Albatross
An optimistic consensus algorithm

Version 4 – December 10, 2019

Bruno França, *Nimiq Foundation*, bruno@franca.xyz
Marvin Wissfeld, *Nimiq Foundation*, marvin@nimiq.com
Pascal Berrang, *Nimiq Foundation*, pascal@nimiq.com
Philipp von Styp-Rekowsky, *Nimiq Foundation*, philipp@nimiq.com

***Abstract*—The area of distributed ledgers is a vast and quickly developing landscape. At the heart of most distributed ledgers is their consensus protocol. The consensus protocol describes the way participants in a distributed network interact with each other to obtain and agree on a shared state. While classical consensus Byzantine fault tolerant (BFT) algorithms are designed to work in closed, size-limited networks only, modern distributed ledgers – and blockchains in particular – often focus on open, permissionless networks.**

In this paper, we present a novel blockchain consensus algorithm, called Albatross, inspired by speculative BFT algorithms. Transactions in Albatross benefit from strong probabilistic finality. We describe the technical specification of Albatross in detail and analyse its security and performance. We conclude that the protocol is secure under regular PBFT security assumptions and has a performance close to the theoretical maximum for single-chain Proof-of-Stake consensus algorithms.

I. Introduction

The most famous classical consensus algorithm is PBFT, or practical Byzantine fault tolerance [1]. PBFT has greatly influenced the field since its creation in 1999 and is now a major component in many of the consensus algorithms being used or developed for blockchains. However, classical consensus theory has evolved significantly and, nowadays, the BFT algorithms providing the highest performance are speculative BFT algorithms.

Speculative BFT refers to a class of algorithms that have two modes for consensus: (1) the *optimistic* mode, where it is assumed that the nodes are well-behaved and so little security measures are applied, instead preferring speed, and (2) the *pessimistic* mode, where no such assumption is made and the only goal is to make progress even in the presence of malicious nodes.

The reason for speculative BFT algorithms being significantly faster than non-speculative versions lies in the optimistic mode, which allows them to compete with centralized systems in terms of speed. The optimistic mode, however, is not robust or safe at all, with any node being able to make an invalid update. When that happens, speculative BFT algorithms automatically enter into pessimistic mode, revert the invalid update and then change back into optimistic mode.

The idea here is deceptively simple. In PBFT, the nodes adopt a *"never trust"* attitude, all updates to the ledger being carried out with a focus on maximum security. In speculative BFT, the nodes adopt a *"trust but verify"* attitude, being allowed to make an update by themselves but with other nodes verifying the update afterwards and it being reverted if it is not valid.

Albatross, our novel blockchain [1] consensus algorithm, is modeled after some of these speculative BFT algorithms like Zyzzyva [2], but it also takes inspiration from other consensus algorithms like Byzcoin [3], Algorand [4] and Bitcoin-NG [5].

II. Overview

In this section, we will give a summary of Albatross, describing in an intuitive fashion the different types of validators, blocks and the chain selection algorithm. We also present the behavior of the protocol in the optimistic mode, discuss the ways validators may misbehave and how such misbehavior is countered in our protocol.

We will focus on the general structure of Albatross and leave the details for Section IV.

A. Validators

There are two types of validators: potential and active validators. A *potential validator* is any node that has staked tokens (i.e., tokens that are locked away for this purpose), so that it can be selected for block production. An *active validator* is a potential validator that was selected to produce blocks during the current *epoch* and thus has an active role.

The set of all potential and active validators is called the *validator set*. The list of active validators tasked to produce blocks during a given epoch is called the *validator list*. Similarly to other hybrid consensus algorithms, the validator list is chosen at random from the validator set, and the probability of a validator being chosen is proportional to its stake. Any given validator can appear more than once in the validator list, but only appears exactly one time in the validator set.

[1]It is important to note that we focus on a cryptocurrency use case here. We assume that blocks contain transactions and that the underlying token has a monetary value.

B. Blocks

There are two types of blocks that are produced by the validators:

- **Macro blocks:** These blocks are used to change the validator list and contain the identities of the new active validators. They do not contain user-generated transactions. Macro blocks are produced with PBFT.
- **Micro blocks:** These blocks are the ones that contain user-generated transactions. Each micro block is produced by a randomly chosen active validator and contains not only the transactions to be included, but also the current state and a random seed produced by the validator. Micro blocks only need to be signed by the corresponding elected validator.

One macro block is always followed by m micro blocks, with this pattern repeating throughout the blockchain. An *epoch* is composed of a macro block and the m micro blocks that preceded it.

C. Chain selection

Nodes will believe the longest chain, by number of blocks, to be the *main* chain. Note that, since macro blocks have finality and are thus forkless, any fork can only happen between two macro blocks. So, nodes only need to consider chains that include the last macro block.

D. Optimistic mode

Now, we will consider what happens during an epoch in Albatross, assuming that all validators are honest. We call this the *optimistic mode* because we are trusting that the validators will not misbehave.

1) At the beginning of an epoch a new validator list is chosen at random. The validators are chosen proportionally to their stake from the validator set.
2) Using a random seed, a validator is selected at random to produce the next micro block. This validator is called the *slot owner*.
3) The slot owner chooses which transactions to include in the block and calculates the resulting state. He also produces a new random seed that will be used to select the slot owner for the next block.
4) He includes the transactions, the current state, and the random seed in the block, signs it and relays it.
5) Now the next slot owner repeats the process and produces another micro block that appends some transactions to the blockchain and picks the next slot owner. This continues until the last micro block in the current epoch, which will pick the validator that will be PBFT leader for the macro block.
6) The PBFT leader will also produce a new random seed which will be used to select a new validator list for the next epoch. However, there are no transactions to include in the block.
7) The leader and the other validators sign the block and relay it.

Note that the validators will produce their random seeds using a *verifiable random function*. A verifiable random function is a pseudo-random function that cannot be manipulated by the validator and so guarantees a fair selection.

E. Misbehaving validators

The protocol above constitutes the optimistic case, assuming honest validators. However, Albatross also needs to be able to withstand malicious validators. To this end, we present appropriate measures for the three ways in which a validator can misbehave: producing an invalid block, creating a fork, and delaying a block.

1) Forks: Forks are not possible during a macro block, because PBFT is a forkless protocol. But a fork can be created if a validator produces more than one micro block in the same slot. To deal with this, we introduce *fork proofs*. Anyone can create a fork proof. Two block headers, at the same slot, signed by the same validator, constitute sufficient proof.

If a malicious validator creates or continues a fork, eventually an honest validator will produce a block containing a fork proof and the malicious validator will be punished.

2) Delays: Validators can potentially take a long time to produce a block, either because they went offline or because they maliciously try to delay the block production. In both cases, we need to change the slot owner. To this end, we use the *view change* protocol of PBFT.

After receiving a block, each validator starts a countdown. If he doesn't receive the next block before the timer ends, he sends out a message requesting a view change (a change of the slot owner). Any validator that receives such messages from at least two-thirds of the validator list, will no longer accept a block from the current slot owner. Instead, a new slot owner is chosen to produce a block and a new countdown begins.

