CYPHERIUM YELLOW PAPER

CYPHERIUM DEVELOPMENT TEAM

1. Introduction

Blockchain is the underlying technology that has made possible Bitcoin and all subsequent cryptocurrencies. A blockchain is a distributed database that records all transactions or digital events that have been executed and shared among participating parties. Each transaction verified by the majority of participants of the system. It contains a record of every single transaction. Bitcoin is the first and most popular cryptocurrency and example of blockchain technology, which first came to light when a person or collective named 'Satoshi Nakamoto' published a white paper on "Bitcoin: A peer to peer electronic cash system" Sprankel [2013] in 2008. Blockchain technology distributes records of its transactions throughout its network thus making the data incorruptible. Anything of value, such as land assets, cars, etc. can be traced and transacted through blockchain technology.

A blockchain based application called "Distributed ledger" is gaining more and more popular today. Various decentralized applications on top of the blockchain platform were developed for the past few years. While many people think blockchain technology is the new generation of technological revolution, there do exists a few critical limitations that need to be solved before this new technology can start replacing the existing ones such as the centralized internet based applications.

The first issue with one of the most popular blockchain application, the bitcoin, is that it can only perform very simple token transactions. One person may transfer a certain amount of bitcoin token to anther person, using the underlying blockchain and the so-called "proof of work" consensus mechanism to reach to a decentralized agreement among all the nodes. However, this is the only functionality that bitcoin can offer. It is impossible to use this simple functionality to replace the extremely well designed internet.

As a result, the distributed smart contract platform was invented by various groups, among which the most famous one is Ethereum. Comparing with Bitcoin, Ethereum is more like a computing platform. It has all the features Bitcoin has, and third party developers can also write smart contract code and deploy to the Ethereum network to accomplish fairly complex business logics.

However, both Bitcoin and Ethereum are facing an even more critical problem. Due to the natural of the decentralized consensus algorithm such as proof of work, and a few security concerns, the total transactions per seconds for Bitcoin and Ethereum are only 6 to 15. On the other hands, the more traditional centralized applications such as PayPal or VISA can support hundreds of thousands of transactions per seconds. Moreover, single Bitcoin transaction will take up to one hour to finally confirm, due to some security concerns; Ethereum is a little better, but it will also take up to multiple minutes to complete. When the network is busy, it will take hours to complete, which is absolutely unacceptable.

To avoid Bitcoin's sluggish fate Ethereum has decided to abandon Proof-of-Work for Proof-of-Stake. However, Proof-of-Stake is not without its drawbacks. Firstly there is the so-called "nothing-at-stake" problem, which undermines network's ability to determine which nodes are honest and which are bad actors. But more fundamentally, Proof-of-Stake runs the risk of centralization. Instead of relying on cryptography and math as in Proof-of-Work, Proof-of-Stake uses power wealth to determine an actor's trustworthiness.

For these reasons, this yellow paper presents a new blockchain that addresses the present shortcomings of current public blockchain infrastructures, called Cypherium. Cypherium is designed to mainly solve the performance bottleneck of the current blockchain applications, while at the same time still maintain the same security level. Key features of Cypherium's design include double-chain consensus, federated architecture and separate sandboxes for test and production smart contract. Chapter 2 will describe the overall architecture of Cypherium, which is followed by Chapter 3 the detailed design of each component, including our PBFT-based consensus, reconfiguration, Java-based virtual machine, etc. Then, next chapter will briefly summarize some limitations of the current version of Cypherium and some future work, and finally the paper is concluded in Chapter 5.

2. Architecture

Cypherium blockchain consists of two types of blocks: key blocks and transaction blocks. Key blocks determine consensus group membership via Proof-of-Work. Consensus groups collectively decide which transactions will be included in transaction blocks. Transaction blocks contain records of transactions to be committed into the permanent ledger and do not affect consensus group membership; they are smaller and created more often. These features distinguish Cypherium from Ethereum and many other blockchain protocols where each block contains transactions and also determines new miner. Another distinguishing factor is that in Cypherium each block is created by a committee, i.e., a set of nodes, not by a single miner. Each key block removes the oldest node from the current committee and replace it with a new one. Committee members regularly run a variation of Practical Byzantine Fault Tolerance protocol (PBFT) to reach consensus about the contents of each new block. Once consensus is reached, the block is sent to the whole network and can be considered irreversible.

Following this short introduction, we will discuss model details in the next sections of this chapter. Some of the notions are borrowed from well-known models, while others are specific and have to be understood for further reading.

2.1. **World State.** All data issued to the blockchain must be deemed either valid or invalid. The data verification proceeds against all previously issued data, especially blocks. A sequence of blocks forms a World State, which can be used for verification. Basically by mapping between addresses to their corresponding account states.

An address is a 160-bit identifier, and an account state holds information about that account. In a simple case, it would be just the account balance, but in Cypherium we have smart contracts so that address can hold various kinds of data.

All these addresses form a Merkle Patricia tree (trie), an immutable data structure that gives access to any previous version knowing only the root hash of the needed version. Also, the root node of this structure is cryptographically dependent on all internal data, and thus, its hash can be used as a secure identity for the entire system state.

Cypherium's World State is designed similar to that of Ethereum, so for a fuller understanding, you might read the corresponding chapter of the Ethereum's yellow paper.

2.2. **Committee. A leader of the Committee.** In Cypherium, there exists a set of participants who drive the consensus algorithms and verify issued data (e.g., transactions and blocks). This set is called a *committee.* Every participant is identified by a unique public key. Public keys involved in the committee are also called *members* of a committee.

A node may be presented in the committee by several public keys. Consider committee M.

