup**: This step is exactly the same as in the classical EC-Schnorr multisignature protocol as presented in Appendix A. Only if the set up successfully terminates, the next steps of the protocol can start.
2) **Commitment Generation**: This step is similar to the classical EC-Schnorr multisignature protocol. The only difference being that each participant P_i also sends its public key pk_i along with a Q_i to the aggregator.
3) **Challenge Generation**: At this step the aggregator maintains a bitmap $B_Q[1, \ldots, |P|]$ initialized to 0. For every (Q_i, pk_i) received in the previous step, the aggregator sets $B_Q[i]$ to 1. The aggregator waits for a stipulated time to handle network propagation delay and then computes the following:
   a) The aggregated keys: $pk \leftarrow \sum_{pk_i \in P} pk_i \cdot B_Q[i]$, i.e., she adds the public keys for which she received a Q_i.
   b) She also computes $Q \leftarrow \sum_{i:B_Q[i]=1} Q_i$ for Q_i's received in the previous step.

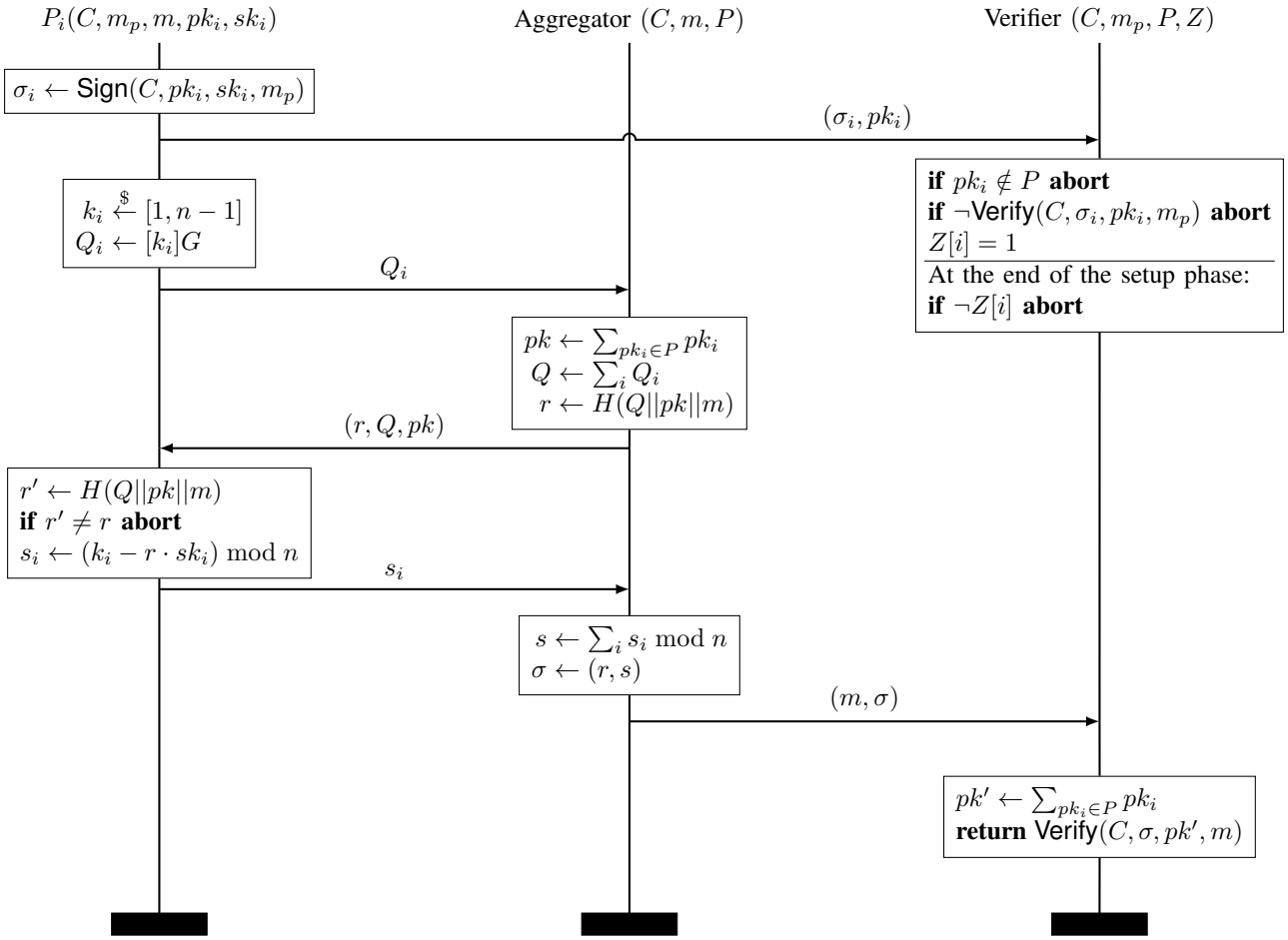


Fig. 2: Multisignature using EC-Schnorr. Verifier stores a bit map $Z[1, \ldots, |P|]$, where each entry is initialized to 0.

c) She then computes $r \leftarrow H(Q||pk||m) \bmod n$ and sends (r, Q, pk) to each P_i.