The new slot owner is chosen in the following way. The random seed and a counter are hashed together to produce a random value that is used to pick a validator from the validator list. By increasing the counter, we can create an ordered list of slot owners. If the first slot owner does not produce a block in time, the second slot owner is selected to produce the block. If the second slot owner also fails to produce a block, the third slot owner is selected and so on.

3) Invalid blocks: When a validator produces an invalid block, either when he is the slot owner for a micro block or the PBFT leader for a macro block, the other validators just need to ignore that block.

4) Punishment: Validators that fork or delay blocks are punished in the same way: they get banned from producing any more micro blocks during the current epoch and their reward for the whole epoch is burned. Moreover, such validators are marked to be automatically removed from the registry of potential validators if they do not prove being active within two epochs.

III. Preliminaries

A. Practical Byzantine Fault Tolerance

The practical Byzantine fault tolerance consensus algorithm, or PBFT for short, was introduced by Castro and Liskov in

1999 [1].

Assuming that there are $3f+1$ nodes, PBFT can tolerate up to f faults and, thus, is optimal for a partially synchronous BFT protocol. There are several slightly different variants of PBFT but we will only describe the one used in Albatross.

PBFT proceeds in four rounds:

- **Pre-prepare:** In this phase, the leader multicasts a block proposal to the rest of the validator list.
- **Prepare:** After receiving the block proposal, each validator determines if it is valid. If it is, he calculates the block hash h and signs a message saying $\langle \text{PREPARE}, h\rangle$. Then, he multicasts the signature to the rest of the validator list. The leader also sends a prepare message.
- **Commit:** If a validator receives prepare messages from at least $2f+1$ validators, he signs a message saying $\langle \text{COMMIT}, h\rangle$ and multicasts it to the rest of the validator list.
- **Reply:** If a validator receives $2f+1$ commit messages from validators who also sent prepare messages, the block is considered finalized. Then, he gossips the block to the network.

Instead of sending all signatures in the block separately, we can save space by aggregating them. This way all the prepare messages are aggregated into a single signature and a bitmap is created stating which validators sent prepare messages. The exact same is done to the commit messages. So the justification for the block consists just of the two aggregated signatures and the corresponding bitmaps.

B. Boneh-Lynn-Shacham signatures

Boneh-Lynn-Shacham (BLS) is a signature scheme that was first introduced in 2004 [6] and uses both elliptic curve cryptography and bilinear pairings. Its security is less conservative than some other more commonly used digital signature schemes (for example ECDSA and Schnorr), but is still considered secure by the majority of the cryptography community. It also offers several advantages over ECDSA and Schnorr, namely: shorter signature sizes, deterministic signatures, and simple schemes for signature aggregation, threshold signatures, and multisignatures.

To understand BLS, first it is necessary to understand bilinear pairings. A bilinear pairing is a function that takes two elliptic curves points, possibly in two different curves and outputs a point in another curve. It also must have the bilinearity property. A more precise definition is:

> Let G_1 and G_2 be additive groups of prime order p, and G_T be a multiplicative group also of prime order p. Let $P \in G_1$ and $Q \in G_2$ be generators of G_1 and G_2, respectively, and $a, b \in \mathbb{Z}_p$. Then, a bilinear is a map $e : G_1 \times G_2 \to G_T$ such that $e(aP, bQ) = e(P,Q)^{ab}$ and $e(P,Q) \neq 1$.

For such bilinear pairings, $e(aP, bQ) = e(P,Q)^{ab}$ implies that $e(xP, Q) = e(P, xQ)$.

BLS signatures only require a bilinear pairing (with the curves to support it) and a hash function that maps to elliptic curve points in G_1. This hash function is defined as $h : \mathcal{M} \to G_1$. Given these primitives, the BLS signature scheme is defined as follows:

- **Key generation:** Choose at random an integer $x \overset{R}{\leftarrow} \mathbb{Z}_p$ and calculate $Y = xQ$. The secret key is x and the public key is Y.
- **Signing:** Let $m \in \mathcal{M}$ be the message. To create a signature, the message needs to be hashed $H = h(m)$. Then, calculate the signature as $\sigma = xH$.
- **Verifying:** Given a signature σ and a public key Y, accept the signature if $e(\sigma, Q) = e(H, Y)$.

As we have said before, BLS has a number of desirable features. One that is useful for our design is signature aggregation. BLS allows to combine n signatures of n different signers into a single signature thus saving a considerable amount of space. The scheme is as follows:

- **Setup:** Each of the n signers creates a secret key x_i and a public key Y_i.
- **Signing:** Each of the n signers uses their secret key to create signatures σ_i.
- **Aggregation:** The aggregated signature is simply the sum of all individual signatures $\sigma = \sum_{i=1}^{n} \sigma_i$.
- **Verifying:** Calculate the aggregate public key $Y = \sum_{i=1}^{n} Y_i$. Accept the signature σ if $e(\sigma, Q) = e(H, Y)$.

C. Verifiable Random Functions

A verifiable random function is a pseudo-random function that can provide publicly verifiable proofs that its output is correct.

More formally, after a user creates a public key Y and a secret key x, given an input s, the user can calculate the VRF $\pi, r = \mathsf{VRF}_x(s)$. Here, r is the pseudo-random output and π is a proof for the correct computation of it. Then, anyone who knows the public key and the proof can verify that r was correctly computed, without learning the secret key x.

Verifiable random functions have three important properties:

- **Pseudo-randomness:** The only way of predicting the output, better than guessing randomly, is to calculate the function.
- **Uniqueness:** For each input s and secret key x there is only one possible output r.
- **Public verifiability:** Anyone can verify that the output was correctly computed.

The BLS signature scheme has all these properties with the advantage that the signature serve as both the pseudo-random value and proof of correctness. For this paper we will use we will use BLS signatures as a simple verifiable random function.

IV. Specification

We will now give the technical specification of Albatross, describing in detail the networking, all of the validator's functions and processes, the block's format, the misbehavior handling protocols, the rewards and punishments, and the chain selection algorithm.

A. Networking

We choose S/Kademlia [7] for peer routing. S/Kademlia is a hardened version of the Kademlia distributed hash table, which greatly improves resistance of the network to eclipse and Sybil attacks. We recommend that all communication between peers is encrypted and gossiped.

By gossip we refer to when a node sends a message to its connected neighbors, those neighbors relay the message to their neighbors and so on until the message has been propagated through the entire network.

Always gossiping messages increases the DoS resistance of the validators. This is because each node only accepts incoming connections from a small set of other nodes and those connections tend to be long-lived. An attacker most likely won't be able to connect directly to a validator and must instead connect to other full nodes.

Honest full nodes will not relay invalid messages and so end up acting as guards to the validators. As the number of honest full nodes increases, so does the resistance of the network to DoS attacks.

We also define transactions as any input to the state transition function and differentiate between two types of transactions: internal and external. External transactions are transactions that are propagated through the network and signed. They are typically used by the users to interact with the blockchain, for example for transferring tokens. Internal transactions are neither propagated nor signed, and are instead included in a block by its producer. An example would be a timestamp of a block.

B. Validator keys

Each potential validator needs to keep a set of three keys: a cold key, a warm key and a hot key.

