Albatross

An optimistic consensus algorithm

Bruno França*, Marvin Wissfeld†, Pascal Berrang†, Philipp von Styp-Rekowsky† and Reto Trinkler*
* Trinkler Software
company@trinkler.software
† Nimiq Foundation
research@nimiq.com

Version 3 – March 4, 2019

***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 a strong confirmation, and instant confirmations can be achieved as well. 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 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. Namely, it is inspired by Byzcoin [3] and Tendermint [4] for their method of making PBFT permissionless, Algorand [5] for its resistance to adaptive adversaries, and Ouroboros-BFT [6] for its simplicity.

II. Overview

In this section, we will give a summary of Albatross, describing the different types of validators, blocks, and how the chain selection works intuitively. We also present the behavior of the protocol in optimistic mode and discuss the ways validators may misbehave. We describe how such misbehavior is countered in our protocol. We will focus on the general structure of Albatross and leave the details for Section III.

A. Validators

There are two types of validators: potential and active validators. Given a certain point in time, 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 and thus has an active part during the current *epoch*. An *epoch* is the period of time for which a set of active validators was selected.

The set of all active validators is called the *validator list*. Similar to other hybrid consensus algorithms, the validator

[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.

list is chosen at random from all potential validators and the probability of a validator being chosen is proportional to its stake. A validator can occur more than once in this list.

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 only the identities of the new active validators and the random seed used to select them. Macro blocks are produced with PBFT.
- **Micro blocks:** These blocks are the ones that contain the user transactions. Each micro block is produced by a verifiably 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 honest validators. We call this the *optimistic mode*, because we are trusting that the validators will not misbehave.

1) A macro block was produced containing a random seed, which was used to randomly select the new validator list proportionally to the validators' stake. It was also used to choose which validator, from the new validator list, is allowed to produce the first micro block in the following epoch.
2) The chosen validator, also called the *slot owner*, now produces his micro block. He first chooses which transactions to include and calculates the resulting state. Then, he produces the next random seed using a *verifiable random function* (VRF) that takes as an input the previous block's seed. This new random seed is used to select, from the validator list, which validator will produce the next block. Finally, he includes the transactions, the current state, and the random seed in the block, signs it and relays it.
3) 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.
4) The PBFT leader will also produce a new random seed, by applying the VRF to the last block's random seed, which will be used to select a new validator list. The leader signs this information and multicasts it to the other validators. The signature is not part of the block and only required to ensure integrity of the proposal. The validators then collect the two rounds of signatures necessary for the PBFT protocol. Lastly, everything is combined into the block and relayed.

We propose to instantiate the VRF used in our protocol by BLS signatures. This way, producing the next random seed boils down to signing the previous random seed with the validator's secret key. In the Appendix A we will give details on the VRF construction we use.

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) 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. Additionally, they will ignore any more blocks from that validators during the current slot, this helps prevent DoS attacks.

2) 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 *slash inherents*[2]. A slash inherent confiscates the stake of a validator that produced a fork. Anyone can create a slash inherent; they only need a proof that the validator forked. Two block headers, at the same slot, signed by the same validator, are a sufficient proof.

If a malicious validator creates or continues a fork, eventually a honest (or rational) validator will produce a block containing a slash inherent and the malicious validator will lose his entire stake. The slash inherent will also reward the other validators in the current epoch and thus incentivize those to report such forks.

3) 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

[2]An inherent is a type of extrinsic. Extrinsics are any input to the state transition function. Transactions are extrinsics that are propagated through the network and signed. Inherents are neither propagated nor signed. An example of an inherent is a timestamp.

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 (starting with 0) 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.

III. Specification

We will now give the technical specification of Albatross, focusing specifically on: networking, validator signaling and selection, block format, slash inherents, view change protocol, chain selection, and rewards. These specifications aim at implementing Albatross as a consensus module on Parity's Substrate, but can easily be adapted to other frameworks.

A. Networking

Substrate uses 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, for the rest of this paper, we differentiate between two types of communication:

- **Multicast:** When a node sends a message directly to a specific set of other nodes.
- **Gossip:** 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.

Transactions are always multicasted to the validator list (or a subset thereof), and then the active validators forward it to the potential validators. There is no need to burden the network by gossiping a transaction since it is only of interest to the validators. Also, this makes *instant confirmation* possible (see Section IV-C).

