Ethereum Proof-of-Stake under Scrutiny (Extended Version)

Ulysse Pavloff, Yackolley Amoussou-Guenou, Sara Tucci-Piergiovanni

Ethereum Proof-of-Stake and the Probabilistic Bouncing Attack

ULYSSE PAVLOFF, Université Paris-Saclay, CEA, List, France
YACKOLLEY AMOUSSOU-GUENOU, Université Paris-Saclay, CEA, List, France
SARA TUCCI-PIERGIOVANNI, Université Paris-Saclay, CEA, List, France

Ethereum has undergone a recent change called *the Merge*, which made Ethereum a Proof-of-Stake blockchain shifting closer to BFT consensus. Ethereum, which wished to keep the best of the two protocol designs (BFT and Nakomoto-style), now has a convoluted consensus protocol as its core. The result is a blockchain being possibly produced in a tree-like form while participants try to finalize blocks. We categorize different attacks jeopardizing the liveness of the protocol. The Ethereum community has responded by creating patches against some of them. We discovered a new attack on the patched protocol. To support our analysis, we propose a new high-level formalization of the properties of liveness and availability of the Ethereum blockchain, and we provide a pseudo-code. We believe this formalization to be helpful for other analyses as well. Our results yield that the Ethereum Proof-of-Stake has safety but only probabilistic liveness. The probability of the liveness is influenced by the parameter describing the time frame allowed for validators to change their mind about the current main chain.

CCS Concepts: • **Theory of computation → Distributed algorithms**; • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

Additional Key Words and Phrases: Ethereum Proof-of-Stake, Liveness, Availability, Bouncing attack

1 INTRODUCTION

Ethereum has recently undergone a major change in its protocol, successfully passing from proof-of-work to proof-of-stake. The change underpins an entirely new consensus protocol, which brings Byzantine fault-tolerance to Ethereum. The main design goal is to keep using a Nakamoto-style consensus, i.e., a protocol that constantly creates blocks in a tree-like form and selects a branch as the current chain using a fork-chain rule. However, a mechanism (called finality gadget) incrementally finalizes blocks in the chain as opposed to pure Nakamoto-style consensus. A *finalized block* is a block that is voted by at least two-thirds of validators[1]. In a system with less than one-third of Byzantine validators, a finalized block is never revoked.

Interestingly, this design aims at guaranteeing, at the same time, the availability of the chain (to let new blocks be continued added) and consistency, i.e., uniqueness of a finalized chain's prefix. Note that classical BFT consensus protocols re-adapted to blockchains such as Tendermint [6] for the Cosmos blockchain [17] or Tenderbake [5] for Tezos blockchain [15], finalize one block at the time: for each height of the blockchain only one block is ever added. On the other hand, Ethereum Proof-of-Stake (PoS) builds a common prefix, but the suffix can change: for a given height of the blockchain, different blocks can be seen at that height over time. The advantage of this approach is to always make progress, regardless of Byzantine behavior and network partitions, while classical BFT consensus protocols stop producing blocks during asynchronous periods and attacks. Ethereum PoS tries then to provide consistency without renouncing availability. This seems to be in striking contrast with the CAP theorem [14], which states that it is impossible to guarantee progress and consistency in the case of network partitions. The caveat here is that Ethereum PoS maintains a data structure where the prefix is consistent and finalized only when possible, while the suffix can grow without being consistent. The resulting protocol, however, is quite involved. For this reason, we aim to provide a formal ground for analysis of the Ethereum PoS protocol.

[1]To become a validator, one needs to "stake" an amount of 32 ETH (the native cryptocurrency of the blockchain).

We first provide a novel formalization of the Ethereum PoS blockchain properties, which are a combination of properties of Nakamoto-style blockchains and BFT-style ones. We then provide a high-level formalization of the protocol itself through pseudo-code. Formalization allows for analysis of the protocol in terms of its properties.

In this paper, we study the security and the liveness of the protocol. Liveness is the ability to always finalize new blocks and security is the impossibility of having two checkpoints finalized on different chains. Our analysis reveals a new possible attack on the protocol's liveness. Indeed previous work has already pointed out the risks of a so-called *bouncing attack* on liveness [19]. A bouncing attack is an attack that prevents the chain from being finalized because the main chain selected through the fork choice rule continually bounces between two alternative branches. After the attack was identified, the Ethereum community responded by implementing a patch to the protocol to prevent this attack. The patch aims at mitigating bouncing by forcing validators to stick to a chain after a while. Interestingly, we managed to find a new bouncing attack on the patched version of the protocol. We found that the bouncing can be repeated over time but with decreasing probability of success. This shows that the liveness of the patched protocol is probabilistic. Note that the attack is plausible in a Byzantine environment since it only relies on the Byzantine validator's capacity to withhold votes and to release them at the right time to make honest validators change their mind on the chosen chain.

The paper is organised as follows: Section 2 elaborates on the system model while Section 3 defines the properties essential to our formalization. In Section 4, we explain and formalize the Ethereum PoS protocol providing all the materials necessary for its understanding, including pseudo-code. Section 5 presents a new liveness attack still possible in the current version of the Ethereum PoS protocol. Section 6 proves the safety of Ethereum PoS. We present the related works in Section 7, and categorize the different types of attacks in Section 8. We conclude in Section 9.

2 SYSTEM MODEL

We consider a system composed of a finite set Π of processes called *validators*[2]. There are a total of n validators. Each validator has an associated public/private key pair for signing and can be identified by its public key. We assume that digital signatures cannot be forged. Validators have synchronized clocks[3]. Time is measured by periods of 12 seconds called *slots*, a period of 32 slots is called an *epoch*.

Network. Processes communicate by message passing. We assume the existence of an underlying broadcast primitive, which is a best effort broadcast. This means that when a correct process broadcasts a value, all the correct processes eventually deliver it. Messages are created with a digital signature.

We assume a *partially synchronous model* [9], where after some unknown Global Stabilization Time (GST), the system becomes synchronous, and there is a finite known bound Δ on the message transfer delay. Note that even if we have synchronized clocks, having an asynchronous network before GST still makes the system partially synchronous.

Fault Model. Validators can be *correct* or *Byzantine*. Correct validators (also called honest validators) follow the protocol, while Byzantines ones may arbitrarily deviate from the protocol[4]. We denote by f the number of Byzantine validators, with $f < n/3$.

[2]At the implementation level, validators are the processes with ETH staked that allow them to vote as part of the consensus protocol.
[3]Clocks can be offset by at most τ, this way, the offset can be captured as part of the network delay.
[4]Since in this paper we are only interested in the consensus part of the protocol, we only characterize validator's behavior. For clients submitting transactions, as in any blockchain, we assume they can be Byzantine.

3 BLOCKCHAIN PROPERTIES

In our analysis, we will continuously use the term Ethereum Proof-of-Stake as [26] to name the new protocol of Ethereum[5]. To begin our analysis of Ethereum Proof-of-Stake, we start by defining the terms and properties we will investigate.

Similarly to [3], we formalize the blockchain data structure as a *BlockTree*. Indeed the blockchain takes the form of a tree in which every node is a block pointing to its unique parent, and the tree's root is the *genesis block*. Among the different branches of the BlockTree, the protocol indicates a unique branch, or chain, to build upon with a so-called fork choice rule (e.g., the longest chain rule in Bitcoin). The selected chain is called the *candidate chain*.

*Definition 3.1 (**Candidate chain**).* We call **candidate chain** the chain designated as the one to build upon by the fork choice rule. Considering the view of the chain of an honest validator i, i's associated candidate chain is noted C_i.

The blocks in the candidate chain can be finalized or not.

*Definition 3.2 (**Finalized block**).* A block is finalized for a validator i if and only if the block cannot be revoked, i.e., it permanently belongs to the candidate chain C_i .

Note: It stems from the definition that all the predecessors of a finalized block are finalized.

*Definition 3.3 (**Finalized chain**).* The finalized chain is the chain constituted of all the finalized blocks.

Note: The finalized chain C_{fi} is always a prefix of any candidate chain C_i.

To analyse the protocol, one needs to examine the capability of the Ethereum Proof-of-Stake protocol to construct a consistent blockchain (safety), to allow validators to add blocks despite network partitions and failures (availability), and to make progress on the finalization of new blocks (liveness). These are paramount properties characterizing blockchains. Safety, availability, and liveness are expressed as follows:

*Definition 3.4 (**Safety**).* A blockchain is consistent or **safe** if for any two correct validators with a finalized chain, then one chain is necessarily the prefix of the other. More formally, for two validators i and j with respective finalized chain C_{fi} and C_{fj}, then C_{fi}'s is the prefix of C_{fj} or vice-versa.

*Definition 3.5 (**Availability**).* A blockchain is **available** if the following two conditions hold: (1) any correct validator is able to append a block to its candidate chain in bounded time, regardless of the failures of other validators and the network partitions; (2) the candidate chains of all correct validators are eventually growing, i.e., given a block b_k added to a candidate chain at a distance d from the genesis block b_0, where the distance is the number of blocks separating b_k from b_0, then eventually a block b_l will be added to the candidate chain at a distance $d' > d$.

*Definition 3.6 (**Liveness**).* A blockchain is **live** if the finalized chain is ever growing.

The fundamental difference between the finalized and the candidate chain lies in the fact that blocks of the finalized chain can never be revoked, while the candidate chain can change from one branch to another in the tree so that a suffix of blocks of the previously selected branch might be revoked. Availability, on the other hand, guarantees that adding blocks to the candidate chain is a wait-free operation whose time to complete does not depend on network failures or Byzantine behaviors. Availability also implies that blocks are constantly added in such as way that the height

[5]Other appellation such as Ethereum 2.0 or Consensus Layer can be found, we have chosen to stick with Ethereum Proof-of-Stake.

of the candidate chain eventually grows. This property avoids the pathological scenario in which all the blocks are added to the genesis block to form a star.