The cold key is simply the key of the account of the validator. It is used to both stake and unstake funds.

The warm key is a key chosen by the validator when he initially stakes funds. This key will be used both to prove availability and sign some signaling transactions (see Section IV-C).

Finally, the hot key is a BLS key (see Section III-B), which is created for the sole purpose of signing blocks and producing new random seeds (see Section III-C). This key is also chosen by the validator.

The reason to have three separate keys is operational security. The hot key and the warm key need to be kept online in order to produce blocks and do other consensus-related tasks. But having a cold key, which is used only for accessing the account, allows validators to still keep their funds in a cold wallet.

C. Validator signaling

There are three types of transactions that nodes can send to the network to change their validator status. These transactions signal their desire to become a validator, to stop being a validator and to change their validator information.

1) Staking: When nodes want to become validators, they must send a *staking* transaction. A staking transaction has six pieces of information:

- **Main address:** The funds to be staked are taken from this address. It also doubles as the public cold key.
- **Amount:** The amount of tokens to be locked up while the node is a validator.
- **Warm key:** The public warm key.
- **Hot key:** The public hot key.
- **Proof of knowledge of secret key:** This is a signature of the hot public key using the corresponding secret key. It is used to prevent rogue key attacks.
- **Reward address:** An optional address the rewards from staking are paid to. If no reward address is given, the rewards will be added to the stake.

The result of the staking transaction is to lock the desired amount of tokens from the main address and adding the amount, the main address, the warm key, the hot key and the reward address to an *on-chain* registry of potential validators.

After the node is added to the registry, it becomes a *potential* validator. So, if the node gets selected to be an *active* validator in the next macro block, it can start producing blocks.

A staking transaction is, of course, signed with the validator's *cold* key.

2) Restaking: A *restaking* transaction allows validators to change some of the information stored in the validator registry. The transaction is signed with the *warm* key and consists of these four fields:

- **Amount:** The new amount of tokens to be staked.
- **Hot key:** The new public hot key.
- **Proof of knowledge of secret key:** The proof of knowledge for the new hot key.
- **Reward address:** The new optional reward address.

Evidently, this transaction can be used to update the stake amount, the hot key and the reward address of a validator. For efficiency reasons, some of the fields can be left blank, if the validator does not wish to update them.

3) Unstaking: When nodes wish to stop being validators they need to send an *unstaking* transaction. An unstaking transaction is a message, signed with the validator's *cold* key, expressing that the node no longer wishes to be a validator. After the transaction gets accepted, the deposit is returned to the validator's address and all his information (address, warm key and hot key) is deleted from the registry.

After sending an unstaking transaction, validators are required to remain as validators until the end of the epoch and their deposit can only be withdrawn after the end of the next epoch. Delaying the return of the deposit is required to allow the punishment of past validators who misbehaved right before the end of the epoch.

D. Validator selection

There are two cases in which it is needed to select a random subset of validators: at the end of an epoch when a new validator list is chosen and before every micro block to choose the next slot owner from the validator list.

1) Random seed generation: For producing the random seeds we rely on the BLS signature scheme as an instantiation of a verifiable random function.

In the *genesis* block there will be an initial random seed. This initial seed will need to be sourced from the outside world. We can use, for example, lottery numbers [8] or the hashes of Bitcoin headers [9]. Another possibility, to further reduce any possibility of bias in the initial seed, is to employ distributed randomness generation algorithms [10]. We just take that entropy, hash it and convert it into an elliptic curve point.

In subsequent blocks, the random seed is produced as the BLS signature of the previous seed by the block producer (or PBFT leader in the case of a macro block). This creates an infinite chain of BLS signatures and random seeds.

2) Validator list: In Albatross the entire validator list is changed every epoch. The random seed present in every block, which we will use to select the new set, is a BLS signature and, hence, an elliptic curve point. We denote that point by S.

To select the new validator list, we start by getting the addresses and the corresponding deposit amount of every potential validator. Then, we order the addresses in a deterministic way (for example, by lexicographic order). Lastly, we map the ordered addresses to their deposit amount, such that the deposit amount represents a range. For example, if there are 10 tokens staked by A_1, 50 tokens by A_2 and 15 tokens by A_3, then the mapping would be the one shown in table IV-D2.

TABLE I
VALIDATOR STAKES ORDERED AND MAPPED TO RANGES

Address	Deposit	Range
A_1	10	$[0, 9]$
A_2	50	$[10, 59]$
A_3	15	$[60, 74]$

With this mapping, we can now run **Algorithm 1** to select the new validator list [2].

Algorithm 1 Validator list selection algorithm

```
validator list = {}
$S$ = random seed
$t$ = total amount staked
$i = 0$
while validator list is not full do
    $r = hash(S \,||\, i) \mod t$
    $v$ = potential validator whose range contains $r$
    add $v$ to validator list
    $i = i + 1$
end while
return validator list
```

[2]This way of selecting a validator list is only illustrative, and so is purposefully simple and naive. There are many algorithms that can be used to sample from a discrete distribution, several of which are more efficient than the one showed here.

3) Slot owner list: Given a random seed S and a validator list, we can create a random ordering of the validator list to produce the next block. This random ordering is called the *slot owner list*. While, in general, only the first validator of that list will actually produce the block, the rest of the list can be relied upon in case the first validator does not respond in time.

The algorithm for calculating this ordered list is as follows. First, we take the addresses of all the n active validators and order them deterministically. Then, we number them from 0 to $n-1$. Now, we can run **Algorithm 2** to produce the list of slot owners.

Algorithm 2 Slot owner selection algorithm

```
validator list = $\{v_1, v_2, \ldots, v_n\}$
slot owner list = {}
$S$ = random seed
$i = 0$
while validator list is not empty do
    $n$ = validator list size
    $r = hash(S \,||\, i) \mod n$
    $v$ = active validator numbered $r$
    add $v$ to slot owner list
    remove $v$ from validator list
    $i = i + 1$
end while
return slot owner list
```

E. Block format

We model the format of both macro and micro blocks like this:

- **Header:** The block header.
- **Digest:** A field containing auxiliary data that may be necessary for synchronization (see Section VI). It will mostly be composed of internal transactions.
- **Transactions:** The data to be input into the state transition function. It will consist mostly of external transactions.
- **Justification:** The information necessary to make the block valid according to the consensus rules.

Although the state would also technically be part of the block, we do not include it since it usually is not propagated in the network as part of a block. The block header consists of the following components:

- **Parent hash:** The hash of the previous block header.
- **Block number:** The number of the current block.
- **View number:** The view number (see Section IV-F) of the current block.
- **Digest root:** The root of the Merkle tree containing the digest.
- **Transactions root:** The root of the Merkle tree of the transactions.

- **State root:** The root of the Merkle tree [3] of the state.

However, we can be more specific regarding the information that needs to be included in macro and micro blocks.

The header is equal for both micro and macro blocks. For the body, we will explain both the digest, the transactions and the justification in more detail.

1) Micro blocks:

- **Digest:** The digest consists of the timestamp, the random seed and, when necessary, any aggregated *view change* messages and *fork proofs*.
- **Transactions:** The transactions field contains all of the external transactions.
- **Justification:** The justification consists only of an identifier [4] and the signature of the validator that produced the block.