Blocks are always gossiped. In a micro block, the validator who produced the block initiates the gossip. In a macro block, all validators that successfully finish the PBFT protocol initiate the gossip.

B. 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 start, continue or stop being validators.

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

- **Deposit:** The amount of tokens to be locked up while the node is a validator.
- **Validating key:** A BLS public key (see Appendix A), which is created by the node for the sole purpose of signing blocks and producing new random seeds. It is important that the BLS keypair is chosen prior to being a validator to prevent malicious validators from biasing the next random seed.
- **Proof of knowledge of secret key:** This is a signature of the validating key using the corresponding secret key. It is used to prevent rogue key attacks.

The result of the staking transaction is to transfer the desired amount of tokens from the node's account to a special lock account and adding his validating key 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.

The reason to have a separate key just for signing blocks is operational security. Having two keys, one for signing blocks and one for accessing the account, allows validators to be online to sign blocks and still keep their funds in a cold wallet.

2) Restaking: When a node becomes a validator a countdown of q epochs starts, where q is a predefined constant. At the end of those q epochs, the node stops being a potential validator.

For a node to maintain its validator status, it must send a *restaking* transaction, signed with the validating key, which consists of these two fields:

- **New validating key:** A new validating key.
- **Proof of knowledge of secret key:** The proof of knowledge for the new validating key.

After a *restaking* transaction gets accepted, the countdown resets and the potential validator registry is updated with the new validating key.

The restaking transaction guarantees that old nodes that went offline are expelled from the validator registry, so that it only contains live nodes.

3) Unstaking: Nodes may wish to stop being validators before the countdown of q epochs ends. In this case, 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 can be withdrawn by the node and the validating key is deleted from the registry.

However, if the unstaking transaction is sent while the node is still an *active* validator, then he is required to remain a validator until the end of the epoch and his deposit can only be withdrawn m blocks after the end of the epoch. Delaying the return of the deposit is required to allow the new validator list to punish past validators who misbehaved right before the end of the epoch.

C. Validator selection

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

1) 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 validating keys and the corresponding deposit amount of every potential validator. Then, we order the validating keys in a deterministic way (for example, by lexicographic order). Lastly, we map the ordered keys to their deposit amount, such that the deposit amount represents a range. For example, if there are 10 tokens staked by pk_a, 50 tokens by pk_b and 15 tokens by pk_c, then the mapping would be as follows:

Key	Deposit	Range
pk_a	10	$[0, 9]$
pk_b	50	$[10, 59]$
pk_c	15	$[60, 74]$

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

Algorithm 1 validator list selection algorithm

validator list $= \varnothing$
$S =$ random seed
$t =$ total amount staked
$i = 0$
while validator list not full **do**
  $r = hash(S \mid\mid i) \mod t$
  v = potential validator whose range contains r
  add v to validator list
  $i++$
end while

2) Slot owners: Given a random seed S and a validator list, we can randomly choose an infinite ordered list of validators to produce the next block. 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 of the first validator not responding in time (see Section III-F).

The algorithm for calculating this infinite ordered list is as follows: First, we take the validating keys 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

$S =$ random seed
$n =$ validator list size
$i = 0$
loop
  $r = hash(S \mid\mid i) \mod n$
  $v =$ active validator numbered$r$
  add v to slot owner list
  $i++$
end loop

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

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 newspaper headlines. We just take that information, hash it and convert it into an elliptic curve point. To reduce any possibility of bias in the initial seed, distributed randomness generation algorithms can be employed.

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.

D. Block format

We model the format of both macro and micro blocks similarly to Substrate. In Substrate, blocks are composed by:

- **Header:** The block header.
- **Extrinsics:** The data to be input into the state transition function. Includes *transactions* and *inherents*.
- **Justification:** The information necessary to make the block valid according to the consensus rules. For example, the validator's signature of the header.

While the header consists of the following components:

- **Parent hash:** The hash of the previous block header.
- **Digest:** A field containing auxiliary data that may be necessary for light-clients.
- **Extrinsics root:** The root of the Merkle tree of the extrinsics.
- **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.

For the header, the only field that needs further description is the digest. For the body, we will explain both the extrinsics and the justification in more detail.