As in any distributed system, blockchains are faced with the dilemma brought by the CAP-Theorem [14]. This theorem states that no distributed system can satisfy these three properties at the same time: *consistency*, *availability*, and *partition tolerance*. Indeed, if network partitions occur, either the system remains available at the expense of consistency, or stops making progress until the network partition is resolved to guarantee consistency. This means that no blockchain can simultaneously be available and consistent. However, by maintaining the candidate and the finalized chain simultaneously, Ethereum Proof-of-stake aims to offer both safety and availability. The candidate chain aims to be available but without guaranteeing consistency all the time, while the finalized chain falls on the other side of the spectrum, guaranteeing consistency without availability. Therefore, the finalized chain will finalize blocks only when it is safe to do so where the candidate chain will still be available during network partitions (caused by network failures or attacks). The only caveat here is that the finalized chain grows by finalizing blocks of the candidate chain, which means that the properties of the two chains are interdependent. In particular, to assure liveness, it is necessary that the candidate chain steadily grows. This interdependence is a source of vulnerability that must be thoroughly analysed.

4 ETHEREUM PROOF-OF-STAKE PROTOCOL

4.1 Overview

The Ethereum Proof-of-Stake (PoS) protocol design is quite involved. We identify, similarly to [21], the objectives underlying its design as follows: (i) finalizing blocks and (ii) having an available candidate chain that does not rely on block finality to grow. To this end, the Ethereum PoS protocol combines two blockchain designs: a Nakamoto-style protocol to build the tree of blocks containing the transactions and a BFT finalization protocol to progressively finalize blocks in the tree. The objective is to keep the blockchain creation process always available while guaranteeing the finalization of blocks through Byzantine-tolerant voting mechanisms. The finalization mechanism is a *Finality Gadget* called *Casper FFG*, and the fork choice rule to select candidate chains is *LMD GHOST*.

Before introducing how the fork choice rule and the finality gadget work together, we will introduce the following basic concepts: (i) slots, epochs, and checkpoints, which set the pace of the protocol allowing validators to synchronize together on the different steps, (ii) committees formation and assignment of roles to validators as proposers and voters for each slot, and (iii) the different types of votes the validators must send in order to grow and maintain the candidate chain as well as the finalized chain.

In this section, we focus on providing a formal version of the protocol through pseudo-code, following the specification given by the Ethereum Foundation [11]. Every implementation of the protocol must be compliant with the specification. Note that a description of an initial plan of the protocol is proposed by Buterin et al. in [7]. In this paper, we describe and formalize the current implementation of the protocol [11].

4.1.1 Slots, Epochs & Checkpoints. In proof-of-work protocols, such as originally in [18], the average frequency of the block creation is predetermined in the protocol, and the mining difficulty changes to follow that pace. On the contrary, in Ethereum PoS, it is assumed that validators have synchronized clocks to propose blocks at regular intervals. More specifically, in the protocol, time is measured in *slots* and *epochs*. A slot lasts 12 seconds. Slots are assigned with consecutive numbers; the first slot is slot 0. Slots are encapsulated in *epochs*. An epoch is composed of 32 slots, thus lasting 6 minutes and 24 seconds. The first epoch (epoch 0) contains from slot 0 to slot 31; then epoch 1 contains slot 32 to 63, and so on. These slots and epochs allow associating the validators' roles to the corresponding time frame. An

essential feature of epochs is the *checkpoint*. A checkpoint is a pair block-epoch (b, e) where b is the block of the first slot[6] of epoch e.

4.1.2 Validators & Committees. Validators have two main roles: *proposer* and *attester*. The proposer's role consists in proposing a block during a specific slot[7]. This role is pseudo-randomly[8] given to 32 validators by epoch (one for each slot). The attester's role consists in producing an attestation sharing the validator's view of the chain. This role is given once by epoch to each validator.

In each epoch, a validator is assigned to exactly one committee (of attesters). A committee C_j is a subset of the whole set of validators. Each validator belongs to exactly one committee, i.e., $\forall j \neq k, C_j \bigcap C_k = \emptyset$ and for each epoch $\bigcup_i C_i = \Pi$. Each committee is associated with a slot. During this slot, each member of the committee will have to cast an *attestation* to indicate its view of the chain.

In short, during an epoch, validators are all attesters once and have a small probability of being proposers $(32/n)$. The roles of proposer and attester are entirely distinct, i.e., the proposer of a slot is not necessarily an attester of that slot.

4.1.3 Vote & Attestation. There are two types of votes in Ethereum PoS, the *block vote*[9] and the *checkpoint vote*[10]. The message containing these two votes is called *attestation*. During an epoch, each validator must make one attestation. The attestation ought to be sent during a specific slot. This slot depends on the committee of which the validator is a member. The two types of votes, checkpoint vote and block vote, have very distinct purposes. The checkpoint vote is used to finalize blocks to grow the finalized chain. The block vote is used to determine the candidate chain. Although validators cast their two types of votes in one attestation, an important distinction must be made between the two. Indeed, the two types of votes do not require the same condition to be taken into account. The checkpoint vote of an attestation is only considered when the attestation is included in a block. In contrast, the block vote is considered one slot after its emission, whether it is included in a block or not. The code associated with the production of attestations is described in Algorithm 3 at subsection 4.2. We then describe in Algorithm 6 how the reception of attestations is handled.

4.1.4 Finality Gadget. The finality gadget is the mechanism that aims at finalizing blocks. The finality gadget grows the finalized chain disregarding the block production. This decoupling of the finality mechanism and the block production permits block availability even when the finalizing process is slowed down. This differs from protocols like Tendermint [6], where a new block can be added to the chain only after being finalized.

The finality gadget works at the level of epochs. Instead of finalizing blocks one by one, the protocol uses checkpoint votes to finalize entire epochs. We now present in more detail how the finality gadget of Ethereum PoS grows the finalized chain.

Recall that to be taken into account, a checkpoint vote needs to be included in a block. The vote will then influence the behavior of validators regarding this particular branch. Thus, in Algorithm 9 of subsection 4.2 the function `countMatchingCheckpointVote` only counts the matching checkpoint votes of attestations included in a block.

[6] In the event of an epoch without a block for the first slot, the block used for the checkpoint is the last block in the candidate chain, belonging then to a previous epoch. On the contrary, if the proposer of the first slot proposes multiple blocks, this will make multiple checkpoints for the other validators to choose from using the fork choice rule.
[7] The current protocol specifications [11] indicate that correct validators should send their block proposition during the first third of their designated slot.
[8] Detailed explanation in subsubsection 4.1.6.
[9] Also called GHOST vote in [7] and in the specifications [11].
[10] Also called FFG vote in [7] and in the specifications [11].

Justification. The justification process is a step to achieve finalization[11]. It operates on checkpoints at the level of epochs. Justification occurs thanks to checkpoint votes. The checkpoint vote contains a pair of checkpoints: the checkpoint *source* and the checkpoint *target*. We can count with `countMatchingCheckpointVote` the sum of balances of the validator's checkpoint votes with the same source and target. If validators controlling more than two-thirds of the stake make the same checkpoint vote, then we say there is a *supermajority link* from the checkpoint source to the checkpoint target. The checkpoint target of a supermajority link is said to be *justified*.

More formally, a checkpoint vote is in the form of a pair of checkpoints: $\big((a, e_a), (b, e_b)\big)$, also noted $(a, e_a) \rightarrow (b, e_b)$. For the checkpoint vote $(a, e_a) \rightarrow (b, e_b)$ we call (a, e_a) the checkpoint source and (b, e_b) the checkpoint target. The checkpoint source is necessarily of an earlier epoch than the checkpoint target, i.e., $e_a < e_b$. In line with [7], we say there is a *supermajority link* from checkpoint (a, e_a) to checkpoint (b, e_b) if validators controlling more than two-thirds of the stake cast an attestation with checkpoint vote $(a, e_a) \rightarrow (b, e_b)$. In this case, we write $(a, e_a) \xrightarrow{\text{J}} (b, e_b)$ and the checkpoint (b, e_b) is justified.

Finalization. The finalization process aims at finalizing checkpoints, thus growing the finalized chain. Checkpoints need to be justified before being finalized. Let us illustrate the finalization process with the two scenarios that can lead to finalization. The first case presents the main scenario in the synchronous setting. It shows how a checkpoint can be finalized in two epochs, the least amount of epochs needed for finalization.
Case 1: The scenario is depicted in Figure 5.

(1) Let $A = (a, e)$ and $B = (b, e + 1)$ be checkpoints of two consecutive epochs such that $A = (a, e)$ is justified.
(2) A supermajority link occurs between checkpoints A and B where A is the source and B the target. This justifies checkpoint B. Hence, we can write: $(a, e) \xrightarrow{\text{J}} (b, e + 1)$ or equivalently $A \xrightarrow{\text{J}} B$.
(3) This leads to A being finalized.

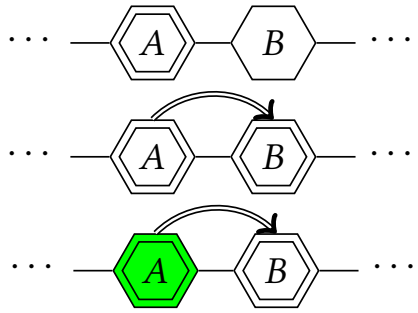

Fig. 1. The figure depicts the finalization scenario of **Case 1** with the 3 steps from top to bottom. We represent a checkpoint with a hexagon, a justified checkpoint with a double hexagon, and a finalized checkpoint with double hexagon coloured. The arrow between two checkpoints indicates a supermajority link.

The second case illustrates the scenario in which two consecutive checkpoints are justified but not finalized. This means that the current highest justified checkpoint (e.g., B in Figure 2) was not justified with a supermajority link having the previous checkpoint A as its source. Then occurs a new justification with the source and target being at the maximum distance (2 epochs) for the source to become finalized. An important note is that there is no limit on the distance between two checkpoints for justification to be possible. This limit only exists for finalization.