2) Macro blocks:

- **Digest:** The digest of a macro block contains all the same information as the digest of a micro block plus the (public) hot keys of the new validator list and the hash of the previous macro block header.
- **Transactions:** A macro block cannot have any external transactions so this field will be empty.
- **Justification:** Since this is a PBFT block, the justification consists of the two rounds of validator signatures (see Section III-A).

F. View change protocol

If a validator, for some reason, does not produce a block during his slot there needs to be a process to allow another validator to produce the block. This process is the *view change* protocol and is closely modeled after the protocol of the same name in PBFT.

Given $3f + 1$ active validators (of which at most f are malicious), a list of slot owners $\{s_1, s_2, \dots, s_n\}$ and a timeout parameter [5] Δ, each active validator runs **Algorithm 3** immediately after receiving a block. A *view number* keeps track of the current index within the slot owner list.

It is important to clarify and understand the following points with regards to this algorithm.

First, a view change message is a signed message containing the statement $\langle \text{VIEW-CHANGE}, i, b \rangle$, where i is the current view number as defined in the algorithm above and b is the current block number.

Second, for the next block to be accepted, it must include $2f + 1$ view change messages accepting its producer at $i + 1$.

Third, after the timeout, a node will wait indefinitely for either the block or $2f+1$ view change messages to be received.

[3]It is worth noting that the state does not necessarily need to be represented in a Merkle tree, but that the usage of other advancements, such as batchable RSA based accumulators [11], are possible as well.

[4]This identifier can either be the (public) hot key of the validator or the position of the validator in the validator list.

[5]For this paper we will consider that this parameter is static and hardcoded into the software, but it is possible to have the parameter be updated dynamically by using a combination of the timestamps and the number of view changes that happened in the recent past. The dynamic update protocol would be similar to how PoW blockchains adjust their mining difficulty.

Algorithm 3 View change algorithm

```
$i = 0$ (view change number)
loop
    wait for $(i + 1) \cdot \Delta$ time
    if a valid block was received from $s_i$ then
        terminate algorithm
    else
        broadcast a view change message
    end if
    if $2f + 1$ view change messages are received then
        commit to not accepting a block from $s_i$ in this slot
        $i = i + 1$
    end if
end loop
```

Fourth, note that after a node receives $2f + 1$ view change messages, it will no longer accept, or build on, a block from the delayed slot owner. Even if the node has received the block before completing the $2f + 1$ view change messages.

Fifth, a block with a higher view number always has priority over a block with a lower view number. So, if a fork is created because of a view change, the chain that starts with the block containing the highest view number is always preferred.

G. Fork proofs

When a validator creates two or more micro blocks in the same slot, he is punished by having his reward for the epoch slashed. We do not care if a PBFT leader proposes two macro blocks because this situation will not result in a fork.

It is worth noting that if a validator produces two micro blocks, one valid and one invalid, he will still be slashed, even though he did not create a fork. We opt for this in order to reduce the size of the fork proof. It is easier to prove that two blocks exist in the same slot than that two *valid* blocks exist in the same slot.

The fork proof consists of two block headers and their respective justifications. In order for the fork proof to be valid the following conditions must be met:

- The block headers must have the same block number and view number.
- The justifications must be valid.

This essentially proves that a validator created, or continued, a fork. A fork proof only punishes a single entry in the list of active validators. If a validator has multiple entries in the active validator list, only the one that owned the misbehaving slot will be punished. Also, if there are multiple misbehaving validators, several fork proofs have to be included into the blockchain.

H. Rewards

The rewards, consisting of the coinbase plus the transaction fees, are always divided equally among all active validators.

Additionally, the rewards for an entire epoch are only distributed at the end of the next epoch (on the macro block). So, the reward distribution is always delayed by one epoch.

The reason for this delay is to allow ample time for validators to submit fork proofs before the rewards are distributed.

Note that validators do not need a large incentive to produce blocks since block production in Albatross is extremely cheap. No expensive mining equipment or GPUs are needed, and any regular computer with a good internet connection suffices.

I. Punishments

Whether a validator delays a block or creates a fork, he is punished in the same way:

- The misbehaving validator is no longer considered in the slot owner selection, i.e., he is barred from producing any more micro blocks, and from being the leader of the macro block, during the current epoch. However, he can still participate in the view change and macro block voting.
- His reward is confiscated and burned [6].
- He is marked to be expelled from the validator registry. Just like in an unstaking transaction, the deposit is returned to the validator and his information is deleted from the registry.

The prohibition of producing micro blocks is applied immediately after a view change (for a delay) or in the block where a fork proof is included (for a fork).

However, the reward confiscation and the validator registry expulsion only happen on the macro block at the end of the next epoch. In other words, it happens at the same time the rewards are distributed.

Note that fork proofs can be submitted until the end of the epoch after the one where the fork occurred. In addition, a validator that delayed a block can avoid being expelled from the validator registry if he can prove, before the end of the next epoch, that he is available and still knows his secret hot key [7].

Producing invalid blocks is not punished in any way since it does not impact the consensus.

J. Chain selection

The chain selection algorithm is more complex than our brief description of it in the overview section since it needs to take into account malicious forks and view changes. We use the following cumulative conditions, from highest to lowest priority, to choose a chain:

1) The chain with the most macro blocks.
2) The chain that has the blocks with the highest view number.
3) The chain with the most blocks.

Still, it is possible for two chains to tie on all three conditions (for example, when a malicious validator creates a fork). In that case both chains are considered equal and there is no clear chain to select. Thus, the next slot owner can build on top of either one.

[6] The reward is not divided among the other validators so as to not incentivize them to attack each other (ex: by doing a denial-of-service).

[7] This can be done either by signing the macro block or perhaps by submitting a special transaction (signed with the warm key) containing the validator's public hot key and a proof of knowledge of secret key. The details are left open for the implementation.

V. Staking pools

In proof-of-work protocols, miners tend to form groups in order to combine their hashpower together and share the rewards. These groups are, of course, mining pools and they are a major part of PoW blockchains.

A similar concept exists for proof-of-stake protocols: the staking pools. Like with mining pools, in staking pools stakers combine their stakes together and divide the rewards.

A. Comparison with mining pools

The reason for the existence of mining pools is that they reduce the variance of rewards for the miners. A miner that only has a small fraction of the total hashpower of a given blockchain can only expect to receive a reward very rarely but, if he joins a mining pool he can receive smaller rewards with a higher frequency. The miner ends up with a more predictable and steady cashflow.

Evidently, mining pools charge a fee for their service but it is normally very small. For most users the reduction in reward variance compensates the small reduction in the expected reward. This is because of two reasons:

1) A majority of people are risk-averse.
2) Miners have fixed expenses (electricity, equipment, etc). If they don't receive a reward for a long enough time, they can become bankrupt.

Staking pools, similarly to mining pools, allow stakers to trade profit for decreased variance, resulting in more steady earnings. So, the first reason for the existence of mining pools, *risk aversion*, also applies to staking pools.