1) Macro blocks:

- **Digest:** The digest contains the validating keys of the new validator list. This is necessary for light-clients to sync. In addition, it contains the hash of previous macro block header, the block number, and the *view change* number (see Section III-F).
- **Extrinsics:** The extrinsics of a macro block do not list any transactions but only contain inherents. Namely, this field holds the timestamp and the random seed. Also, it may contain aggregated *view change* messages.
- **Justification:** Since this is a PBFT block, the justification consists of the two rounds of validator signatures (see Appendix B).

[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 [9], are possible as well.

2) Micro blocks:

- **Digest:** The digest consists of the block number, the *view change* number, and some identifier of the validator that produced the block.
- **Extrinsics:** The extrinsics field includes the transactions, the timestamp, and the random seed. Also, it may contain aggregated *view change* messages and/or *slash inherents*.
- **Justification:** The justification consists only of the signature of the validator that produced the block.

E. Slash inherents

When a validator creates two or more micro blocks in the same slot, he is punished by having his stake 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 slash inherent. It is easier to prove that two blocks exist in the same slot than that two *valid* blocks exist in the same slot.

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

- The block headers must have the same digest.
- The justifications must be valid.

This essentially proves that a validator created, or continued, a fork. When a slash inherent is included in a block, it results in the following consequences:

- 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 macro block voting.
- His stake is confiscated and divided equally between the rest of the validator list. This creates an incentive for other validators to report forks.
- His validating key is deleted from the validator registry.

A slash inherent only punishes a single validator. If there are multiple misbehaving validators, several slash inherents have to be included in the same block.

F. View change protocol

If a validator does not produce a block during his slot, for some reason, 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), an infinite list of slot owners $[s_1, s_2, \ldots]$ and a timeout parameter Δ [4], each active validator runs **Algorithm 3** immediately after receiving a block. A *view change* number keeps track of current index within the infinite list.

Algorithm 3 View change algorithm

```
i = 0 (view change number)
loop
    wait for (i + 1) · Δ time
    if a valid block was received from s_i then
        terminate algorithm
    else
        multicast a view change message to all other active validators
    end if
    if 2f + 1 view change messages are received then
        commit to not accepting a block from s_i in this slot
        i++
    end if
end loop
```

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 a statement $\langle \text{VIEW-CHANGE}, i+1, b \rangle$, where i is the current view change 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.

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

G. 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 change number.
3) The chain with the most blocks.

Still, it is possible for two chains to tie on all three conditions. 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.

[4]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.

H. Rewards

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

However, validators could bias the selection of the validator list and the slot owners by refusing to produce a block, thus doubling the probability of a favorable outcome. Hence, we disincentivize such behavior by giving the entire block reward to the block producer or, in the case of macro blocks, the block proposer.

IV. Features

In this section we discuss some miscellaneous features of Albatross.

A. Bootstrapping

New nodes, especially light clients, who want to join the network need a way of downloading the blockchain. Full nodes need to download and verify all the blocks, except for the *genesis* block which is hardcoded into the client software.

Light clients can use a faster method of bootstrapping since they do not need to verify the entire blockchain. They only need to download all the macro blocks since genesis, which are necessary to learn the current validator list, and all the micro blocks since the last macro block. Because there is a set number of micro blocks between any two macro blocks, the amount of data that a new light client needs to download only grows with each new macro block.

It is also possible to hardcode a recent macro block into the light client software, thus reducing the amount of data necessary to bootstrap. With periodic releases, this method makes that amount of data constant. Considering that the client software needs to be verified or downloaded from a trusted party anyway, this does not have an immediate impact on the security.

B. Strong confirmation

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

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. 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:

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

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 the following table that a certainty of 99.9% is reached after only 6 blocks (including the block containing the transaction).

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

C. Instant confirmation

It is possible to get even faster confirmation, at the expense of public verifiability. We can use the fact that transactions are directly sent to the validator list by the client, to give the client a very high confidence that his transaction will be accepted.

When the client sends his transaction, he can also ask for a *receipt* from the validators, which is a signed message from a validator stating that the transaction was added to his mempool. Honest validators will not send fake receipts so, if the client receives several receipts, he can be reasonable sure that his transaction will eventually be included into the blockchain.

Instant confirmation is also very fast, in theory taking only the time of a round-trip message between the client and the validators.

However, the receipts can not be used to prove publicly that the transaction will be accepted, since a client can send conflicting transactions and receive receipts for both.