Members of the committee can dynamically change over time, but their amount is constant. The size of M is $3f+1$ and f is constant and predefined. At least $2f+1$ members of the committee are supposed to be available and honest. This requirement arises from PBFT (Section 2.3) underlying algorithm.

There is a dedicated member of the committee which is called a *leader.* A leader drives a *reconfiguration* process (Section 2.4) and issues new *transaction blocks* (Section 2.7), while other committee members serve as witnesses and verify that the leader behavior, as well as issued data, was correct.

2.3. **PBFT consensus.** PBFT (Practical Byzantine Fault Tolerance) algorithm is a form of state machine replication: the blockchain is modeled as a state machine that is replicated across different nodes. Each node maintains the blockchain world state and accepts new incoming data.

So, committee members are involved in the PBFT algorithm. A leader is the only one node inside a committee who has the right to propose changes to the state. The other nodes can either agree or disagree with them. After the leader's proposal, a change has to pass three rounds for being applied to state: *propose*, *prepare* and *commit* rounds. These rounds are driven collectively by committee members. After each round that the data has passed successfully, each node obtains a *certificate*, or in other words, a *collective signature*, which proves this fact. Hence, every change of the state is proven by a certificate, and moreover, every change is *irreversible*, meaning once the change is applied, it can't be undone.

A slightly modified version of the PBFT algorithm is a foundation of Cypherium, and it is introduced briefly in this section to simplify the understanding of further aspects of the network. There is a separate chapter which describes PBFT in more details.

2.4. **Reconfiguration.** Here we walk through the process of updating a committee.

We define the state of a committee as a *configuration* and we enumerate configurations using non-negative integers. A set of committee members in a configuration number c will be denoted as $M(c)$.

Reconfiguration is a process in which the oldest committee member is replaced with another different participant. In this case, a number of a configuration is increased by 1 and:

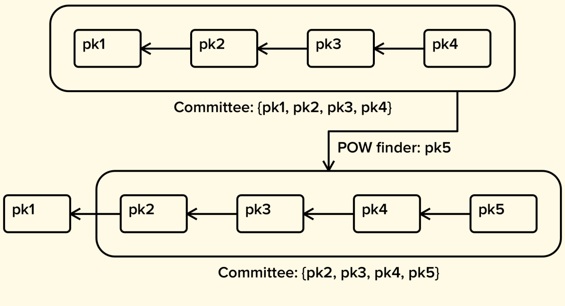
$$M(c+1) = M(c) \setminus P_{oldest} \cup P \tag{1}$$

Where P is a public key of a participant joining the committee; P_{oldest} is a public key of the member who has joined the committee first; c is a number of the configuration. After joining the committee, a participant may become a new leader.

If a node wants to join a committee, it must solve a difficult computation problem, similar to that which is used in Bitcoin's Nakamoto consensus. The node has to find *Proof of Work* nonce η, which satisfies

$$H(puzzle_c \oplus P \oplus \eta) \leq T \tag{2}$$

Where η is a found nonce; T is a threshold which regulates the frequency of reconfiguration; Usually, T may be chosen to cause one reconfiguration in approximately 10 minutes; Also, it can be changed over time because of the gain of the hash power of participants; $puzzle_c$ is hash of data which reflects the up-to-date state of the system; We will discuss the $puzzle_c$ in more details in Section 3; H is the 256 Keccak hash function; $\oplus$ stands for concatenation.


pk1
pk2
pk3
pk4
Committee: {pk1, pk2, pk3, pk4}
POW finder: pk5
pk1
pk2
pk3
pk4
pk5
Committee: {pk2, pk3, pk4, pk5}

Reconfiguration process

2.5. **Key Block.** After a node has found an appropriate nonce, in order to be included in a committee, it must send a *key block* with the found nonce and other auxiliary information.

A key block consists of a header and a body.

The header includes:

version: The correct version of the protocol.

parentHash: The Keccak 256-bit hash of the parent key block header.

difficulty: A scalar value corresponding to the difficulty level of this block; T from (2).

- **number:** A scalar value equal to the number of ancestor transaction blocks. The genesis block's number is zero.
- **time:** A scalar value equal to the relevant output of Unix's time() at this block's inception.
- **extra:** An arbitrary byte array containing data relevant to this block. Must have a size of 32 bytes or fewer.
- **nonce:** A 64-bit value which, combined with $puzzle_c$ and a miner's public key, proves that a sufficient amount of computation has been carried out on this block; η from (2).

The body includes:

- **signature:** The bytearray, a collective signature from the *commit* round of PBFT, where it is decided if a new member can join the committee.
- **mask:** The bytearray consisting of $3f + 1$ elements, where 1 value stands on the position i if i-th member of a committee is involved in a collective signature.
- **leaderPubKey:** The public key of a participant who found the PoW nonce and is going to become a new leader; P from (1) and (2).
- **outPubKey:** The public key of the oldest committee member who is going to leave the committee; P_{oldest} from (1).

2.6. **Transaction.** Transactions in Cypherium is similar to those of Ethereum. There are two types of transactions: the first type results in message calls and the second one results in the creation of new accounts with associated code (known informally as 'contract creation'). Both types specify a number of common fields:

- **version:** An Integer value, a version of the protocol.
- **sender:** The public key of a transaction sender.
- **nonce:** A scalar value equal to the number of transactions sent by the sender; formally T_{n}.
- **gasPrice:** A scalar value equal to the number-total of money to be paid per unit of *gas* for all computation costs incurred during the transaction execution; formally T_{p}.
- **gasLimit:** A scalar value equal to the maximum amount of gas that should be used in the process of the transaction execution. It is paid in advance, before any computation is performed, and may not be increased later; formally T_{g}.
- **recipient:** The 160-bit address of the message call's recipient or, for a contract creation transaction, $\varnothing$, used here to denote the only member of $\mathbb{B}_0$; formally T_{t}.
- **amount:** A scalar value equal to the number-total of money to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account; formally T_{v}.
- **v, r, s:** Values corresponding to the signature of the transaction and used to determine the transaction sender; formally T_{w}, T_{r} and T_{s}. It is discussed in more detail in Appendix **??**.