The second reason though does not apply because, unlike miners, stakers don't have significant fixed expenses. A staker, in addition to coins, only needs a decent computer and a good internet connection, both of which are relatively cheap. Consequently, stakers face little risk of ruin.

Stakers have a different reason to use staking pools, and it is *convenience*. While acquiring hashpower can be quite difficult, requiring a high degree of technical knowledge and high capital costs, acquiring coins is easily achieved by most cryptocurrency users. The issue then becomes that many coin-holders will not have the technical knowledge or disposition to run a full-node, a task that is necessary to produce blocks. Staking pools solve this problem, by offering to handle all the required technical complexity in exchange for a small fee.

B. Decentralization

The different set of incentives that rule mining pools and staking pools also influence the decentralization of PoW and PoS blockchains.

The barrier of entry to become a miner is high, which results in a small number of miners to begin with. And since miners only care about reward variance, they tend to join the largest mining pools. All of this results in PoW blockchains being dominated by just a few very large mining pools.

However, to become a staker it is only needed to buy coins so the number of stakers can potentially be as large as the number of coinholders. Variance is also not as important to stakers, for whom convenience is often the primary concern. This normally means that there is a diverse array of institutions offering staking services, like exchanges, wallets, custodians, etc. Thus, PoS blockchains tend to be significantly more decentralized when compared to PoW blockchains.

In general, we believe that staking pools contribute positively to the security of PoS blockchains because they increase the total number of coins being staked while keeping the network reasonably decentralized.

C. *Stake delegation*

Stake delegation refers to the practice of a coinholder *delegating* his staking power [8] to a third-party without giving the same third-party access to the coins. Normally, stake delegation is a layer 1 feature of PoS blockchains and is absolutely necessary for the existence of staking pools. Without stake delegation, only custodial staking pools can exist.

In Albatross, thanks to the different validating keys, it is possible to do stake delegation. Imagine that Charlie is a coinholder and Paula is a staking pool operator. All that Paula needs to do in order to start a pool is to create an address and a BLS keypair and publish them. Then Charlie can send a staking transaction (see Section IV-C) with Paula's public BLS key as the hot key and Paula's address as the reward address.

Now Charlie knows the cold and warm keys, which gives him control over his funds, while Paula knows the hot key, giving her control over the block production. The rewards are deposited into Paula's address, who then will share them with the pool users. To join another stake pool Charlie only needs to send a restaking transaction with the new pool's hot key and reward address.

Also, since Paula can watch the blockchain and see when any staking (also restaking or unstaking) transactions appear that use her hot key, there is no need for Charlie to coordinate with Paula. All needed information is already communicated on-chain.

In case Charlie tries leveraging Paula's stake pool by adding her hot key while retaining the rewards for himself by using his own reward address, Paula can detect this misbehavior. Moreover, she does not need to produce any blocks in case Charlie's entry is selected as a slot owner as this will only slash the rewards for Charlie.

VI. Synchronization

New nodes who want to join the network need a way of synchronizing with the blockchain, since the only information they start with is the *genesis* block, which is hardcoded into the client software. However, the way in which they synchronize depends on what type of node they are: *archival*, *full* or *light*.

[8]In some PoS blockchains, stake delegation can also refer to delegating voting power.

A. *Archival nodes*

Archival nodes need to download and verify all the blocks, including the micro blocks, without exception. They are the safest option to synchronize but also the slowest, requiring the node to download vast amounts of data. Also has the disadvantage that the amount of data to download grows linearly with time, thus the synchronization time for archival nodes increases over the life of the blockchain. We expect archival nodes to be run only by businesses and other public services (for example, block explorers).

B. *Full nodes*

Full nodes only need to be able to produce blocks and as such they mainly need to learn the current state of the blockchain. To synchronize, full nodes begin by downloading all the macro block headers (including the digests) since genesis. Since each macro block header contains the hash of the previous macro block header, and the validator list only changes at the macro blocks, the macro block headers form a chain that allows anyone to be certain of the latest macro block header.

After verifying the latest macro block header, the full node then requests the state, which he can check against the state root. To finalize he downloads and verifies all of the micro blocks that were produced since the last macro block. The full node synchronization is now complete and he can start verifying, or even producing, new blocks.

This synchronization option is much faster than the archival node synchronization, in fact only the macro block headers, the state and a constant number of micro blocks (in average half of one epoch) are downloaded. The synchronization time for a full node still grows linearly over time, but much more slowly, only adding one macro block header each epoch. Furthermore, in order to save disk space, full nodes can safely prune the micro blocks of old epochs.

Alternatively, if syncing the state is deemed to be too slow, it is also possible to augment the macro blocks with a hash of a Merkle tree over all transactions in the corresponding epoch. Then, instead of downloading the state, full nodes can just download all the transactions and calculate the resulting state.

Note that full nodes in Albatross do not verify blocks before the current epoch, instead only verifying blocks after the last macro block. This makes them strictly unsafer than archival nodes, but only marginally [9]. As long as, at all points of the life of the blockchain, there was at least one honest full node, this synchronization method is secure.

[9]In order for an invalid block to permanently become part of the blockchain and be accepted by every future full node, then *all* of the existent full nodes when the invalid block was produced must collude. If a single honest validator exists at that time, he can store the block and provide it as proof that the current blockchain is not valid. The process to inform the community that the blockchain has been compromised has to be necessarily *off-chain* and *informal*, but is nevertheless possible.

C. Light nodes

Light nodes, also called light clients, have the fastest synchronization but also offer the fewest security guarantees. In addition to that, they are unable to verify or produce blocks. Light clients start their synchronization in the same way as full nodes, by downloading all of the macro block headers and their digests. But then they only download the headers and the digests of all of the micro blocks that were produced in the current epoch.

After synchronization, light nodes are only aware of the current validator list and the state root. However, this is all that is needed to securely query the blockchain state. The light client can simply ask other full nodes or archival nodes for specific parts of the state together with corresponding Merkle proofs and just check it against the state root.

The synchronization time for light nodes also grows linearly, in fact at the same rate as for full nodes, but since a light client only downloads headers and digests and does no verification, it is much faster to synchronize a light node.

Light nodes are intended to be run only by end-users of the blockchain, especially in highly constrained environments like browsers and smartphones.

There is also a different way of implementing light nodes, by relying on zero-knowledge proofs, that can significantly speed up the process of synchronizing the chain. In particular, there are two options to use zero-knowledge proofs in the light nodes implementation.

First, other nodes can provide non-recursive zero-knowledge proofs, proving that the chain of macro blocks was correctly constructed. This includes that each macro block is signed by at least two thirds of the keys present in the previous macro block and that the hash pointing to the previous macro block is correct. Depending on the zero-knowledge proof system employed, this can lead to a constant or logarithmic sized proof, and constant or logarithmic verification complexity.

The downside, however, is that the prover has to construct a new zero-knowledge proof for every macro block. To reduce the computation needed, nodes could create such proofs only for some macro blocks and a full or light sync is performed from there on.

The second option is to leverage recent advances in the construction of recursive zero-knowledge proofs [12] [13]. These proofs have the advantage that they can be extended recursively. Thus, the prover can construct a new zero-knowledge proof for every macro block only with the knowledge of the previous proof, the previous macro block and the current macro block.