4) **Response Generation**: This step is similar to the classical EC-Schnorr multisignature protocol. The only difference being that each participant P_i also sends its public key pk_i along with a s_i to the aggregator.
5) **Response Aggregation**: At this step the aggregator maintains a bitmap $B_s[1, \ldots, |P|]$ initialized to 0. For every (s_i, pk_i) received in the previous step, the aggregator checks if the received s_i is valid by computing $Q_i' \leftarrow [s_i]G + [r]pk_i$ and then verifying if Q_i' is equal to the received Q_i. If the two values are equal then she sets $B_s[i]$ to 1. This step allows to detect participants who send an arbitrary value of s_i and attempt to mount a DoS attack. The aggregator then waits for a stipulated time to handle network propagation delay and then computes the following:
   a) If the two bitmaps B_Q and B_s are equal, which means the same set of participants sent messages to the aggregator in the commitment generation and response generation steps, then the aggregator computes the aggregated response $s = \sum_i s_i \bmod n$ and builds an aggregated signature $\sigma = (r, s)$. She then sends (σ, m, B_Q) to the verifier.
   b) If the two bitmaps are not equal, which means a participant sent a Q_i but not the corresponding s_i, then the aggregator computes the set-theoretic difference of B_Q and B_s, i.e., the set of public keys $pk_i \in P$ for which the aggregator received a Q_i but not the corresponding s_i. The corresponding set of public keys can then be blacklisted. The aggregator re-initializes B_s to 0 and computes the intersection between B_Q and B_s and stores it in B_Q. Finally, it repeats the protocol starting from the challenge generation step.
6) **Signature Verification**: The verifier first checks if the signature was generated by at least $\frac{2}{3}|P| + 1$ participants and then checks whether the multisignature is valid. The rest of the steps are same as in the classical EC-Schnorr multisignature protocol.

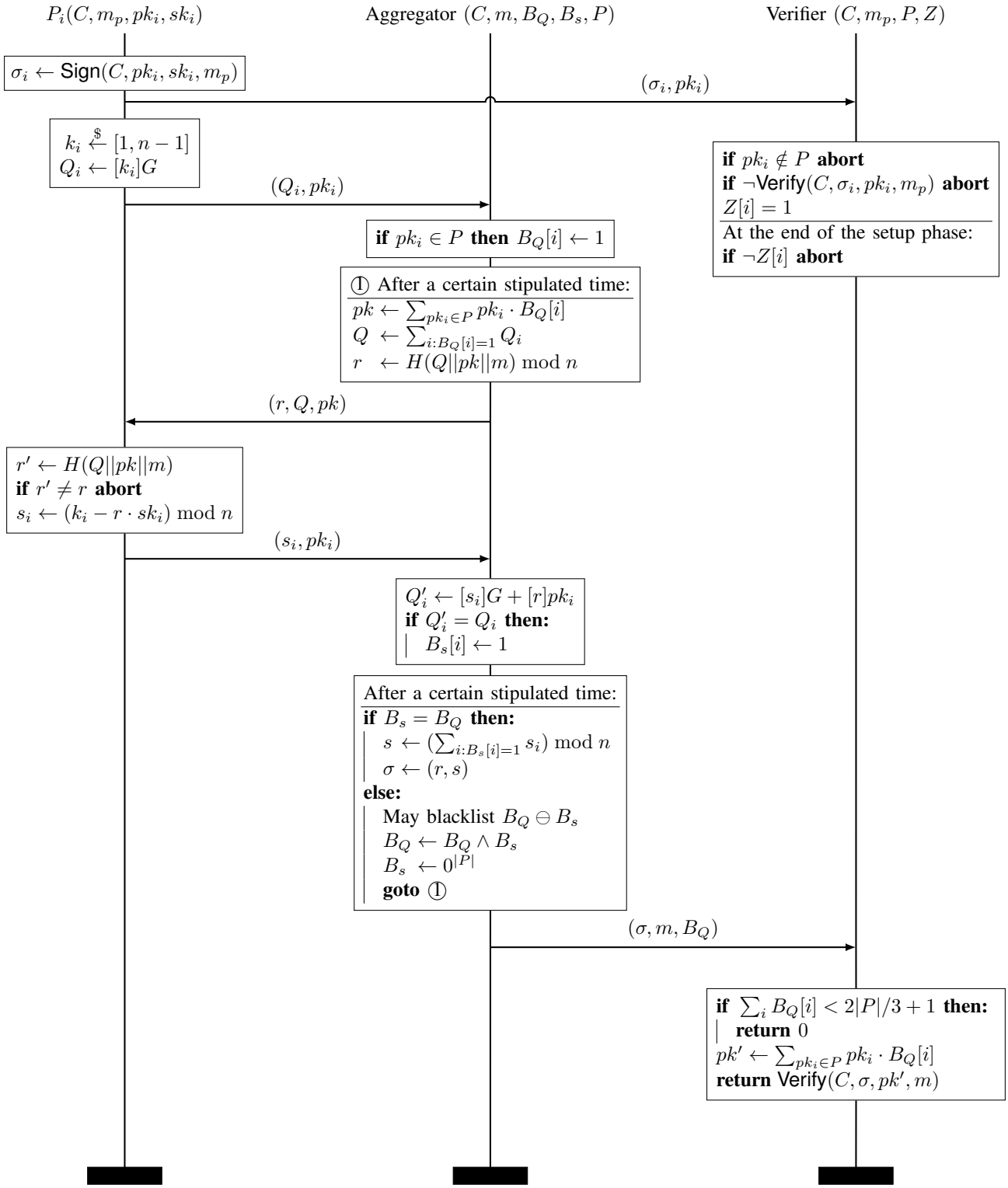


Fig. 3: EC-Schnorr multisignature variant used in PBFT. The leader of each committee plays the role of the aggregator. The aggregator maintains two bitmaps $B_Q[1, \ldots, |P|]$ and $B_s[1, \ldots, |P|]$, while the verifier stores a bit map $Z[1, \ldots, |P|]$. The entries of the bitmaps are initialized to 0. $B_Q \ominus B_s$ returns a bitmap that represents the set-theoretic difference of B_Q and B_s, i.e., it represents the set of public keys $pk_i \in P$ for which the aggregator received a Q_i but not the corresponding s_i.