[11]The genesis checkpoint (i.e., the checkpoint of the first epoch) is the exception to this rule: it is justified and finalized by definition.

Case 2: The scenario is depicted in Figure 2.

(1) Let $A = (a, e)$, $B = (b, e + 1)$ and $C = (c, e + 2)$ be checkpoints of consecutive epochs such that A and B are justified. There is no supermajority link between A and B, A cannot be finalized as in Case 1 above.
(2) Now, a supermajority link occurs between checkpoints A and C where A is the source and C the target. This justifies checkpoint C, i.e., $A \xrightarrow{\text{J}} C$.
(3) This leads to A being finalized.

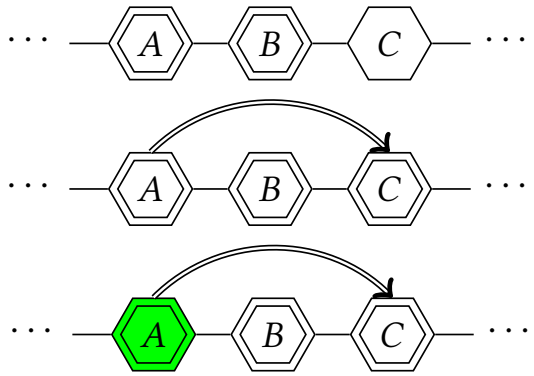

Fig. 2. The figure depicts the finalization scenario of **case 2** with the 3 steps from top to bottom. between two checkpoints indicates a supermajority link.

These two cases illustrate the fact that for a checkpoint to become finalized, it needs to be the source of a supermajority link between justified checkpoints. Once a checkpoint is finalized, all the blocks leading to it (including the block in the pair constituting the checkpoint) become finalized. We now describe the conditions for a checkpoint to be finalized more formally. Let (a, e_a) and (b, e_b) be two checkpoints such that $e_a < e_b$. The checkpoint (a, e_a) is finalized if the following conditions are respected:

- Source justified: checkpoint (a, e_a) is justified.
- Supermajority link: there exists a supermajority link $(a, e_a) \xrightarrow{\text{J}} (b, e_b)$.
- Maximal gap: $e_b - e_a \leq 2$.[12] Moreover, if $e_b - e_a = 2$ then the checkpoint in between at epoch $e_a + 1 (= e_b - 1)$ must be justified.

The importance of the last condition is illustrated by the Figure 3. In practice, these three conditions are only applied on the last four epochs. As mentioned in [7], at the implementation level, checkpoints more than 4 epochs old are not considered for finalization. All the conditions for finalization are illustrated by the last 4 conditions of Algorithm 9 in subsection 4.2.

4.1.5 Fork choice rule & Block proposition. The fork choice rule is the mechanism that allows each validator to determine the candidate chain depending on their view of the BlockTree and the state of checkpoints. Ethereum PoS fork choice rule is LMD GHOST. The LMD GHOST fork choice rule stems from the Greedy Heaviest-Observed Sub-Tree (GHOST) rule [27] which considers only each participant's most recent vote (Latest Message Driven). During an epoch, each validator must make one *block vote* on the block considered as the head of the candidate chain according to its view. To determine the head of the candidate chain, the fork choice rule does the following:

[12]This last condition necessitating the two checkpoints to be at most 2 epochs away from each other is also called *2-finality* [7].

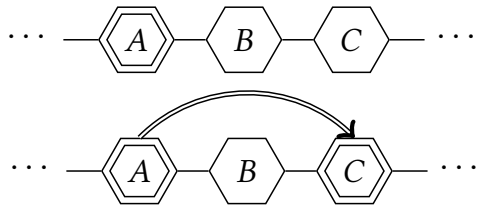

Fig. 3. This figure illustrates the case of two checkpoints A and C respecting all the conditions for finalization but the one that stipulates that a checkpoint B in-between must be justified for A to be finalized.

(1) Go through the list of validators and check the last block vote of each.
(2) For each block vote, add a weight to each block of the chain that has the block voted as a descendent. The weight added is proportional to the stake of the corresponding validator.
(3) Start from the block of the justified checkpoint with the highest epoch and continue the chain by following the block with the highest weight at each connection. Return the block without any child block. This block is the head of the candidate chain.

The actual implementation is presented in Algorithm 7 in subsection 4.2. This algorithm is similar to the one already presented in [7]. Albeit each *block vote* being for a specific block, the fork choice rule considers all the chains leading to that block. This reflects the fact that a vote for a block is a vote for the chain leading to that block. Figure 4 offers an explanation with a visualization of how attestations influence the fork choice rule. At each chain intersection, the fork choice rule favors the chain with the most attestations.

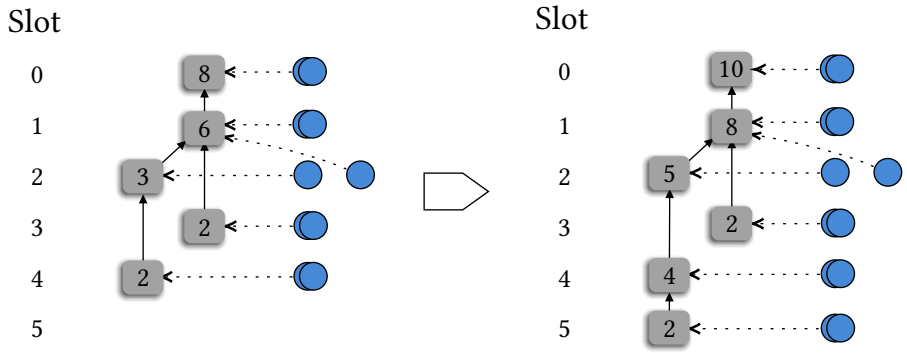

Fig. 4. Fork choice rule example observed from validator i's point of view. We represent block votes with blue circles. Block votes point to specific blocks indicating the block considered as the *headblock* of the candidate chain at the moment of the vote. Each block has a number representing the value attributed by the fork choice rule algorithm (cf. Algorithm 7) to determine the candidate chain - we assume for this example that each validator has the same stake of 1. On the left we represent the chain at the end of slot 4, and on the right at the end of slot 5. On the left, i's fork choice rule gives the block of slot 4 as C_i's head. On the right, the fork choice rule designates the block of slot 5 as the head of the candidate chain.

4.1.6 Pseudo-Randomness. The solution used by Ethereum to incorporate randomness in the consensus is called *RANDAO*. RANDAO is a mechanism that creates pseudo-random numbers in a decentralized fashion. It works by aggregating different pseudo-random sources and mixing them.

Seed creation. Each epoch produces a seed. This seed is created with the help of the block proposers of the said epoch. Each valid block contains a field called `randao_reveal`[13]. The seed is the hash of a `XOR` of all the `randao_reveal` of an epoch plus an epoch number.

To prevent manipulation in the seed creation, each block's `randao_reveal` must be the signature of specific data. The data to sign is the current epoch number. Anyone can then check that this signature is from the block proposer and for the correct data.

Seed utilization. The algorithm using the seed is called `compute_shuffled_index` (cf. Algorithm 12). This algorithm stems from the algorithm *swap-or-not* that was introduced by [16]. `compute_shuffled_index` shuffles the validators list and gives new roles depending on their shuffled index. This pseudo-random shuffling function is used two times in Ethereum PoS consensus algorithm: for the proposer selection and for the committee selection. The proposer selection is described in Algorithm 13 and the committee selection in Algorithm 14.

4.2 Pseudo Code

In this section, we dive into a practical understanding of the mechanism behind the Ethereum PoS protocol. According to the specifications [11] and various implementations (such as Prysm [24] and Teku [8]) we formalize the main functions of the protocol through pseudo-codes for a better understanding and for analysis purposes.

Each validator p runs an instance of this particular pseudo code. For instance, when a validator p proposes a block, he broadcasts the following message: $\langle PROPOSE, (slot, \texttt{hash}(headBlock_p), content)\rangle$, where $slot$ is the slot at which the proposer proposes the block, the hash of the $headBlock_p$ is the hash of the block considered to be the head of the candidate chain according to the fork choice rule (see Algorithm 7), and *content* contains data used for pseudo-randomness, among other things that we will not detail here. We instead focus on the consensus protocol.

We describe in the following paragraphs the variables and functions used in the pseudo-code and the goal of these functions.

Variables. During the computation, each variable takes a value that is subjective and may depend on the validator. We indicate with p the fact that the value of variables depends on each process. The variable $tree_p$ is considered to be a graph of blocks with each block linked to its predecessor and thus represents the view of the blockchain (more precisely, $tree_p$ represents the view of all blocks received by the validator since the genesis of the system). Each $tree_p$ starts with the genesis block. $role_p$ corresponds to the different roles a validator can have, which is possibly none (i.e., for each slot, the validator can be proposer, attester, or have no role). $role_p$ is a list containing the role(s) of the validator for the current slot. The $slot_p$ is a measure of time. In particular, a slot corresponds to 12 seconds. $slot_p \in \mathbb{N}$. Slot 0 begins at the time of the genesis block and is incremented every 12 seconds. $headblock_p$ is the head of the candidate chain according to p's local view and the fork choice rule. A checkpoint C is a pair block-epoch that is used for finalization. C has two attributes which are $justified$ and $finalized$ which can be true or false (e.g., if C is only justified $C.justified = \texttt{true}$ and $C.finalized = \texttt{false}$). $lastJustifiedcheckpoint_p$ is the *justified* checkpoint with the highest epoch. $currentCheckpoint_p$ is the checkpoint of the current epoch. The list $attestation_p$ is a list of size n (i.e., the total number of validators). This list is updated only to contain the latest messages of validators (of at most one epoch old). $CheckpointVote_p$ is a pair of checkpoints, so a pair of pairs, used to make a checkpoint vote. Let us stress

[13] See subsection 4.2 for the detailed explanation of its use.

the fact that all those variables are local, and at any time, two different validators may have different valuations of those variables.