VII. Security analysis

We will now analyze the security of Albatross. First, we will introduce the adversary model and then we will discuss static and adaptive adversaries, network partitions, probabilistic finality, and transaction censorship.

A. Adversarial Model

We need to start by defining the adversarial model, in other words, we need to state what type of attacker we expect to encounter. First, we will give definitions for the different types of economic actors:

- **Altruistic actor:** Follows the protocol even if it is prejudicial to him.
- **Honest actor:** Follows the protocol as long as it is not prejudicial to him.
- **Rational adversary:** Deviates from the protocol if it is profitable to him.
- **Malicious adversary:** Deviates from the protocol even if it is prejudicial to him.

In Albatross, the validator list is chosen randomly from the larger set of potential validators. There is a connection between the percentage of the total stake controlled by an individual and the number of validators that he gets to control. In fact, for a validator list of size n, if someone controls a fraction p of the entire stake then the probability of him gaining control of at least x validators is given by the cumulative binomial distribution:

$$P(X \geq x) = \sum_{k=x}^{n} \binom{n}{k} p^k (1-p)^{n-k} \tag{1}$$

Lamport et al. have shown that any reliable, deterministic Byzantine fault tolerant algorithm is only resistant up to $\lfloor \frac{n-1}{3} \rfloor$ of malicious nodes [14]. Thus, we are interested in the maximum fraction of stake p an adversary may control before exceeding this bound:

$$P(X \geq \lfloor \frac{n-1}{3} \rfloor) \leq \epsilon \Leftrightarrow \tag{2}$$

$$\Leftrightarrow \sum_{k=x}^{n} \binom{n}{k} p^k (1-p)^{n-k} \leq \epsilon \tag{3}$$

with $\epsilon > 0$ being practically negligible.

For our protocol, we show that an adversary controlling at most $p = \frac{1}{4}$ of the total stake, yields a practically negligible probability ϵ for a suitable size n of the validator list. Table II presents the result of our calculations based on varying numbers of n.

TABLE II
PROBABILITY OF AN ADVERSARY WITH $\frac{1}{4}$ OF THE TOTAL STAKE CONTROLLING $\frac{1}{3}$ OF THE VALIDATOR LIST

Number of validators n	200	300	400	500
Probability (%)	0.678	0.075	0.013	0.002

If Albatross has a validator list greater than 500 validators, we can consider the statement *"controlling less than $\frac{1}{4}$ of the total stake"* to be with overwhelming probability equivalent to the statement *"controlling less than $\frac{1}{3}$ of the validator list"*.

Thus, the main security assumption that we use for Albatross is the following:

Less than $\frac{1}{4}$ of the total stake is controlled by malicious adversaries.

This corresponds to:

Less than $\frac{1}{3}$ of the validator list is controlled by malicious adversaries.

Since it is more convenient, it is the latter one that we will actually use during the remainder of this analysis. The other assumption we need to make regarding the adversarial model is the following:

There is no set of rational adversaries, controlling $\frac{1}{3}$ or more of the validator list, that is capable of colluding.

Although similar to the first assumption, it actually models a different case. The first assumption deals only with malicious adversaries, who do not need any strong coordination in order to stop the network. The second assumption deals with rational adversaries for whom stopping the network is not a viable attack since they cannot derive any economic reward from it.

Rational adversaries only care about double-spending attacks, which they can use to increase their profit. However, double-spending is an attack that requires all of the attackers to collude. So there is no security issue even if all of the validator list is controlled by rational adversaries, only if those adversaries are capable of colluding.

To conclude, in the next sections we will analyze the security of Albatross and argue that it is secure under the following two assumptions:

1) *Less than $\frac{1}{3}$ of the validator list is controlled by malicious adversaries.*
2) *There is no set of rational adversaries, controlling $\frac{1}{3}$ or more of the validator list, that is capable of colluding.*

B. Static adversary

To begin, we will discuss attacks by a static adversary, which in this context means an adversary that, at the beginning of the protocol, can corrupt specific nodes but later cannot change which nodes are corrupted.

In this case, Albatross has a security model very similar to PBFT. The main difference between PBFT and Albatross is that in PBFT all blocks have *provable finality*, so all transactions are irreversible as soon as they get published in a block. In contrast to PBFT, transactions in Albatross have only *probabilistic finality* within an epoch, although the probability of reversibility is exponentially decreasing.

Depending on the number x of validators controlled by the adversary, on a validator list of size $3f + 1$, there are three possible cases:

- $x \leq f$: The attacker can't harm the network in any way.
- $f < x \leq 2f$: The attacker can delay the network indefinitely by stalling the view change protocol. Also, by exploiting the view change protocol and the chain selection rules, he can revert arbitrarily long chains within an epoch [10].
- $x > 2f$: The attacker has complete control over the network: he can delay it, create forks and publish invalid blocks.

A rational adversary will not produce forks or invalid blocks, or deviate from the protocol in any way, unless he owns more than f of the validators, since their misbehavior is easily observable and it would result in loss of the rewards. Only malicious adversaries would deviate from the protocol in this situation and, by our assumption, they must control f validators or less.

Note that, if the rational adversary owns more than f validators, he will certainly perform double-spending attacks since he can gain money from the attack and still receive the validator reward. So double-spending would be the profit-maximizing behavior for him.

C. Adaptive adversary

Next, we discuss adaptive adversaries. These are adversaries that can only corrupt a given number of nodes but, at any time, can change which nodes are corrupted.

We only need to consider the case in which the adversary can corrupt at most f validators. If he can corrupt more than that, he can already compromise the consensus algorithm in the static case.

The simplest way of attacking Albatross in the adaptive case is for the adversary to be always corrupting the slot owner and refusing to produce and propose blocks, thus preventing the algorithm from making any progress.

To achieve this, however, the adversary needs to know the next slot owner before he has a chance of producing a block. Since the slot owner for a given slot is only selected in the previous block and the slot owner does not require any interaction with other validators to produce a block, *the adversary must learn who is the slot owner before the slot owner himself does.*

Naively, the attacker can create many nodes in the network so that he can receive a block before the next slot owner does. That will give him an antecedence of roughly the *block propagation time* over the slot owner. The attacker would thus need to compromise the next slot owner during this short period of time.

Another strategy for him is to wait for his turn to produce a block. Now he can learn the identity of the next slot owner before he gossips the block. However, he still needs to gossip the block before a view change happens, or another validator

[10]To perform this attack the adversary needs to be the first owner in a slot for a micro block. Lets also imagine that he controls $f + 1$ validators. Then he withholds his block until he receives view change messages from at least f other nodes. His validators will produce view change messages but not broadcast them. Finally, he will release his block. Now, until the end of the epoch, he can release the view change messages that he withheld. This will result in a total of $2f+1$ view changes being broadcast and a new block being produced for that slot. Since the new block has a higher priority according to the chain selection rules, the network will ignore the previous fork and start building on the new block.

will produce a block and the slot owner will change. Using this technique the attacker now can have an antecedence equal to the timeout Δ.

After the attacker corrupts the first slot owner, because the timeout increases linearly, he has Δ time to corrupt the second slot owner, 2Δ time to corrupt the third, and so on.