Additionally, a contract creation transaction contains:

- **init:** An unlimited size byte array specifying the CVM-code for the account initialization procedure, formally $T_{\mathbf{i}}$.

init is an EVM-code fragment; it returns the **body**, the second fragment of code that executes each time an account receives a message call (either through a transaction or due to the internal execution of code). **init** is executed only once at account creation and gets discarded immediately after that.

By contrast, a message call transaction contains:

- **data:** An unlimited size byte array specifying the input data of the message call, formally $T_{\mathbf{d}}$.

Appendix **??** specifies the function, S, which maps transactions to the sender and happens through the ECDSA of the SECP-256k1 curve, using the hash of the transaction (excepting the latter three signature fields) as the datum to sign. For the present, we simply assert that the sender of a given transaction T can be represented with $S(T)$.

(3)
$$L_T(T) \equiv \begin{cases} (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, T_{\mathbf{i}}, T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{if } T_{\mathrm{t}} = \varnothing \\ (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, T_{\mathbf{d}}, T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{otherwise} \end{cases}$$

Here, we assume all components are interpreted by the RLP as integer values, with the exception of the arbitrary length byte arrays $T_{\mathbf{i}}$ and $T_{\mathbf{d}}$.

$$(4)\qquad \begin{array}{lllll} T_{\mathrm{n}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{v}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{p}} \in \mathbb{N}_{256} \quad \wedge \\ T_{\mathrm{g}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{w}} \in \mathbb{N}_{5} & \wedge & T_{\mathrm{r}} \in \mathbb{N}_{256} \quad \wedge \\ T_{\mathrm{s}} \in \mathbb{N}_{256} & \wedge & T_{\mathbf{d}} \in \mathbb{B} & \wedge & T_{\mathbf{i}} \in \mathbb{B} \end{array}$$

where

$$(5)\qquad \mathbb{N}_{\mathrm{n}} = \{P : P \in \mathbb{N} \wedge P < 2^n\}$$

The address hash $T_{\mathbf{t}}$ is slightly different: it is either a 20-byte address hash or, in the case of being a contract-creation transaction (and thus formally equal to $\varnothing$), it is the RLP empty byte sequence and thus the member of $\mathbb{B}_0$:

$$(6)\qquad T_{\mathbf{t}} \in \begin{cases} \mathbb{B}_{20} & \text{if} \quad T_{\mathrm{t}} \neq \varnothing \\ \mathbb{B}_0 & \text{otherwise} \end{cases}$$

2.7. **Transaction Block.** Besides key block, Cypherium also utilizes *transaction block*. The goal of this type of block is to carry sent transactions. Compared to key blocks, transaction blocks are issued much more frequently: a block per approximately 20 seconds.

The transaction block is the collection of relevant pieces of information (known as the block *header*), H, together with information corresponding to the comprised transactions, $\mathbf{T}$.

- **parentHash:** The Keccak 256-bit hash of the parent transaction block's header, in its entirety; formally H_{p}.
- **stateRoot:** The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalizations applied; formally H_{r}.
- **transactionsRoot:** The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block; formally H_{t}.
- **receiptsRoot:** The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block; formally H_{e}.
- **logsBloom:** The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of

each transaction in the transactions list; formally H_{b}.

number: A scalar value equal to the number of ancestor transaction and key blocks. The genesis block has a number of zero; formally H_{i}.

gasLimit: A scalar value equal to the current limit of gas expenditure per block; formally H_{l}.

gasUsed: A scalar value equal to the total gas used in transactions in this block; formally H_{g}.

timestamp: A scalar value equal to the relevant output of Unix's time() at this block's inception; formally H_{s}.

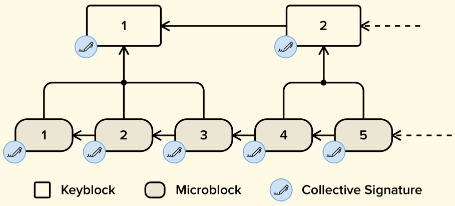
extraData: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer; formally H_{x}.

keyblockHash: The Keccak 256-bit hash of the header of the last applied key block.

keySignature: The bytearray, a collective signature from the *commit* round of PBFT, where it was decided that this block is correct.

isKeyReward: A boolean value indicating whether a block is formed of transactions curring rewards for previous committee members.

exceptions: The bytearray consisting of $3f+1$ elements, where 1 value stands on the position i iff i-th member of a committee is **not** involved to collective signature *keysignature*.



Transaction and key blocks

3. Consensus algorithm

At a high level, Cypherium runs a Byzantine consensus protocol, driven by a committee. The acting committee commits transactions into a linearizable log using a modified version of PBFT (Castro et al. [1999]). The log forms a World State of the system. We say each consensus decision fills a *slot* in the state. A consensus decision can be either a batch of transactions or a reconfiguration event. The first type of decision is analogous to a block in Bitcoin. The second type records the membership change in the committee. The slot number is equal to a number of transaction and key blocks committed before.

Cypherium consensus algorithm relies on Solida consensus algorithm (Abraham et al. [2016]), which relies on PBFT algorithm, where participants of a committee may be either honest or Byzantine. Cypherium's design is also influenced by ByzCoin (Kogias et al. [2016]). Honest participants always follow the predefined protocol and also they work within the frame a model, which is strictly defined and the same for all participants. Notice, that an honest node which behaves properly, but not satisfying the model due to some external factors treated as a Byzantine node.