Functions. We describe the main functions of the protocol succinctly before giving the pseudo code associated and a more detailed explanation:

- `validatorMain` is the primary function of the validator, which launches the execution of all subsidiary functions.
- `sync` is a function that runs in parallel of the `validatorMain` function and ensures the synchronization of the validator. It updates the slot, the role(s) and processes justification and finalization at the end of the epoch and when a new validator joins the system.
- `getHeadBlock` applies the fork choice rule. This is the function that indicates the head block of the candidate chain.
- `justificationFinalization` is the function that handles the justification and finalization of checkpoints.

We depict in Algorithm 1 the main procedure of the validator. This procedure initializes all the values necessary to run a validator. In this paper, we consider the selection of validators already made to focus on the description of the consensus algorithm itself. The main starts a routine called `sync` to run in parallel. Then there is an infinite loop that handles the call to an appropriate function when a validator needs to take action for its role(s).

Algorithm 1 Main code for a validator p

```
1:  procedure validatorMain( )
2:      tree_p ← nil                                   ▷ The tree represents the linked received blocks
3:      role_p ← [ ]          ▷ role_p can be ROLE_PROPOSER and/or ROLE_ATTESTER when it is not empty
4:      slot_p ← 0                                                              ▷ slot_p ∈ ℕ
5:      lastJustifiedCheckpoint_p ← (0, genesisBlock)   ▷ A checkpoint is a tuple (epoch, block)
6:      attestation_p ← [ ]          ▷ List of latest attestations received for each validator
7:      validatorIndex_p ← index of the validator                ▷ Each validator has a unique index
8:      listValidator ← [p_0, p_1, ..., p_{N−1}]                 ▷ A list of the validators index
9:      balances ← [ ]                  ▷ A list of the balances of the validators, their stake
10:
11:     start sync(tree_p, slot_p, attestation_p, lastJustifiedCheckpoint_p, role_p, balances)
12:
13:     while true do
14:         if role_p ≠ ∅ then
15:             if ROLE_PROPOSER ∈ role_p then
16:                 prepareBlock()
17:             if ROLE_ATTESTER ∈ role_p then
18:                 prepareAttestation()
19:             role_p ← [ ]
20:         else
21:             no role assigned                                    ▷ No action required
```

The roles performed by the validator when proposer or attester are defined in Algorithm 2 and Algorithm 3, respectively. The proposer of a block does the three following tasks:

(1) Get the head of its candidate chain to have a block to build upon;
(2) Sign a predefined pair to participate in the process of pseudo-randomness;
(3) Broadcast a new block built on top of the head of the candidate chain.

The attestation is composed of three parts: the slot, the block vote, and the checkpoint vote. The validator uses the fork choice rule presented in Algorithm 7 to obtain the block chosen for the block vote. Algorithm 7 and the one stemming from it, Algorithm 8, have already been defined in [7]. We restated them here for the sake of completeness. For the checkpoint vote, an honest validator should always vote for the current epoch as the target and take the justified checkpoint with the highest epoch (i.e., *lastJustifiedCheckpoint*) as the source. In order to broadcast this attestation, the attester must wait for one of two things, either a block has been proposed for this slot or 1/3 of the slot (i.e., 4 seconds) has elapsed. This is ensure by the function `waitForBlockOrOneThird`.

Algorithm 2 broadcast block

1: **procedure** PREPAREBLOCK( )
2: $headBlock_p \leftarrow$ `getHeadBlock( )`
3: $randaoReveal \leftarrow$ `sign(` `epochOf(`$slot$`))`
4: **broadcast** $\langle PROPOSE, (slot, \texttt{hash}(headBlock_p), randaoReveal_p, content)\rangle$

Algorithm 3 Broadcast Attestation

1: **procedure** PREPAREATTESTATION( )
2: `waitForBlockOrOneThird()` ▷ wait for a new block in this slot or $\frac{1}{3}$ of the slot
3: $headBlock_p \leftarrow$ `getHeadBlock()`
4: $currentCheckpoint_p \leftarrow$ (first block of the epoch, `epochOf(`$slot$`)`)
5: $CheckpointVote_p \leftarrow (lastJustifiedCheckpoint_p, currentCheckpoint_p)$
6: **broadcast** $\langle ATTEST, (slot_p, \underbrace{\texttt{hash}(headBlock_p)}_{\text{block vote}}, \underbrace{CheckpointVote_p}_{\text{checkpoint vote}})\rangle$

The synchronization of the validator p is handled by the function `sync` described in Algorithm 4. This algorithm allows the validator to update its view of the blockchain, in particular, the current slot, the list of attestations, the last justified checkpoint, the validator's role, and the balances of all validators. To determine its role(s), the validator verify the index of the designated validator for the current slot and the set of indexes forming the committee of the current slot. In more details, there are two conditions to give a role to a validator for the current slot. The first condition calls Algorithm 13 and gives the validator p the role of proposer if its index is the one of the current proposer. The second condtion checks whether p belongs to the committee of the current slot (see Algorithm 14). The two roles of proposer and attester are entirely distinct, i.e., the proposer of a slot is not necessarily an attester of that slot.

The synchronization function also starts two other routines which are `syncBlock` and `syncAttestation` corresponding to Algorithm 5 and Algorithm 6, respectively. These routines are used to handle the broadcast of proposers and attesters. In both functions, upon receiving a block or an attestation, the validator p verifies that it is valid thanks to the `isValid` function. It is important to note that upon receiving a block, a validator can update the last justified checkpoint only if the current epoch has not started more than 8 slots ago. This particular condition is what the patch has brought to prevent a liveness attack (see section 5).

Algorithm 9 can be considered the most intricate one. This algorithm is responsible for the justification or finalization of the checkpoints at the end of an each epoch. To do so, it counts the number of checkpoint votes with the same source and target. If this number corresponds to more than 2/3 of the stake of all validators, then the target is considered justified for the validator running this algorithm. The last four conditions concern finalization. They verify among

Algorithm 4 Sync

```
1:  procedure SYNC(tree, slot, attestation, role, lastJustifiedCheckpoint,)
2:      start syncBlock(slot, tree)
3:      start syncAttestation(attestation)
4:      repeat
5:          previousSlot ← slot
6:          slot ← ⌊ time in seconds since genesis block / 12 ⌋
7:          if previousSlot ≠ slot then                          ▷ If we start a new slot
8:              roleSlotDone ← false
9:              if validatorIndex_p = getProposerIndex(getSeed(current epoch), slot) then
10:                 append ROLE_PROPOSER to role_p
11:             if validatorIndex_p ∈ computeCommittee(getSeed(current epoch), slot) then
12:                 append ROLE_ATTESTER to role_p
13:         if slot (mod 32) = 0 then                            ▷ First slot of an epoch
14:             jutificationFinalization(tree, lastJustifiedCheckpoint)
15:     until validator exit
```

Algorithm 5 Sync Block

```
1:  procedure SYNCBLOCK(slot, tree)
2:      upon ⟨PROPOSE, (slot_i, hash(headBlock_i), randaoReveal_i, content_i)⟩ from validator i do
3:          block ← ⟨PROPOSE, (slot_i, hash(headBlock_i), randaoReveal_i, content_i)⟩
4:          if isValid(block) then
5:              add block to tree
6:              if slot (mod 32) ≤ 8 then
7:                  update justified checkpoint if necessary
```

Algorithm 6 Sync Attestation

```
1:  procedure SYNCATTESTATION(attestation)
2:      upon ⟨ATTEST, (slot_i, headBlock_i, checkpointEdge_i)⟩ from validator i do
3:          attestation_i ← ⟨ATTEST, (slot_i, headBlock_i, checkpointEdge_i)⟩
4:          if isValid(attestation_i) then
5:              attestation[i] ← attestation_i
```

Algorithm 7 Get Head Block

```
1:  procedure GETHEADBLOCK( )
2:      block ← block of the justified checkpoint with the highest epoch
3:      while block has at least one child do
4:          block ← arg max_{b' child of block} weight(tree, Attestation, b')
5:          (ties are broken by hash of the block header)
6:      return block
```

the last four checkpoints which one fulfills the conditions to become finalized. The conditions to become finalized are formally described in subsection 4.1 and can be summarized by: the checkpoint must be the source of a supermajority link, and all the checkpoints between the source and target included must be justified.

Algorithm 8 Weight

```
1: procedure WEIGHT(tree, Attestation, block)
2:     w ← 0
3:     for every validator v_i do
4:         if ∃a ∈ Attestation an attestation of v_i for block or a descendant of block then
5:             w ← w+ stake of v_i
6:     return w
```

Algorithm 9 Justification and Finalization

```
1: procedure JUTIFICATIONFINALIZATION(tree, lastJustifiedCheckpoint)
2:     source ← lastJustifiedCheckpoint
3:     target ← the current checkpoint
4:     nbCheckpointVote ← countMatchingCheckpointVote(source, target)
5:     ▷ justification process:
6:     if nbCheckpointVote ≥ 2/3 * total balance of validators then
7:         target.justified ← true
8:         lastJustifiedCheckpoint ← target
9:     ▷ finalization process:
10:    A, B, C, D ← the last 4 checkpoints          ▷ With D being the current checkpoint.
11:    if A.justified ∧ B.justified ∧ (A -J-> C) then
12:        A.finalized ← true                       ▷ Finalization of A
13:    if B.justified ∧ (B -J-> C) then
14:        B.finalized ← true                       ▷ Finalization of B
15:    if B.justified ∧ C.justified ∧ (B -J-> D) then
16:        B.finalized ← true                       ▷ Finalization of B
17:    if C.justified ∧ (C -J-> D) then
18:        C.finalized ← true                       ▷ Finalization of C
```

Algorithm 10 Get randao mix

```
1: procedure GETRANDAOMIX(epoch)
2:     mix ← 0
3:     headBlock ← getHeadBlock()
4:     for each block parent of headBlock and belonging to epoch do
5:         mix ← mix ⊕ hash(block.randaoReveal)     ▷ ⊕ is a bit-wise XOR operator
6:     return mix
```

Algorithm 11 Get seed

```
1: procedure GETSEED(epoch)
2:     mix ← getRandaoMix(epoch − 2)    ▷ The seed of an epoch i is based on the randao mix of epoch i − 2
3:     return hash(epoch + mix)
```

The pseudo-randomness needs a different seed for each epoch to yield different results. This is ensured by hashing the RANDAO mix and the epoch number as shown in Algorithm 11. Adding the epoch number is helpful if no block is

proposed during an entire epoch. This corner case would always result in the same seed if it were not for the epoch number.