Strictly speaking, an adaptive adversary can corrupt nodes instantly, but a more realistic model considers that it takes some time to corrupt a node. Independently of the strategy used, in Albatross an adaptive attacker would need to corrupt nodes on the order of seconds.

D. Network partition

From the *CAP theorem* [15], we know that when suffering a network partition a blockchain can only maintain either consistency or availability. PBFT favors consistency over availability and will stop in the presence of a network partition. Albatross also favors consistency, but can still produce a few micro blocks before stopping.

Note that, if the network is split in half, it is possible for one half to contain the owners of the next z slots. In this case, z blocks will be produced and then a view change will be attempted (because the next slot owner is part of the other partition) but will fail because it needs $2f + 1$ view change messages. Hence, as a result of the network partition, one half will immediately stop, while the other half will produce one or more blocks before stopping.

When the network partition ends, Albatross will quickly resume its normal operation. The z blocks produced by one half will be accepted by the other half and then the nodes can start producing blocks from there.

It is worth noting that when the network splits into two parts, if one of the parts has $2f+1$ or more *rational* validators, then Albatross is potentially able to continue normally, preserving both consistency and availability.

E. Probabilistic finality

When receiving a transaction it is important to know if the transaction can be reversed because of a fork. Albatross only offers provable finality at macro blocks, which might be too far apart to be useful for most use cases. However, Albatross does have very strong probabilistic finality.

Only malicious validators will create, or build on, a fork or an invalid block. Such a series of *illegal* blocks is called a *malicious subchain*.

Since every slot has a specific owner, as soon as we reach a slot controlled by a rational validator, the malicious subchain is resolved. There is no way for the subchain to continue past that slot. So, the only way for an attacker to create a malicious subchain of length d is for him to be the slot owner for d slots in a row.

The main security assumption of Albatross is that, if we have a validator list of size $n = 3f + 1$, then there are at most f malicious validators (or colluding rational validators). Thus, for our analysis, we assume the worst case of an attacker controlling f validators. Given that the slot owner selection is random, the probability of the attacker being a slot owner for d slots in a row is:

$$P(d) = \left(\frac{f}{n}\right)^d \approx \left(\frac{n/3}{n}\right)^d = \left(\frac{1}{3}\right)^d = 3^{-d} \tag{4}$$

This means that the probability of a transaction being reverted, because it is on a malicious subchain, decreases exponentially. Based on the worst case scenario, a client can easily calculate the probability that a transaction is final by taking into account the number of blocks built on top of the block that includes the transaction.

We can see from table VII-E that a certainty of 99.9% is reached after only 6 blocks (including the block containing the transaction).

TABLE III
PROBABILITY OF A TRANSACTION BEING FINAL AFTER *n* BLOCKS

Blocks	1	2	3	4	5	6
P (%)	66.6	88.9	96.3	98.8	99.6	99.9

F. Transaction censorship

Lastly, we want to discuss transaction censorship. Since the client can gossip a transaction to the entire validator list, as long as there is one honest active validator, there is a high probability that a block containing the transaction will be produced. The transaction can still be censored by creating a view change, given that the censor has control of $2f + 1$ validators.

Hence, as long as there is a single honest active validator and the censor does not control $2f+1$ validators, a transaction cannot be censored indefinitely.

VIII. UPGRADES

When a new software version of a blockchain is released it is said to be a fork and it is normally divided in one of two types: soft forks and hard forks.

A soft fork changes the protocol in such a way that it *strictly reduces* the set of valid transactions. In other words, all transactions that are valid in the new version are also valid in the old version, but the converse is not true.

A hard fork in the other hand introduces transactions that are not valid in the old version. Transactions that were valid in the old version may or may not be valid in the newer version, and that distinguishes between *strictly expanding* hard forks and *bilateral* hard forks.

The main difference between the two is that in hard forks the users need to upgrade to the new version in order to remain in the chain, soft forks don't require the users to upgrade. The consequence of this is that hard forks are much more likely to lead to chain splits than soft forks.

However, this is only true for blockchains that prefer *availability* over *consistency*, for example in proof-of-work blockchains. But for Albatross and other PBFT-like blockchains, and in fact for any blockchain that prefers *consistency* over *availability*, it is not possible to have chain splits.

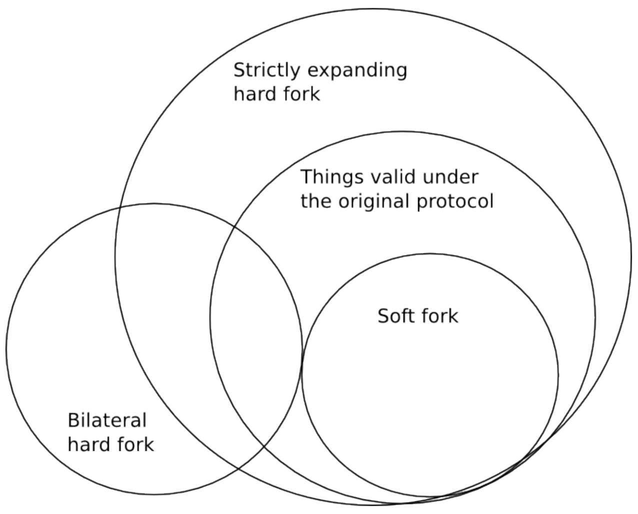
Strictly expanding hard fork
Things valid under the original protocol
Soft fork
Bilateral hard fork

Fig. 1. Venn diagram illustrating the different types of forks

So it does not make sense to speak of soft forks and hard forks, instead we will simply call them upgrades.

For simplicity, we will define that upgrades in Albatross only happen during macro blocks. This makes the rest of this analysis simpler and also more relevant for other PBFT-like consensus algorithms.

To upgrade the blockchain, the following protocol would be run:

1) The macro block leader proposes a block with the new version rules.
2) If he can get the required $2f+1$ signatures, then the block becomes part of the blockchain and the blockchain is upgraded.
3) Otherwise, the leader will be subject to a view change and the new leader will just propose a block using the old version rules.

Note that in no circumstance there is a chain split, the blockchain will either upgrade or not, but it will remain as one chain. Of course, allowing the validators who did not upgrade their clients to stay in the validator set is dangerous and could cause the chain to halt [11]. These validators that refuse to upgrade, even though they were behaving honestly *prior* to the upgrade and have a right to refuse to upgrade, become malicious validators *after* the upgrade.

Thus it is logical to expel the non-upgraded validators from the validator set, and letting them rejoin after they upgrade. This leads to a better upgrade protocol:

1) A new software client is released and an upgrade is proposed on-chain.
2) Validators that download the new client also publish on-chain their intention to support the upgrade.
3) During macro blocks the leader evaluates if more than $\frac{2}{3}$ of the validator set has stated that they support the upgrade.
4) If not, then the macro block leader proposes a block with the old version rules. If yes, he proposes a block with the new version rules.
5) If he cannot get the required $2f+1$ signatures, then a view change will happen and the new leader will propose a block with the old version rules.
6) Otherwise, if he can get the required $2f+1$ signatures, then the block becomes part of the blockchain and the blockchain is upgraded.
7) After the upgrade, all validators that did not support the upgrade are automatically unstaked. The validator list then is sampled only from the validators that supported the upgrade.