3.1. **Model.**

Before describing the protocol, we would like to limit a model where the system is considered.

3.1.1. *Network.* We consider a permissionless setting in which nodes use their public keys as pseudonyms and can join or leave the system at any time. Nodes are connected to each other in a peer-to-peer network with a small diameter.

We adopt the network model of Pass et al. (Pass et al. [2017]): bounded message delay of Δ that is known a priori to all participants. Note that this is the standard synchronous network model in distributed computing, though Pass et al. call it "asynchronous networks." What they really meant was that Bitcoin does not behave like a conventional synchronous protocol. In a conventional synchronous protocol, participants move forward in synchronized rounds of duration Δ, but Bitcoin does nothing of this sort and has no clear notion of rounds.

So, whenever a participant sends a message to another participant, the message is guaranteed to reach the recipient within Δ time. For convenience, we define Δ to be the end-to-end message delay bound. If there are multiple hops from the sender to the recipient, Δ is the time upper bound for traveling all those hops. Our protocol uses computational puzzles, i.e., PoW. Similar to Bitcoin, the difficulty of the puzzle is periodically adjusted so that the expected time to find a PoW stays at D. D should be set significantly larger than Δ.

Our synchronous network model may raise a few immediate questions, which we discuss below.

Is the bounded message delay assumption realistic for the Internet? It depends a lot on the parameter Δ. With a conservative estimate, say $\Delta = 5$ seconds, and Byzantine fault tolerance, the assumption may be believable. Fault tolerance also helps here. Participants who experience slow networks and are unable to deliver messages within Δ are considered Byzantine. So, in Cypherium we require the assumption that adversarial participants and "slow" participants collectively control no more than about 1/3 of the mining power.

Why do we use PBFT if the network is synchronous? There are protocols that can tolerate $f < n/2$ Byzantine faults in a synchronous network (Katz and Koo [2009], Ren et al. [2017]). By requiring $f < n/3$, PBFT preserves safety under asynchrony. But this seems unnecessary given that we assume synchrony. We adopt PBFT (with modifications) because it is an established protocol that provides responsiveness. Most protocols that tolerate $f < n/2$ are synchronous and not responsive because they advance in rounds of duration Δ. If the actual latency of the network is better than Δ — either because the Δ estimate is too pessimistic (the network is faster in the common case), or the network speed has improved over time — a responsive protocol would offer better performance than a synchronous one. In other words, we opt to use an asynchronous protocol in a synchronous setting, essentially abandoning the strength of the bounded message delay guarantee, in order to run as fast as the network permits. But like all PoW-based blockchains, we rely on synchrony for safety in the worse case.

3.1.2. *Adversary consideration.* As we've mentioned, honest participants always follow the prescribed protocol and are capable of delivering a message to other honest participants within Δ time. We will consider the worst case when Byzantine participants are controlled by a single adversary and can thus coordinate their actions. At any time, Byzantine participants collectively control no more than ρ fraction of the total computation power.

We assume it takes time for the adversary to corrupt an honest participant. It captures the idea that it takes time to bribe a miner or infect a clean host. Most committee-based designs (Decker et al. [2016], Kogias et al. [2016], Pass and Shi [2017]) require this assumption. Otherwise, an adversary can easily violate safety by corrupting the entire committee, which may be small compared to the whole miner population. This is formalized as a delayed adaptive adversary by Pass and Shi (Pass and Shi [2017]). Specifically, we assume that even if the adversary started corrupting a member as soon as it emerged with a PoW, by the time of corruption the member would have already left the committee. Whether this assumption holds in practice remains to be examined.

3.2. **The Protocol.**

3.2.1. *Basic notions.* Firstly we describe the protocol outside reconfiguration, i.e., when no miner has found a PoW for the current puzzle. Herienafter we'll see that this part consists of two sub-protocols: *steady state* under a stable leader (Section 3.2.2), and *view change* to replace the leader (Section 3.2.3). Committee members monitor the current leader, and if the leader does not show progress in some time, members support the next leader. The new leader has to learn what values have been proposed, and possibly re-propose those values.

When a new miner finds a PoW, the system switches to the *external leader* for reconfiguration. The successful miner would act as the leader L for reconfiguration and try to be elected onto the committee. When the L's reconfiguration proposal is committed (i.e., becomes a consensus decision), reconfiguration is finished, and the system starts processing transactions under the new committee with L being the leader. Note that at this point L becomes a committee member, so the system has seamlessly transitioned back to internal leaders.

Each configuration can have multiple lifespans, and each lifespan can have many views. We use consecutive integers to number configurations, lifespans within a configuration, and views within a lifespan.

The lifespan number is the number of PoWs (honest) that committee members have seen so far in the current configuration. Intuitively, when a new PoW is found, the lifespan of the previous PoW ends.

For this purpose, we will associate each leader with a configuration-lifespan-view tuple (c, e, v) and rank leaders by c, e, and v in this order of priority. Each (c, e, v) tuple also defines a steady state where the leader of that view, denoted as $L(c, e, v)$, proposes values for members to commit. The view tuple and the leader role are managed as follows.

- Upon learning that a reconfiguration event is committed into the ledger, a member M increases its configuration number c by 1 and resets e and v to 0. $L(c, 0, 0)$ is the member newly added by the previous reconfiguration consensus decision.
- Upon receiving a new PoW for the current configuration, a member M increases its lifespan number e by 1 and resets v to 0. M now supports the finder of the new PoW as $L(c, e, 0)$ $(e \geq 1)$.
- Upon detecting that the current leader is not making progress (timed out), a member M increases its view number v by 1. $L(c, e, v)$ $(v \geq 1)$ is the l-th member on the current committee, by the order in which they joined the committee, where $l = (H(c, e) + v) \mod n$, and H is a hash function.