The RANDAO mix is computed in Algorithm 10. The computation of the RANDAO mix of a given epoch consists of *XORing* all the *randaoReveal* of the blocks in that particular epoch. We consider only the blocks of that particular epoch that belong to the candidate chain.

The RANDAO mix of epoch $e-2$ determines the role of validators in epoch e. Hence, with Algorithm 14, as soon as epoch $e-2$ is over, validators can know to which committee they belong to at epoch e. `computeCommittee` (Algorithm 14) is the function that, given a seed and an epoch, returns the list of validators index corresponding to the committee for the slot specified. The number of validators in each committee[14] is computed to be less than $N/32$ (with N the number of validators). Then using the shuffled index computed with Algorithm 12 a committee of the given size is drawn according to the slot in argument. All validators of the committee will have to perform the role of attester during this slot.

Since the balance can change until the previous epoch, block proposers are known at the end of epoch $e-1$ for epoch e. Algorithm 13 is the one handling the selection of a proposer for a designated slot. It starts by creating a seed specifically for the slot in question. Then there is a loop starting with a pseudo-random selection of the validator's index. The loop stops only when a validator meets the condition criteria. This condition is equivalent to being selected with a probability depending on the balance. Thus, the validator with index *proposerIndex* is selected with probability $\frac{effectiveBalance}{32}$, with *effectiveBalance* being the stake of *proposerIndex* capped to 32, i.e., $min(balance, 32)$.

Both algorithms 13 and 14 make the proposer and the committee selection resort to Algorithm 12 to imbue randomness. As mentioned in section 4, Algorithm 12 stems from the algorithm *swap-or-not* [16]. Its name helps us understand the principle behind the algorithm: select a validator and its opposite (based on a pivot) and swap them or not. The selection of the validator and the swap depend on the value of a hash. An essential aspect of this algorithm is that it can get the index of validators in the shuffled list without having to compute the shuffling of the whole list of validators. This reduces unnecessary computation.

Algorithm 12 Compute shufflex index

```
1: procedure computeShuffledIndex(index, seed, nbValidators)
2:     for i = 0 to 90 do
3:         pivot ← hash(seed + i) (mod nbValidators)
4:         flip ← pivot + nbValidators − index (mod nbValidators)
5:         position ← max(index, flip)
6:         bit ← hash(seed + i + position) (mod 2)
7:         if bit = 0 (mod nbValidators) then
8:             index ← flip
9:     return index
```

5 LIVENESS ATTACK

In this section, we describe a liveness attack called *bouncing attack* that delays finality in a partially synchronous network after GST. Previous works also exhibit liveness attacks against the protocol using the intertwining of the fork choice rule and the finality gadget [19, 21]. To prevent this type of attack, the protocol now contains a "patch" [25]

[14] In the actual implementation, committees have a maximum size of 2048 [11].

Algorithm 13 Get proposer index

```
1:  procedure getProposerIndex(seed, slot)
2:      MAX_RANDOM_BYTE ← 2^8 − 1
3:      i ← 0
4:      proposerSeed ← hash(seed+slot)
5:      nbValidators ← length(listValidator)
6:      while true do
7:          proposerIndex ← listValidator[computeShuffledIndex(i, seed, nbValidators)]
8:          randomByte ← first byte of hash(proposerSeed + i (mod nbValidators))
9:          effectiveBalance ← listValidators[proposerIndex].effectiveBalance
10:         if effectiveBalance * MAX_RANDOM_BYTE ≥ MAX_EFFECTIVE_BALANCE *randomByte then
11:             return proposerIndex
12:         i ← i + 1
```

Algorithm 14 Compute Committee

```
1:  procedure computeCommittee(seed, slot)
2:      committee ← [ ]
3:      nbValidatorByCommittee ← ⌈lenght(listValidator)/32⌉
4:      for i = (slot (mod 32))*nbValidatorByCommittee to (slot + 1 (mod 32))*nbValidatorByCommittee −1 do
5:          committee.append(listValidator[computeShuffledIndex(i, seed, nbValidators)])
6:      return committee
```

that was suggested on the Ethereum research forum [20]. We show that the implemented patch is insufficient and this type of attack is still possible if certain conditions are verified. This is a probabilistic liveness attack against the protocol of Ethereum Proof-of-Stake. Our attack can happen with less than 1/3 of Byzantine validators, as discussed in subsection 5.3.

5.1 Bouncing Attack

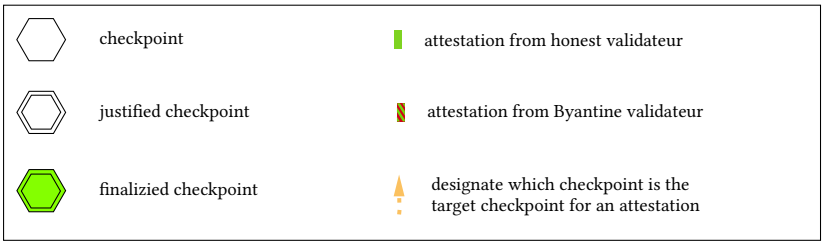

Fig. 5. This figure serves as a summary of the signification of the main diagrams of other figures.

The *Bouncing Attack* [19] describes a liveness attack where the suffix of the chain changes repetitively between two candidate chains, thus preventing the chain from finalizing any checkpoint. The Bouncing Attack exploits the fact that the candidate chains should start from the justified checkpoint with the highest epoch. It is possible for Byzantine validators to divide honest validators' opinions by justifying a new checkpoint once some honest validators have already cast their vote (made an attestation) during the asynchronous period before GST.

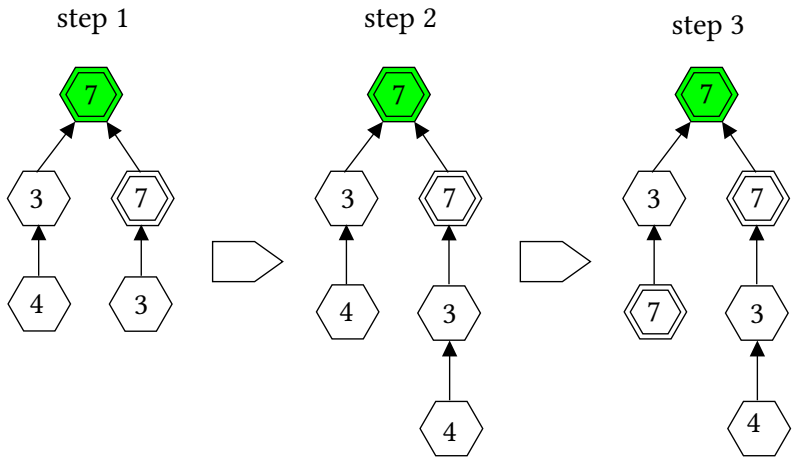

Fig. 6. A bouncing attack presented in 3 steps. We have 10 validators, of which 3 are Byzantines. The number inside each hexagon corresponds to the number of validators who made a checkpoint vote with this checkpoint as target. **1st step:** We start in a situation where there is a fork. A checkpoint is justified on one of the chains and a checkpoint of a higher epoch is *justifiable* on the other. checkpoints are justified. We are at the end of the third epoch in which honest validators have divided their vote on each side. **2nd step:** We have reached GST at the beginning of the fourth epoch and 4 honest validators have already voted (rightfully so). **3rd step:** Here is the moment Byzantine validators take action and release their checkpoint vote for the concurrent chain, thus justifying the previously forsaken checkpoint and thereby changing the highest justifying checkpoint. By repeating this process, the bouncing attack can continue indefinitely.

The bouncing attack becomes possible once there is a *justifiable* checkpoint in a different branch from the one designated by the fork choice rule with a higher epoch than the current highest justified checkpoint. A *justifiable checkpoint* is a checkpoint that can become justified only by adding the checkpoint votes of Byzantine validators. If this setup occurs, the Byzantine validators could make honest validators start voting for a different checkpoint on a different chain, leaving a justifiable checkpoint again for them to repeat their attack and thus making validators *bounce* between two different chains and not finalizing any checkpoint. Hence the name Bouncing attack.

Let us illustrate the attack with a concrete case. We show in Figure 6 an oversimplified case with only 10 validators, among which 3 are Byzantines. To occur, the attack needs to have a justifiable checkpoint with a higher epoch than the last justified checkpoint. We reach this situation before GST, which is presented in the left part of the figure. After reaching GST, Byzantine validators wait for honest validators to make a new checkpoint justifiable. When a new checkpoint is justifiable, the Byzantine validators cast their vote to justify another checkpoint, as shown by the right part of the figure. This will lead honest validators to start voting for the left branch, thus reaching a situation similar to the first step allowing the bouncing attack to continue. The repetition of this behavior is the bouncing attack. We emphasize this example in more detail in Figure 7 by detailing the sequence of votes allowing a "bounce" to occur and leaving a justifiable checkpoint on the other branch.

5.2 Implemented patch

The explication of the patch is described for the first time on the Ethereum research forum [20]. The solution that was found to mitigate the bouncing attack is to engrave in the protocol the fact that validators cannot change their minds regarding justified checkpoints after a part of the epoch has passed.