This new protocol, besides stopping the blockchain from halting, also prevents validators from proposing failing upgrades by allowing validators to signal their support or opposition to the future upgrade.

However, by expelling the validators that do not wish to upgrade, we may also end up increasing the fraction of malicious validators. Since malicious validators can, by definition, act arbitrarily we have to assume the worst scenario. In this case, the worst scenario is that all of the malicious validators choose to upgrade.

Of course, the network could halt if the fraction of malicious validators *post-upgrade* is larger than $\frac{1}{3}$ of the validator set. If we require the fraction of malicious validators to be smaller than $\frac{1}{3}$ after the upgrade, what is the maximum fraction of malicious validators that we can have before the upgrade?

Let us imagine that an upgrade is only attempted if at least a fraction t of the validator set publicly supports it. Furthermore we will assume that all of the malicious validators support the upgrade. Since we require that exactly $\frac{1}{3}$ of the validator set *post-upgrade* is malicious, then we can divide the *pre-upgrade* validator set in three fractions:

- $(1-t)$ of validators that do not support the upgrade.
- $\frac{1}{3}t$ of validators that support the upgrade and are malicious.
- $\frac{2}{3}t$ of validators that support the upgrade and are rational or honest.

Given all of the above information, we can state the following conjecture:

Imagine a partially synchronous Byzantine tolerant consensus algorithm that favors consistency over availability and that only upgrades whenever a fraction t of the nodes support it. The upgrade can only be executed safely if the fraction of malicious nodes is smaller than $\frac{1}{3}t$.

Given that $t \leq 1$ we can conclude that any PBFT-like consensus algorithm is weaker during an upgrade.

For Albatross specifically, the threshold t must also be larger than $\frac{2}{3}$ since otherwise the nodes that oppose the upgrade can refuse to finalize any block containing the upgrade. But if the

[11] Since the validator list is sampled at random from the larger validator set, you could end up in a situation where the validators that refused to upgrade are less than $\frac{1}{3}$ of the validator set but more than $\frac{1}{3}$ of the validator list. Since those validators would still be operating under the old rules, they would refuse to build on the new chain which would eventually result in the chain halting.

threshold is much larger than $\frac{2}{3}$ then malicious validators can stop any upgrade by simply refusing to support it. In the limit, when $t = 1$ a single malicious validator can force the chain to never be upgraded.

We propose a threshold of $\frac{2}{3}$ which means that, during an upgrade, Albatross is only secure up to $\frac{2}{9}n$ malicious validators.

IX. Performance analysis

In this section, we give a brief theoretical analysis of the performance of Albatross. We show that, in the *optimistic* case, it achieves the theoretical limit for single-chain PoS algorithms, while in the *pessimist* case it still achieves decent performance.

A. Optimistic case

The best possible case is when the network is synchronous (all messages are delivered within a maximum delay d), the network delay d is smaller than the timeout parameter Δ and all validators are honest.

The macro blocks have a message complexity of $\mathcal{O}(n^2)$, since they are produced with PBFT, but they constitute a very small percentage of all blocks so, the overall performance is mostly correlated with the micro block production. Moreover, approaches such as Handel [16] can be used to reduce this message complexity.

Micro blocks have a message complexity of $\mathcal{O}(1)$. In fact, they only require the propagation of the block. The latency, if we ignore the time spent verifying blocks and transactions, is equal to the block propagation time, which is on the order of the network delay d.

In conclusion, Albatross, in the optimistic case, produces blocks as fast as the network allows it.

B. Pessimistic case

If we relax some of the assumptions made for the optimistic case, Albatross still has a performance superior to PBFT. There are three different cases that we will consider:

- **Malicious validators:** The worst case, while still maintaining security, is a scenario with f validators being malicious and refusing to produce blocks. In this case, we can expect one view change every three blocks. The view change protocol requires $\mathcal{O}(n)$ messages and waiting for a timeout. So, in this case, the message complexity will be $\mathcal{O}(n)$ and the latency will be on the order of Δ.
- **Network delay larger than timeout:** If $d > \Delta$ then, because the timeout increases linearly, every block will only be produced after a given number of view changes and several short-lived forks may be created for each block. In this case, because we still only rely on the view change protocol, the message complexity will be $\mathcal{O}(n)$ and the latency will be greater than d.
- **Partially synchronous network:** Under partial synchrony, there are periods where the network becomes asynchronous, before returning to synchrony. In this case, it is possible to completely halt progress of the blockchain while the network is asynchronous. However, Albatross will return to normal operation when the network becomes synchronous again.

X. Conclusion

In this paper we described and analyzed Albatross, a novel consensus algorithm inspired by speculative BFT algorithms. To achieve Albatross we modified PBFT in three main ways: (1) making it permissionless by selecting a validator list proportionally to stake, (2) increasing resistance to adaptive adversaries by only selecting block producers on the previous block using a VRF and (3) increasing performance by relying on speculative execution of blocks.

Despite sacrificing provable finality, Albatross has a strong probabilistic finality which, when coupled with low block latency, means that transactions can have a very high probability of being final in just a few seconds.

References

[1] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[2] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 4, p. 7, 2009.

[3] 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," 2016.

[4] J. Chen and S. Micali, "Algorand," 2016.

[5] I. Eyal, A. E. Gencer, E. G. Sirer, and R. van Renesse, "Bitcoin-ng: A scalable blockchain protocol," 2015.

[6] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Advances in Cryptology–ASIACRYPT 2001*. Springer, 2001, pp. 514–532.

[7] I. Baumgart and S. Mies, "S/kademlia: A practicable approach towards secure key-based routing," in *2007 International Conference on Parallel and Distributed Systems*. IEEE, 2007, pp. 1–8.

[8] T. Baignres, C. Delerable, M. Finiasz, L. Goubin, T. Lepoint, and M. Rivain, "Trap me if you can – million dollar curve," Cryptology ePrint Archive, Report 2015/1249, 2015, https://eprint.iacr.org/2015/1249.

[9] J. Bonneau, J. Clark, and S. Goldfeder, "On bitcoin as a public randomness source," Cryptology ePrint Archive, Report 2015/1015, 2015, https://eprint.iacr.org/2015/1015.

[10] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," Cryptology ePrint Archive, Report 2016/1067, 2016, https://eprint.iacr.org/2016/1067.

[11] D. Boneh, B. Bnz, and B. Fisch, "Batching techniques for accumulators with applications to iops and stateless blockchains," Cryptology ePrint Archive, Report 2018/1188, 2018, https://eprint.iacr.org/2018/1188.

[12] S. Bowe, J. Grigg, and D. Hopwood, "Halo: Recursive proof composition without a trusted setup," Cryptology ePrint Archive, Report 2019/1021, 2019, https://eprint.iacr.org/2019/1021.

[13] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," Cryptology ePrint Archive, Report 2014/595, 2014, https://eprint.iacr.org/2014/595.

[14] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[15] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[16] O. Bgassat, B. Kolad, N. Gailly, and N. Liochon, "Handel: Practical multi-signature aggregation for large byzantine committees," 2019.