As long as the protocol ensures safety so far, members agree on the identities of these leaders. For views of the form $(c, e, 0)$ with $e \geq 1$, members may not agree on the leader's identity if they receive multiple PoWs out of order. It is not a problem for safety because the situation is similar to having an equivocating leader and a BFT protocol can tolerate leader equivocation. However, contending leaders in a view may prevent progress. Luckily, members will time out and move on to views (c, e, v) with $v \geq 1$. Unique leaders in those views will ensure liveness.

It is crucially that our protocol forbids pipelining to simplify reconfiguration: a member M sends messages for a slot s only if it has committed all slots up to $s - 1$. Let $M(c, e, v, s)$ be the set of members who are currently in view (c, e, v) and working on a slot s. Each honest member M can belong to only one set at a time.

Before we start to describe a protocol, let us introduce you several notations. Throughout the protocol we send other nodes messages, so let's describe basic notions we use for these messages:

$$\circledast(Tag, a_1, a_2, ..., a_k, \sigma_K)$$

By this tuple we mean message with fields $a_1, a_2, ..., a_k$, and a message itself has a *tag* Tag. Tags are used to distinguish messages, also a Tag determines fields which will follow it.

$\circledast$ stands for a serialized message broadcasted over the network to all committee members. We also can use designation $\circledast_X$ which means a message is sending to the X node.

σ_K is a signature of all previous elements of the tuple that was made with a private key corresponding to the K node, including the Tag.

3.2.2. *Steady State.* In a Steady State a committee is defined and doesn't change over time:

- a set of nodes which form the committee is known and doesn't change. A public key and a public commitment are known for each node;
- a committee leader is known, it's a member of the committee, and it doesn't change.

During a steady state, the leader proposes a block to be added to the Ledger. Committee members validate and approve the proposed block, and if the block was accepted into the Ledger, we say that the block was committed. Similarly to the plain PBFT protocol, the process of committing each block into history consists of several phases, where nodes exchange messages and form a consensus during each phase. A witness of the reached consensus is called a certificate.

In a plain PFBT protocol a certificate consists of $2f + 1$ similar messages signed by different nodes. Such messages are broadcasted from node to node, and every node can form its own version of a certificate.

In Cypherium a steady state is similar to plain PFBT except for two main differences:

- there are no checkpoints, and instead of periodical checkpoints we send a *notify* message after committing each block;
- certificates are formed using Schnorr multi-signatures, so nodes are required to generate and publish public commitment values V_i together with their public keys; V_i values are used by the Leader to construct certificates.

We call Steady State phases in the same way as Solida does: propose, prepare, commit, notify.

In the remaining part of the chapter, we will introduce a certificates construction process and then describe phases of a Steady State. Successful execution of those phases will commit value into the Ledger.

- **Certificates construction**

Cypherium uses Schnorr multi-signature algorithm (Schnorr [1991]) to construct certificates. The process consists of several steps. The Leader forms a challenge by combining a public commitment of each committee node. Then each committee member signs data using the challenge and a node's private commitment. Finally, the Leader combines all signatures into a multi-signature which together with a public challenge form a certificate.

More specifically, before sending a *Propose* message a leader forms collective challenges G^P and G^C from a public commitment V_i of committee members *which was published by each node together with its signature.*

$$G^P = H(V||S_p)$$

$$G^C = H(V||S_c)$$

where $V = V_0 * V_1 * \ldots * V_{n-1}$, V_i is a public commitment of a i-th node; H is a cryptographic hash; $||$ stands for a concatenation; S_p is a proposed message with *Prepare* tag; S_c is a proposed message with *Commit* tag. So, the leader computes collective challenges and broadcasts it with a *Propose* message to committee members. Then committee members use challenges to sign corresponding messages: at first *Prepare* message and then, if it's accepted, *Commit* message.

Challenge G^P should be used to sign a message S_p in the following way: $r_i = v_i - G^P * x_i$, where v_i is a private commitment corresponding to V_i; x_i is a private key corresponding to X_i; $V_i = g^{v_i}$ and $X_i = g^{x_i}$, g is a given and well-known generator of a group G. A signature itself is a pair of (G^P, r_i).

Signatures from all signing participants can be combined into a certificate. An aggregated signature is a pair (G^P, r) consisted of a challenge G^P and a sum $r = \sum_i r_i$.

To verify resulting signature, one needs to compute $V' = G^r * (X_0 * X_1 * \ldots * X_m)^c$ and check that c equals to $H(V'||S)$, where r and c are from *signature*; S is a certified message; X_i are public keys.

Challenge G^C is used in the similar way to sign and verify *Commit* message.

- **propose** The leader L forms a transaction block from a set of valid transactions tx, broadcasts the transaction block, and broadcasts

$$\circledast(Propose, (H^P, \sigma'_L), viewId, G^p, G^c, \sigma_L)$$

message, where H^P is a header of a newly constructed block signed by the Leader as σ'_L; $viewId$ is a current configuration-lifespan-view tuple (c, e, v) the leader aware of; G^P and G^C are collective challenges for propose and commit phases; σ_L is a leader's signature of the proposed message.

After receiving $(Propose, (H^P, \sigma'_L), viewId, G^p, G^c, \sigma_L)$ message with a transaction block, a member $M \in M(c, e, v, s)$ checks if it's valid by making sure that all the following statements are true:

  - $L = L(c, e, v)$ and L has not sent a different proposal
  - tx is a set of valid transactions which can be applied to World State
  - H^P corresponds to a block's body tx and σ'_L is a valid signature of H^P

- **prepare** After receiving the *Propose* message and checking its validity, a committee member M signs a body of the message using G^P as described earlier. When the signing process is finished, a node M broadcasts a *Prepare* message:

$$\circledast(Prepare, (H^P, \sigma'_M), viewId, \sigma_M)$$

where H^P is a header of a proposed block signed with σ'_M; σ_M is a signature of *Prepare* message.