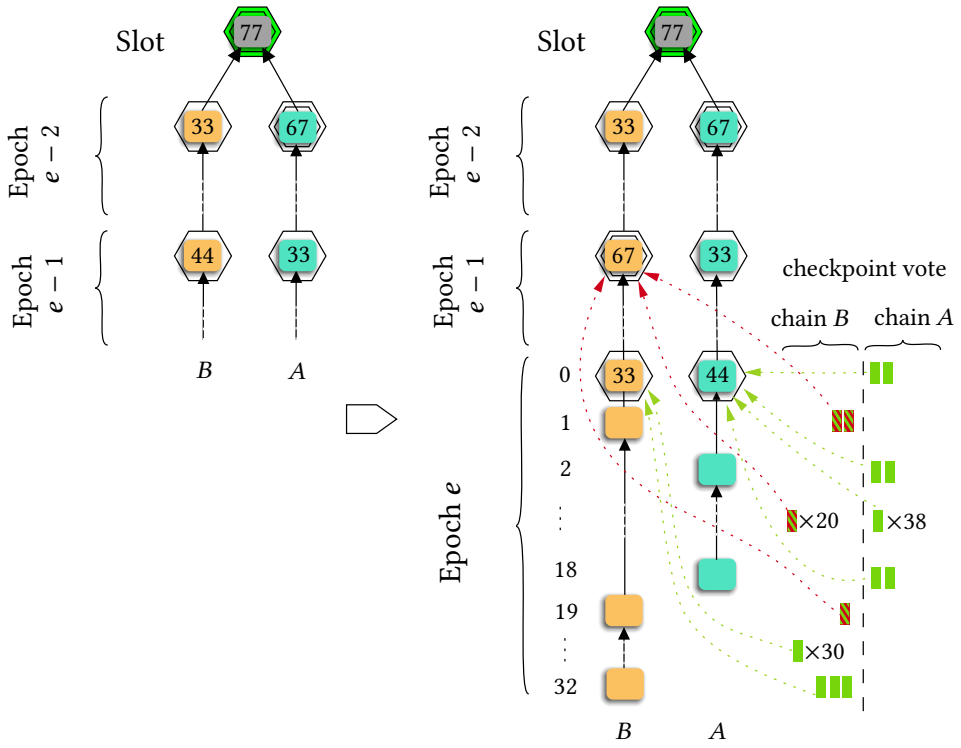

Fig. 7. This figure presents a detailed version of the bouncing attack. In this example, we have a total of 100 validators, of which 23 are Byzantines. A block in a checkpoint corresponds to the block associated with that checkpoint. The number inside each hexagon (hovering a block) corresponds to the number of validators who made a checkpoint vote with this checkpoint as target. We distinguish between two sorts of checkpoint votes, the Byzantine ones, which are bi-color rectangles, and the honest ones, which are uni-color rectangles. We compile the 3 steps of Figure 6 in 2 with more information on how justification's turning point is accomplished because of the Byzantine agents. **First step:** We begin from a situation where epoch $e-1$ just ended and we now reach GST. Notice that the candidate chain is chain A because the checkpoint with the highest epoch is on chain A but not chain B. **Second step:** In this step, the checkpoint vote released during epoch e can change the last justified checkpoint to change the candidate chain for chain A to chain B. Byzantine validators released their checkpoint vote from the previous epoch during epoch e. They send their last checkpoint vote at slot 23 once the checkpoint of epoch e on chain A has reached 44, thus becoming justifiable (i.e., not yet justified but with enough votes so that Byzantine validators can justify it). This trigger the candidate chain to change from chain A to chain B starting the *bounce*.

The goal of the solution proposed is to prevent the possibility of justifiable checkpoints being left out by honest validators. To prevent honest validators from leaving a justifiable checkpoint, the patch must stop validators from changing their view of checkpoints before more than 1/3 of validators have cast their checkpoint vote. This condition stems from the fact that we reckon the proportion of Byzantine validators to be at most $1/3 - \epsilon$. To apply this condition, the patch designates a number of slots after which honest validators cannot change their view of checkpoints. Indeed since validators are scattered equally among the different slots to cast their vote (in attestations) in a specific time frame, stopping validators from changing their view after a number of slots is equivalent to stopping changing their view after a certain proportion of validators have voted. This does look like a solution to prevent Byzantine validators from influencing honest validators into forsaking a checkpoint now *justifiable* for them.

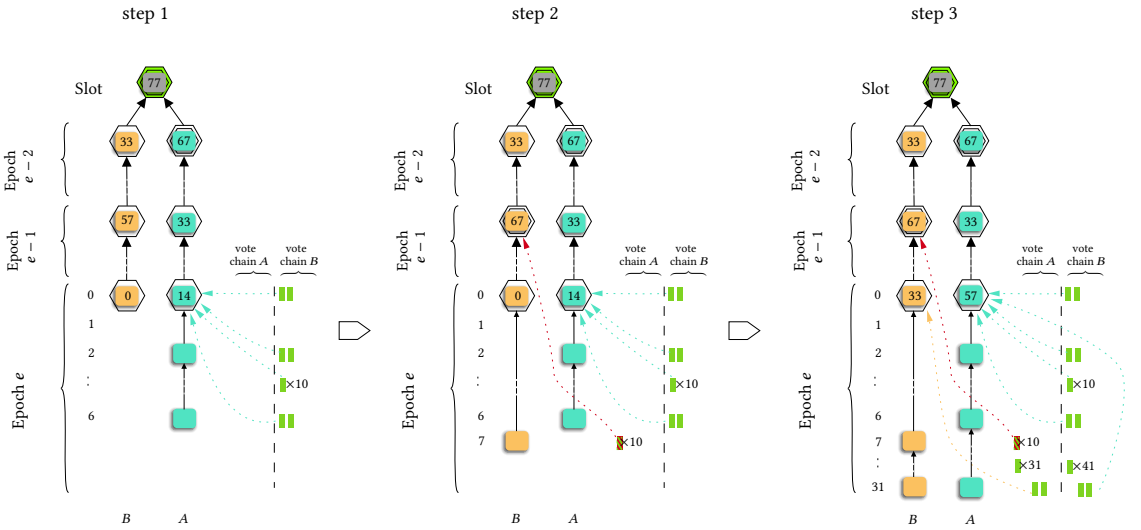

Fig. 8. This figure presents the probabilistic bouncing attack. In this example, we consider 100 validators, of which 10 are Byzantines. A block in a checkpoint corresponds to the block associated with that checkpoint. The number inside each hexagon (hovering a block) corresponds to the number of validators who made a checkpoint vote with this checkpoint as target. The example starts at the slot before the attack in step 1. GST has been reach in epoch e and honest validators have started to vote on chain A. This is the correct action because the justified checkpoint with the highest epoch is on chain A (at epoch $e-2$). During the next slot in step 2, before reaching $limitSlot$, a Byzantine validator sends a block with withheld votes for the checkpoint at epoch $e-1$ on chain B. It is released just in time for a set of honest validators to consider it and too late for the remaining validators. The honest validators that see the block in time will update their view of the justified checkpoint with the highest epoch and consider chain B as the candidate chain. We now show how the epoch continues with step 3. The block produced by a Byzantine having been released just in time, $(1/3)$ of honest validators have changed their view. This results in a situation where Byzantine validators can perform the same attack during the next epoch provided that at least one Byzantine validator is selected to be block proposer on chain A for one of the 8^{th} first slots.

To enforce this behavior, called the "fixation of view", the protocol has a constant j called `SAFE_SLOTS_TO_UPDATE_JUSTIFIED` in the code (cf. Algorithm 5 in subsection 4.2). This constant is the number of slots[15] until when validators can change their view of the justified checkpoints. In the patch introducing this constant j, they mention a possible attack called *splitting attack*. As they point out, the splitting attack relies on a "last minute delivery" family of strategies whereby releasing a message late enough, some validators will consider it too late while others will not. This could thus split the validators into two different chains, not being able to conciliate their view before the end of the epoch. They consider the assumption of attackers being able to send a message at the right time to split honest validators too strong, we discuss in section 7 whether it is the case or not. In 5.3, we present a new attack inspired by the splitting attack with more realistic assumptions.

5.3 Probabilistic Bouncing attack - why the patch is not enough

In this part, we present our novel attack against the protocol of Ethereum Proof-of-Stake. The attack is visually explained in Figure 8.

[15] At the time of writing this paper, $j = 8$ [11].

5.3.1 Attack Conditions. Our attack takes place during the synchronous period and uses the power of *equivocation* of Byzantine processes. Equivocation is caused by a Byzantine process that sends a message only to a subset of validators at a given point in time and potentially another message or none to another subset of validators. The effect is that only a part of the validators will receive the message on time. More in detail, the bounded network delay is used by a Byzantine validator to convey a message to be read on a specific slot by some validators and read on the next slot by the other validators. Note that if a protocol is not tolerant to equivocation, then it is not BFT (Byzantine-Fault Tolerant), since equivocation is the typical action possible for Byzantine validators.

5.3.2 Attack Description and Analysis. Let $\beta \leq f/n$ be the fraction of Byzantine validators in the system. The attack setup is the following. First, as in the traditional bouncing attack, we start in a situation where the network is still partially synchronous. A fork occurs and results in the highest justified checkpoint being on chain A at epoch e, and a justifiable checkpoint at epoch $e+1$ on chain B. Assume now that GST is reached, the attack can proceed[16] as follows:

(1) Since GST is reached, the network is fully synchronous. Chain A is the candidate chain for all validators.
(2) Just before validators must stop updating their view concerning justified checkpoint (i.e., before reaching the limit of j slots[17] in the epoch corresponding to the condition line 6 in Algorithm 5), a Byzantine proposer proposes a block (cf. Algorithm 2) on chain B. This block contains attestations with enough checkpoint votes to justify the justifiable checkpoint that was left by honest validators. The attestations included in the block are attestations of Byzantine validators that were not issued in the previous epoch when they were supposed to. The block must be released just in time, that is, right before the end of slot j, so that $(1/3 - \beta)$ of the validators change their view of the candidate chain to be active on chain B while the rest of honest validators continue on chain A. This is possible due to the patch preventing validators from changing their mind after j slots.
(3) Repeat the process.