V. 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, long-range attacks, 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:

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

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 [10]. Thus, we are interested in the maximum fraction of stake p an adversary may control before exceeding this bound:

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

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

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. The following table presents the result of our calculations based on varying numbers of n:

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. We will argue that Albatross is secure as long as the above assumption is true.

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 *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.
- $x > 2f$: The attacker has complete control over the network: he can delay it, create forks and publish invalid blocks.

Rational adversaries will not produce forks or invalid blocks, even if they own more than f of the validators, since their misbehavior is easily observable and it would have a negative impact the value of the token thus making their stakes worthless. Only malicious adversaries would deviate from the protocol in this situation and, by our assumption, they must control f validators or less.

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 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* [11], 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 continue normally, preserving both consistency and availability.

E. Long-range attack

Long-range attacks deal with cases where an adversary tries to create a fork starting in an old block, maybe even at the genesis block. In this type of attack, an adversary would try to gain control of $\frac{2}{3}$ of the validator list retrospectively. There are two possibilities how that can be achieved. One option is that the adversary obtains control of $\frac{2}{3}$ of the validator list at some point, behaves honestly, and then unlocks his stake and sells all the tokens. The second option is for the adversary to bribe old validators that already sold their stake to obtain $\frac{2}{3}$ of the validator power in the past. After having obtained this control, he is now able to go back to any block produced during his $\frac{2}{3}$ majority and start a fork.

This attack only affects new nodes and nodes that have not synced with the network past the block where the fork occurred. However, these nodes can see that there is a fork and, because they know that one of them is necessarily malicious, they will refuse to sync with the network. All other nodes will not be affected by this attack.

A long-range attack is a difficult and improbable attack that does little damage to the network. Still, in the case of it happening, the community can join together to produce a checkpoint and, thus, resolve the fork.

F. Transaction censorship

Lastly, we want to discuss transaction censorship. Since the client can multicast 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.

VI. 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 [12] 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.

VII. 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 strong consistency, Albatross has a strong confirmation 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. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.6130

[2] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 45–58. [Online]. Available: https://www.cs.cornell.edu/lorenzo/papers/kotla07Zyzzyva.pdf

[3] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 279–296. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_kokoris-kogias.pdf

[4] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *arXiv preprint arXiv:1807.04938*, 2018. [Online]. Available: https://arxiv.org/abs/1807.04938

[5] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 51–68. [Online]. Available: https://dl.acm.org/ft_gateway.cfm?id=3132757&type=pdf

[6] A. Kiayias and A. Russell, "Ouroboros-BFT: A simple byzantine fault tolerant consensus protocol," 2018. [Online]. Available: https://eprint.iacr.org/2018/1049.pdf

[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. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.68.4986

[8] T. Baigneres, C. Delerablée, M. Finiasz, L. Goubin, T. Lepoint, and M. Rivain, "Trap me if you can-million dollar curve." *IACR Cryptology ePrint Archive*, vol. 2015, p. 1249, 2015. [Online]. Available: https://eprint.iacr.org/2015/1249.pdf

[9] D. Boneh, B. Bünz, and B. Fisch, "Batching techniques for accumulators with applications to iops and stateless blockchains," Cryptology ePrint Archive, Report 2018/1188, Tech. Rep., 2018.

[10] 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. [Online]. Available: http://people.cs.uchicago.edu/~shanlu/teaching/33100_wi15/papers/byz.pdf

[11] 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. [Online]. Available: https://courses.e-ce.uth.gr/CE623/CAP_theorem_proof.pdf

[12] N. Gailly, N. Liochon, O. Bgassat, and B. Kolad, "Handel: Practical multi-signature aggregation for large byzantine committees," 2019. [Online]. Available: https://github.com/ConsenSys/handel

[13] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2001, pp. 514–532. [Online]. Available: https://www.iacr.org/archive/asiacrypt2001/22480516.pdf

Appendix

A. Boneh-Lynn-Shacham signatures

Boneh-Lynn-Shacham (BLS) is a signature scheme that was first introduced in 2004 [13] 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 $x \xleftarrow{R} \mathbb{Z}_p$ 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 $e(\sigma, Q) = e(H, Y)$.

Another feature that we use is its ability to act as a verifiable random function. 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.

B. 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 deterministic BFT protocol. There are several slightly different variants of PBFT and 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 justification, we can save space by aggregating them. This way, all the prepare messages are aggregated into a single BLS 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.