Once a member receives signed *Prepare* messages from **all** participants involved into a signing process, the member aggregates signatures into an acceptance certificate A. Aggregated signatures are a pair consisted of the sum of r_is and G^P. Once the node M constructed an acceptance certificate, we say that it's accepted a corresponding proposed block with a header H^P.

- **commit** Upon accepting H^P, a node M broadcasts

$$\circledast(Commit, (H^C, \sigma'_M), viewId, \sigma_M)$$

where H^C is a header which references the accepted block *(H^P and H^C are similar but contain different tags to differentiate between header types)*; σ'_M and σ_M are signatures.

Similarly to a prepare phase, when a member receives signed *Commit* messages from **all** participants involved into a signing process, the member aggregates signatures into a commit certificate C. A commit certificate is constructed similarly to a Prepare certificate by using *Commit* messages instead of *Prepare* message and by using G^C challenge *(from Propose message)* instead of using G^C. Once the node M constructed a commit certificate C, we say that it's committed a corresponding proposed block with a header H^C.

- **notify** Upon committing h, a committee member M broadcasts

$$\circledast(Notify, H^C, \sigma_M)$$

where H^C is a header of a committed block.

3.2.3. *View Change.* A Byzantine leader cannot violate safety but can prevent progress by simply not proposing. Thus, in PBFT, every honest member M monitors the current leader, and if no new slot is committed after some time, M abandons the current leader and supports the next leader. Since we have assumed a worst-case message delay of Δ, it is natural to set the timeout based on Δ. A particular timeout value is 4Δ or 8Δ.

Now we will consider events and messages which happen within the View Change subprotocol:

- **view-change** Whenever a member M moves to a new slot s in a steady state, it starts a timer T. If T has reached 4Δ and M still has not committed a slot s, M abandons the current leader and broadcasts

$$\circledast(ViewChange, viewId, \sigma_M)$$

where $ViewChange$ is a tag indicating a view-change message; σ_M is a signature of a tuple consisted of all previous elements of the message made using M's private key.

- **view-change-ready** Upon receiving $2f+1$ matching $ViewChange$ messages for $viewId$, if a member M is not in a view higher than $viewId$, it sends

$$\circledast_{L'}(ViewChangeReady, viewId, V)$$

message to a new leader $L' = L(c, e, v+1)$, where $ViewChangeReady$ is a tag indicating that the member received $2f+1$ messages, $viewId$ is the same tuple as in $ViewChange$ message; V is *view-change certificate*, basically it is a map from a public key of a $ViewChange$ message sender to its σ; $\circledast_X$ stands for a serialized message sent to a member X. This message is supposed to notify a new leader about a view-change event as soon as possible.

After that, if M does not receive a $NewView$ message from L' within 2Δ, M abandons L' and broadcasts

$$\circledast(ViewChange, next(viewId), \sigma_M)$$

where $next(viewId)$ is a function increasing a passed view of passed $viewId$ by one.

- **new-view** Upon collecting $2f+1$ matching $ViewChange$ messages for $viewId = (c, e, v)$, received either from $ViewChange$s messages or from a $ViewChangeReady$ message, the new leader $L' = L(c, e, v+1)$ concatenates them into a view-change certificate V and broadcasts

$$\circledast(NewView, next(viewId), V, \sigma_{L'})$$

. After that, L' enters a new view $(c, e, v+1)$.

Upon receiving a $(NewView, viewId, V, \sigma_{L'})$ message, if a member M is not in a view higher than $viewId$, it enters a view $viewId$ and starts a timer T. If T reaches 8Δ and still no new slot is committed, M abandons L' and broadcasts

$$\circledast(ViewChange, viewId, \sigma_M)$$

.

- **status** Upon entering a new view $viewId$, M sends

$$\circledast_{L'}((Status, viewId, H^C, H^A, \sigma'_M), C, A, \sigma_M)$$

where $L' = L(c, e, v)$ is a new leader, implying $viewId = (c, e, v)$; H^C is a header hash of a block committed in the previous slot, and C is a corresponding commit certificate; H^A is a header hash of a block accepted by M in the current slot, and A is the corresponding acceptance certificate (H^A and A is None if M has not accepted any value for the current slot).

Status message is used when a committee leader changes after a view-change event to let a new leader know about the latest committed payload and the latest accepted payload.

We call an inner part of the message (i.e., excluding C, A) its header. Upon receiving $2f+1$ status, L' concatenates the $2f+1$ status headers to form a status certificate S.

L' then picks a status message that reports the highest last-committed slot $s*$ (which can be obtained from H^c); if there is a tie, L' picks the one that reports the highest ranked accepted value in slot $s*+1$. Let two certificates in this message be $C*$ and $A*$ (A* might be None).

- **repropose** The new leader L' broadcasts

$$\circledast(Repropose, viewId, H^{C*}, C*, H^{A*}, A*, S, \sigma_{L'})$$

where H^{C*} and $C*$ denote respectively a header of the highest committed block and a corresponding certificate (from the previous step); H^{A*} and $A*$ denote respectively a header of the highest accepted block and a corresponding certificate, both may be equal to None;

The repropose message serves two purposes. Firstly, $C*$ allows everyone to commit a slot $s*$. Secondly, it reproposes H^{A*} for a slot $s*+1$ and carries a proof $(S, A*)$ that H^{A*} is safe for a slot $s*+1$. The repropose message is considered invalid if any of the following conditions is violated:

- $s*$ is not the highest committed slot
- C is not for a slot $s*$
- $A*$ is not for the highest ranked accepted value for $s*+1$
- h^{A*} is not the value certified by A

Upon receiving a valid repropose message, a member M commits a slot $s*$ if it has not already committed; them M executes the **prepare/commit/notify** steps as in the steady state for a slot $s*+1$, and marks all slots $> s*+1$ fresh for view $viewId$. At this point, the system transitions into a new steady state.