An important aspect to consider in the attack is the probability for Byzantine validators to become proposers. This is an important part since without the role of proposer, validators are not legitimate to propose blocks and thus cannot add new attestations containing checkpoint votes on top of the concurrent chain.[18]. The probability of being selected to be a proposer directly impacts how long the probabilistic bouncing attack can continue. In the following theorem, we establish the probability of a probabilistic bouncing attack lasting for a specific number of epochs.

Theorem 5.1. *The probabilistic bouncing attack occurs during k epochs after* GST *and a favorable setting with probability:*

$$P(\textit{bouncing } k \textit{ times}) = (1 - \alpha^j)^k, \tag{1}$$

with $\alpha \in [0, 1]$ the proportion of honest validators and j the number of slots before locking a choice for the justification.

Proof. We denote by α the proportion of honest validators and j the number of slots before locking the choice for justification. We want to know the probability of delaying the finality for k epochs. Once we assume a setup condition sufficient to start a probabilistic bouncing attack, the attack continues until it becomes impossible for Byzantine validators to cast a vote to justify the justifiable checkpoint. To cast their vote, Byzantine validators need one of the j first slots of the concurrent chain to have a Byzantine validator as proposer. Considering the probability of choosing

[16] Note that before GST, no algorithm can ensure liveness since communication delays may not be bounded.
[17] At the time of writing, 8 slots.
[18] Note that Byzantine validators can not use their role of proposer during the previous epoch to release a block with the right attestations because it might not be the last block of the epoch. Indeed because a part of honest validators is on the concurrent chain, they also add blocks to it. For the checkpoint votes contained in the Byzantine attestations to justify the justifiable checkpoint, it must be on the same chain as the attestation making the checkpoint justifiable in the first place.

between each validator, the chance for a Byzantine validator to be a proposer for one of the first j slots is $(1 - \alpha^j)$; α being the proportion of honest validators. For k epochs, we take this result to the power of k. □

As we can see, the probability of the bouncing attack to continue for k epochs depends on two factors: α the proportions of honest validators which cannot be controlled and j the number of slots before which validators are allowed to switch branches. Reducing j to 0 would prevent the bouncing attack from happening (the probability falls to 0), but it would mean that validators are never allowed to change their view of the candidate chain. This naive solution would allow irreconcilable choices between the set of validators and prevent any new checkpoint from being justified, which is a more severe threat to the liveness of Ethereum PoS.

Reducing the number of slots where validators can change their view of the blockchain implies that different views cannot reconcile quickly. Or at least, the window of opportunity for doing so gets smaller. In theory, the proportion of Byzantine validators necessary to perform this attack is $1/n$. This is because we assume a favorable setup and that Byzantine validators can send messages so that only a wished portion of honest validators receives it on time. Our analysis focuses on the course of action of the attackers during the attack rather than the course of action necessary for it to appear.

6 SAFETY

In order to prove the safety of the protocol, we begin by giving lemmas concerning the justification of checkpoints. The first lemma rules out the possibility of two different justified checkpoints having the same epoch. New validators that wants to join the set of validators must send the amount they wish to stake at a specific smart contract[19]. This transaction triggers the process for a validator to join the set of validators. The last step needed for the activation of a validator (allowing it to send attestations and propose blocks) requires that the block adding the validator to the validator set gets finalized[20]. This means that between two finalized checkpoints, the set of validators is fixed.

Lemma 6.1. *If checkpoints C and C' both of epoch e are justified, it must necessarily be that $C = C'$.*

Proof. By hypothesis, we know that Byzantine validators are at most $f < n/3$. For the sake of contradiction, let us assume that C and C' are different checkpoints. Let V be the set of at least $2n/3 - f$ honest validators that cast a checkpoint vote for checkpoint C in epoch e, and V' be the set of at least $2n/3 - f$ honest validators that cast a checkpoint vote for checkpoint C' in epoch e. The intersection of the two sets of honest validators is $|V \cap V'| \geq (2n/3 - f) + (2n/3 - f) - (n - f) = (n/3 - f) > 0$. $|V \cap V'| > 0$ implies that at least one honest validator voted both for checkpoint C and C' in epoch e. This is a contradiction since, according to the protocol specification[21], an honest process signs at most one unique block per epoch, therefore $C = C'$. This proves there cannot be more than one justified checkpoint by epoch. □

The following lemma explains why the finalization of a checkpoint necessarily means that a checkpoint cannot be justified on a different chain afterward.

Lemma 6.2. *If a checkpoint C of epoch e is finalized on chain c, and a checkpoint C' of epoch e' is justified on chain c' with $e' > e$, it necessarily means that c and c' have a common prefix until epoch e.*

[19] Currently 32 ETH is needed to become a validator.

[20] The exact process requires the placement of the validator in the activation queue to be finalized https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md.

[21] This is specified in the specs https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/validator.md#attester-slashing, and implemented on actual client Prysm https://github.com/prysmaticlabs/prysm/blob/0fd52539153e32cfbd0a27ee51f253f8f6bb71c4/validator/client/attest.go#L140. This corresponds to the only attestation done by an honest validator during an epoch, see Algorithm 4.

Proof. C' being justified on chain c', it means that at least $2n/3 - f$ honest validators must have cast a checkpoint vote with C' as checkpoint target for epoch e'.

For the sake of contradiction, let us say that c and c' have a common prefix until epoch $e-1$ at most. For a checkpoint to be justified on chain c' at an epoch strictly superior to e it implies that a set V' of at least $2n/3 - f$ honest validators must have cast a checkpoint vote[22] with a checkpoint target on chain c' and a checkpoint source with epoch inferior to $e-1$.

Checkpoint C of epoch e being finalized on chain c, we have two possibilities[23]. Either the checkpoint at epoch $e+1$ on chain c has been justified with checkpoint C as source. Or the checkpoint at epoch $e+2$ on chain c has been justified with checkpoint C as source and the checkpoint at epoch $e+1$ is justified. Either way, a justification occurred on chain c with checkpoint C as source, and no justification occurred on a different chain before its finalization.

Hence, we know that a set V of at least $2n/3 - f$ honest validators have cast a checkpoint vote with C as checkpoint source before a justification on any other chain.

Seeing that $|V \cap V'| > 0$, at least one honest validator has cast a checkpoint vote with C as checkpoint source and then a checkpoint vote with a checkpoint source of at most epoch $e-1$ and a target on chain c'.

That means that at least one honest validator has cast a checkpoint vote with checkpoint source with epoch inferior to $e-1$ after seeing checkpoint C at epoch e justified. However, the fork choice rule of the protocol (cf. Algorithm 7) requires honest validators to vote on the chain with the highest justified checkpoint. This contradiction proves the lemma. □

We saw with Lemma 6.1 that two checkpoints of the same epoch could not be justified, hence finalized. We then showed with Lemma 6.2 that after a finalization on one chain, no checkpoints could become justified on any other chain. These are the conditions required to have safety, as we prove now.

Theorem 6.3 (Safety). *There cannot be two finalized checkpoints on different chains.*

Proof. Thanks to Lemma 6.1, we know that two different checkpoints C and C' of the same epoch cannot be justified, hence finalized.

For the sake of contradiction, let us assume that two checkpoints C and C' are finalized on different chains c and c' at epoch e and e', respectively. We assume without loss of generality that $e < e'$. C being finalized, we know thanks to Lemma 6.2 that C' cannot be justified on a different chain c', let alone be finalized. □

The blockchain preserves the property of safety at all times. The Ethereum PoS is safe.

7 RELATED WORKS

Blockchain protocol analyses can be divided into two main categories: those that specify or formalize protocols and those that identify vulnerabilities of the protocols. Our work is at the junction of the two categories because we both formalize the protocol to permit its analysis, and we present a novel attack.

The category of specification and formalization includes the "white papers" (e.g., Bitcoin [18], Ethereum [28]). It also includes academic papers providing formal specifications and demonstrating properties guaranteed by the protocols. For example, [13] formally describes and analyses the Bitcoin protocol, proving its security guarantees, [2] does the same for the Tendermint's protocol (the consensus protocol of the Cosmos blockchain[6]). Our work lies in this category of

[22] See section 4.1.4 for details on justification.
[23] See section 4.1.4 for details on finalization.

formalization by proposing a specification of Ethereum Proof-of-Stake protocol's properties and a high-level description through pseudo-code. For what concerns protocol formalization of Ethereum Proof-of-Stake, [7] is the first to propose draft specification of the Ethereum PoS protocol and related properties. However, that specification is outdated and not complete. Our paper provides a high-level formalization of the consensus mechanisms with respect to the current implemented code, along with a novel specification of its properties.

A famous example of papers identifying vulnerabilities is [10], which presents the selfish mining attack on Bitcoin. [10] shows that in Bitcoin (and proof-of-work in general), miners can benefit from deviating from the prescribed protocol by withholding blocks for a while at the expense of honest miners. [1] points out a liveness vulnerability of the Tendermint protocol. [23] presents an attack where nodes can reorganize the Tezos' Emmy+ chain and then do a double-spend attack. Our work follows the line of research focusing on flaws of Ethereum Proof-of-Stake [21, 22, 26]. Neu et al. [21] exhibit a balancing attack, highlighting the shortcomings of a consensus mechanism being separated into two layers (finality gadget, fork choice rule). Mitigation against this attack was proposed, but [22] overcame this mitigation with a new balancing attack. [26] presented reorg attacks revealing that validators with the role of proposer could gain from disturbing the protocol by releasing their block late. * Our paper presents another flaw regarding the liveness of the current Ethereum Proof-of-stake protocol, on which some attacks do not seem feasible anymore, and thus insists on the importance of finding new ways to conciliate availability and finality. We differ from [12] that aims at formally verifying the protocol of *Hybrid Capser* focusing on an outdated version of the protocol. While we formalize through pseudo-code the current implemented version of the protocol and exhibit a liveness attack on the protocol.