3.2.4. *Reconfiguration.* As described in the Chapter 3, to join a committee a node has to find a nonce η which satisfies

$$H(puzzle_c \oplus P \oplus \eta) \leq T \tag{7}$$

Upon finding a PoW, a miner tries to join the committee by driving a reconfiguration consensus as an external leader. To do so, it firstly broadcasts the PoW to committee members.

- **new-lifespan** Upon finding a PoW, a miner broadcasts it to the committee members:

$$\circledast(NewLifespan, minerPk, nonce, puzzle, viewId, \sigma_{miner})$$

Upon receiving a new valid PoW for the current configuration (that M has not seen before), M forwards the PoW to other members,

$$\circledast(NewLifespan, minerPk, nonce, puzzle, viewId, \sigma_M)$$

Then M enters view $(c, e+1, 0)$, sets the PoW finder as its 0-th leader $L' = L(c, e+1, 0)$ of the new lifespan, and starts a timer T. If T reaches 8Δ and still no new slot is committed, then M abandons L' and broadcasts

$$\circledast(ViewChange, (c, e+1, 0), \sigma_M)$$

- **status** Upon entering a new lifespan, M sends to L'

$$\circledast((Status, (c, e, 0), H^C, H^A, \sigma'_M), C, A, \sigma_M)$$

where H^C, H^A, C and A are defined the same way as in the View Change sub-protocol. Upon receiving $2f+1$ status, L' constructs $s*$, S, $C*$ and $A*$ as in View Change.

- **repropose and reconfigure** Let H^C be the committed value in the highest committed slot $s*$ among S. Let H^A be the highest ranked accepted value (can be None) into slot $s*+1$ among S. Note that H^C and H^A are certified by $S, C*$ and $A*$. Now depending on whether H^C

and H^A are transactions or reconfiguration events, the new external leader L' takes different actions:

- If H^C corresponds to a reconfiguration event to configuration $c+1$, then L' simply broadcasts C and terminates. Terminating means L' gives up its endeavor to join the committee (it is too late and some other leader has already finished reconfiguration) and starts working on $puzzle(c+1)$.
- If H^C corresponds to a block with a batch of transactions, and H^A is a reconfiguration event to configuration $c+1$, then L' broadcasts

  $$\circledast(Repropose, (c, e, 0), H^C, C*, H^A, A*, S, \sigma_{L'})$$

  and then terminates.
- If H^C corresponds to a block with a batch of transactions, and H^A is None (it means that $A*$ is None, too), then L' tries to drive the reconfiguration consensus into a slot $s*+1$ by broadcasting

  $$\circledast(Repropose, (c, e, 0), H^C, C*, H^{A'}, A*, S, \sigma_{L'})$$

  where $H^{A'}$ is a header reconfiguration event that lets L' join the committee. If this proposal becomes committed, then starting from a slot $s*+2$, the system reconfigures to the next configuration and L' joins the committee replacing the oldest member.
- If H^C and H^A are both batches of transactions, then L' first reproposes H^A for a slot $s*+1$ by broadcasting

  $$(Repropose, (c, e, 0), H^C, C*, H^A, A*, S, \sigma_{L'})$$

  After that, L' tries to drive a reconfiguration consensus into a slot $s*+2$ by broadcasting

  $$(Propose, (c, e, 0), (H^{A'}, \sigma_{L'}), G^p, G^c, \sigma_{L'})$$

  where $H^{A'}$ is a header of reconfiguration event that lets L' join the committee. If this proposal becomes committed, then starting from a slot $s*+3$, the system reconfigures to the next configuration and L' joins the committee replacing the oldest member.

Whenever a committee node doesn't know if the message by L' was reproposed or proposed, it can request this data by sending

$$\circledast_{L'}(RequestPayload, H^P)$$

where H^P is hash of unknown data, which can be obtained from reproposed or proposed message received from L'.

4. Cypherium Virtual Machine (CVM)

4.1. **Structure.** The Cypherium blockchain comes with a virtual machine environment for executing the smart contracts. In order to ensure the maximum flexibility and the quality of the smart contract execution, we decided to integrate two separate virtual machines into the CVM. Namely, we incorporate the Ethereum Virtual Machine (EVM) along with an in-house implementation of the Java Virtual Machine (JVM). Hence all the smart contracts that are written for the EVM can be executed with CVM seamlessly without any further complications or conversions. On the other hand, our JVM implementation is compatible with the Java 1.8 standard including the bytecode format and the class files.

Figure 1 depicts the development and execution stack of the CVM. The smart contract can be written in either the Java language or the Solidity language which will be consequently compiled by the respective compilers and associated development tools into the bytecode form. The bytecode is stored as a field in the corresponding **transaction** data structure. Upon execution, the CVM execution environment will be responsible for picking up the appropriate underlying virtual machine to execute on.

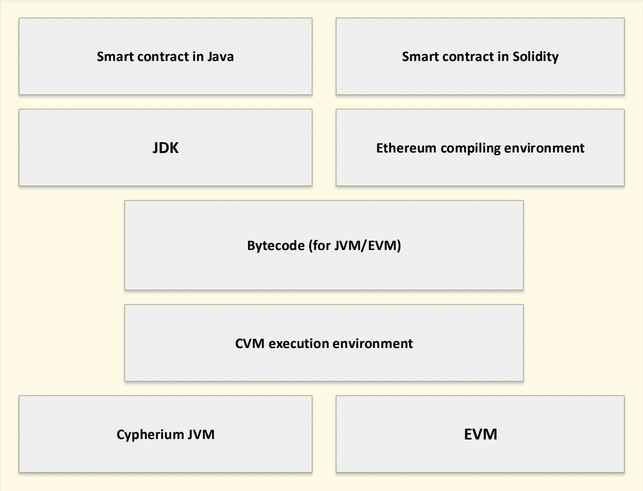

Smart contract in Java
Smart contract in Solidity
JDK
Ethereum compiling environment
Bytecode (for JVM/EVM)
CVM execution environment
Cypherium JVM
EVM

Figure 1. CVM development and execution stack

4.2. **Execution environment.** The bytecode is stored in the **data** field of a **transaction** from 2.6. The CVM execution environment determines the target virtual machine to execute the underlying bytecode on by looking up the magic number in the first 4 bytes (**CA FE BA BE** for a JVM bytecode and otherwise for an EVM bytecode). Algorithm 1 describes the switching mechanism.

Algorithm 1 CVM execution environment

```
code := transaction->code ▷ Read the bytecode from the transaction
magic := code[0:3]        ▷ Read the first 4 bytes of the bytecode
if magic == 0xCAFEBABE then
    JVM_engine(code)
else
    EVM_engine(code)
end if
```

5. Future Work

5.1. **Security.** An attacker cannot forge a transaction of another user without obtaining the user's private key first. In the worst-case scenario, without harvesting private keys, he can only attempt to recall his own transactions by modifying the blockchain history. However, this too could not succeed as all nodes must check if a key block contains enough Proof-of-Work before accepting it and all transaction blocks associated with it. If a node becomes aware that another validator is maliciously excluding it from the committee, that node can broadcast view change to other validators and have the malicious validator removed from the committee. Merely corrupting 1/3 validators would be insufficient, though, for the attacker to rewrite the blockchain history. A malicious actor would also have to outvote the remaining 2/3 of honest validators to get the transaction block passed, and to do so would require over 2/3 total computing power in the network. In the unlikely event that an attacker is able to outrun all honest nodes, he can only reverse his own past payments, which ultimately yields less profit than participating honestly in the network. The rest of nodes can still verify all transactions if they are willing to do so. The validators do not receive their reward immediately, but only after they have validated all transaction blocks designated to them. If during any round a node goes offline, it will not receive any reward, and thus, the network incentivizes nodes to continue validating throughout their term.

5.1.1. *IP Conceal.* It is ideally that we could hide our IPs from the external world, to avoid various security issues such as DDOS. We can use a mechanism called "IP group", inside which several "virtual IPs" (VIP) are assigned. Each VIP can be binded with a real IP of a node. This is essentially a many-to-one mapping, that is, one or more VIPs can be binded with a single real IP. If, for any reason, a node is down, its VIPs got reassigned to its nearby peers. Each IP group has a single external IP address. During consensus, a request will be routed to the the IP address of the VIP, and we design a load balancing algorithm to distribute the load to one of its internal VIPs. After each consensus operations, i.e. verification, pre-commit, commit, etc. it will send out the response message to the external world, which is the same as out current PBFT mechanism. Our current reconfiguration is to change a single committee node, we can easily extend it to change a single IP group during reconfiguration. Reward handling might become tricky, as we will now have to take care of all the nodes inside a particular IP group. How to distribute reward to a node inside a IP group even if it didn't do much work of the consensus (due to the natural imbalance of the consistent hashing algorithm) is yet to be discussed.

5.2. **Micro Service.** So far, a node will have to deploy all the functional code, and handle all the executions. When the application is small, this architecture is fine. However, our platform will inevitably grow in the future, which will become more difficult for us to extend. Micro service architecture is a good candidate to solve such issues. For example, if we decide we will add a new storage layer into our platform, such as IPFS, certain nodes can apply to become a storage nodes if they have big disks, but other limiting resources such as CPU and network bandwidth. These nodes can only deploy our IPFS module and don't have to deploy any other modules. They will use well defined interfaces to communicate with other peers with different functionalities.

5.3. **Simulation and Load Testing Environment.** We are in the process to testing our platform with a private chain beta testing. Due to the limitation of beta testing, we are not able to test many real customer traffic. As a rule of thumb, we should avoid testing with massive real customer traffic for the sake of load testing. We will develop a generic load testing mechanism, to support our current version of the platform, as well as any future extensions.

6. Conclusion

Proof-of Work remains the only consensus mechanism that keeps the promise of blockchain's decentralized revolution. Introducing Proof-of-Work as the node election mechanism enables anonymous participation in which benevolent actors can join or exit the network at will, while still ensuring that the network is secure and resistant to Sybil attacks. Elected nodes will only hold temporary positions to verify transactions, and thus the network will remain decentralization, despite some federated architecture. With Cypherium, confirmation time will no longer stifle the growing demands for scalable blockchains. Decoupling the mining process and transaction verification reduces confirmation time to the instantaneous standards of modern processing . In an ecosystem where promise and expectation have risen so high as to hurry and frustrate the practical implementations of distributed ledger technology, our system uniquely addresses the issues currently facing so many "next-generation" permission-less blockchain.

References

Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. *arXiv preprint arXiv:1612.02916*, 2016.

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

Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, page 13. ACM, 2016.

Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009.

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)*, pages 279–296, 2016.

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.

Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.

Ling Ren, Kartik Nayak, Ittai Abraham, and Srinivas Devadas. Practical synchronous byzantine consensus. *arXiv preprint arXiv:1704.02397*, 2017.

Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

Simon Sprankel. Technical Basis of Digital Currencies, 2013. URL `http://www.coderblog.de/wp-content/uploads/technical-basis-of-digital-currencies.pdf`.