Nakamura [20] presents an attack called *splitting attack* in which the adversary sends messages to split the set of validators. However, to achieve this attack, Nakamura assumes that the adversary needs to control and play with network delays. This is a strong assumption and can be considered unrealistic. More recently, [26] showed through experiments that attackers can predict the proportion of validators receiving a given message within a specific time frame with sufficient accuracy. This contradicts Nakamura's claim that the attack necessitates the adversary to control the network delay. In this work, we present a form of the splitting attack based on the weaker assumption that the adversary knows the network delay (in line with [26]) but does not control it. Moreover, our attack is repeated, hence the name bouncing, being a threat to the liveness.

Outside of these two categories lies works that provide formal ground for blockchains. [4] describes the types of finality a blockchain can achieve, [3] proposes a formalization of blockchains and their evolutions as BlockTrees. We rely on the definition of BlockTree and finality to express the Ethereum protocol properties.

8 CATEGORIZATION OF ATTACKS ON ETHEREUM PROOF-OF-STAKE

Ethereum Proof-of-Stake has a history of attacks targeting the liveness of the protocol. These attacks can be divided into two main groups: the one targeting the fork choice rule -*Partitioning over the candidate chains*-, and the one targeting the justification of checkpoints -*Swinging over the justified chains*. The former aims at dividing the set of honest validators equally on two concurrent chains. The latter aims at preventing finalization by alternatively justifying on two concurrent chains.

This categorization is summarized in Table 1. The controllable setup indicates the ease for Byzantine validators to start the attack. A controllable setup is one that Byzantines can perform during the synchronous period without requiring a prior state in the asynchronous period. The Resulting Effect expresses the state of the blockchain the attack achieves on Liveness. The Detectable columun indicates if honest validators can detect Byzantine validators that mounted the attack.

Partitioning over the candidate chains. The first attack in this category is the Balancing attack [21]. As for all the Partition attacks, the Balancing attack's purpose is to keep half of the honest validators on one chain and half on another. This fork is then balanced using the Byzantines' votes to ensure that honest validators are kept in check. The execution of the attack is the following. The first proposer of the epoch needs to be Byzantine and proposes 2 blocks. The 2 blocks are released to different parts of the networks to ensure that honest validators disagree on which block was first and thus the one belonging to the candidate chain. Byzantine validators then use their vote to keep the number of validators on each chain balanced. This attack is detectable since the first proposer needs to propose 2 different blocks. It is the only one among the Byzantine validators that performs a visibly reproachable action.

Conversely to other attack that aims at partitioning the validators over two candidates, Refined Balancing attack [26] don't need a Byzantine validator to perform a detectable action. To launch the attack, the Byzantine validators wait for an opportune epoch in which they are elected to propose a block for the two first slots. The Refined Balancing attack is a Balancing attack without the assumption of adversarial network delay. In their model, the adversarial network delay is the capacity for an adversary to arbitrarily delay messages sent in the network, bounded by the network delay. They manage to perform a Balancing attack without an adversarial network delay by gaining knowledge about the network delay by having Byzantine validators scattered in the network. To mitigate this attack and the previous Balance attack, a 'proposer boost score' was suggested and implemented[24]. This mitigation did not actually stop the attack as shown by the LMD-Specific attack.

Another Partition attack is the LMD-Specific Balancing attack [22]. This attack works similarly to other balancing attacks; Byzantine validators create two concurrent chains, and they keep the two chains balanced (half honest validator on each). The main difference with other balancing attacks is that instead of only having a noticeable Byzantine block proposer, a handful of Byzantine validators release votes for both chains using equivocation. This creates two sets of honest validators with different candidate chains depending on which vote they have seen first. This attack is effective even after the patch proposed to stop the Balancing Attack.

Swinging over the justified chains. This category of attacks starts with the Bouncing attack [19]. This attack first outlines a limitation of the Casper FFG protocol, the underlying protocol used to finalize blocks. This limitation resides in that honest validators have to cast checkpoint votes for checkpoints stemming from the last justified checkpoint (having the last justified checkpoint as an ancestor). Using this restriction, Byzantine validators can split the set of validators by ensuring that they don't have the same view of the last justified checkpoint when they make their checkpoint vote. This restriction to only vote for checkpoints (and block) stemming from the justified checkpoints with the highest epoch set by the fork choice rule is used for all justification attacks. We presented the patch (cf. subsection 5.2) that has been implement to prevent this attack.

The splitting attack [20] presents an attack in which Byzantine validators with 'strong control over the network' can split honest validators into two sets. They can do so by sending attestations late enough for some validators to receive them on time while the rest don't. This attack can lead to a situation where a checkpoint gets justifiable but not justified. This means that the set of validators is divided between two checkpoints they try justify. The attack's type is thus a partition over the justified chains.

Our attack differentiates from the splitting attack from two main points. We do not assume Byzantine validators to have control over the network but rather they possess the usual power of equivocation. We also repeat the process of

[24]The pull request to add this mitigation can be found here: https://github.com/ethereum/consensus-specs/pull/2730.

Table 1. Summary of attacks against Ethereum's liveness

Attack original name	Controllable Setup	Attack's type	Resulting liveness	Detectable
"Balancing Attack" [21]	✓	Partitioning over the candidate chains	Stopped	✓
"Refined Balancing Attack" (Balancing attack without adversarial network delay) [26]	✓	Partitioning over candidate chains	Stopped	✗
"LMD-Specific Balancing Attack" [22]	✓	Partitioning over candidate chains	Stopped	✓
"Bouncing Attack" [19]	✗	Swinging on justified chains	Stopped	✗
"Splitting Attack" [20]	✗	Partitioning over the justified chains	Delayed for 1 epoch	✗
Probabilistic Bouncing Attack (this work)	✗	Swinging on justified chains	Delayed for k epochs with decreasing probability	✗

dividing the set of validators on two different chains at each epoch while expressing the probability of the attack to continue. This makes this attack's type fit the swinging on justify chains.

All the attacks to justification chains do not have a controllable setup and share the feature of not being detectable. These attacks rely mainly on messages being viewed at a certain point in time for some set of honest validators and at another time for the rest. There is no need for Byzantine validators to duplicate votes; they just have to withhold their messages/votes and release at the appropriate time.

9 CONCLUSION

We described a framework for a high-level description of the protocol. This is the first step in providing a formalization for verification tools. We proposed a novel distinction between the definition of liveness and availability. This distinction is crucial to pinpoint the difference between Nakamoto-style and BFT consensus. It makes possible a comparison between the two. We outlined an attack against the liveness of the protocol, showing probabilistic liveness of the protocol under this attack and we prove safety of the protocol. Another aspect is that our analysis did not consider the protocol's rewards and incentives. We leave this part, as well as analysing rational behavior in the protocol, as future work.

REFERENCES

[1] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. 2018. Correctness of Tendermint-Core Blockchains. In *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China.* 16:1–16:16.

[2] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. 2019. Dissecting Tendermint. In *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11704).* Springer, 166–182.

[3] Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. 2019. Blockchain Abstract Data Type. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019.* ACM, 349–358.

[4] Emmanuelle Anceaume, Antonella Del Pozzo, Thibault Rieutord, and Sara Tucci-Piergiovanni. 2021. On Finality in Blockchains. In *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France.* 6:1–6:19.

[5] Lacramioara Astefanoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni, and Eugen Zalinescu. 2021. Tenderbake - A Solution to Dynamic Repeated Consensus for Blockchains. In *4th International Symposium on Foundations and Applications of Blockchain 2021, FAB 2021, May 7, 2021, University of California, Davis, California, USA (Virtual Conference).*

[6] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus.

[7] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. 2020. Combining GHOST and Casper.

[8] Consensys. 2022. Teku Consensus Client. https://github.com/ConsenSys/teku/tree/bada6df06edd1f8b5d727a011f440d9825e17d99

[9] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* (1988), 288–323.

[10] Ittay Eyal and Emin Sirer. 2018. Majority Is Not Enough: Bitcoin mining is vulnerable. *Commun. ACM* (2018), 95–102.

[11] Ethereum Foundation. 2022. Consensus specifications GitHub. https://github.com/ethereum/consensus-specs/tree/ae89e4e6158344a5ab736d834e2239efd431ef5f/specs

[12] Letterio Galletta, Cosimo Laneve, Ivan Mercanti, and Adele Veschetti. 2022. Resilience of Hybrid Casper under Varying Values of Parameters. *Distrib. Ledger Technol.* (2022).

[13] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *Advances in Cryptology - EUROCRYPT 2015*. 281–310.

[14] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* (2002).

[15] L.M. Goodman. 2014. Tezos — a self-amending crypto-ledger.

[16] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. 2012. An Enciphering Scheme Based on a Card Shuffle. In *Advances in Cryptology – CRYPTO 2012*. Springer Berlin Heidelberg.

[17] Jae Kwon and Ethan Buchman. 2016. Cosmos.

[18] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008).

[19] Ryuya Nakamura. 2019. Analysis of bouncing attack on FFG. https://ethresear.ch/t/analysis-of-bouncing-attack-on-ffg/6113

[20] Ryuya Nakamura. 2019. Prevention of bouncing attack on FFG. https://ethresear.ch/t/prevention-of-bouncing-attack-on-ffg/6114

[21] Joachim Neu, Ertem Nusret Tas, and David Tse. 2021. Ebb-and-Flow Protocols: A Resolution of the Availability-Finality Dilemma. In *2021 IEEE Symposium on Security and Privacy (SP)*. 446–465.

[22] Joachim Neu, Ertem Nusret Tas, and David Tse. 2022. Two Attacks On Proof-of-Stake GHOST/Ethereum. Cryptology ePrint Archive, Paper 2022/289.

[23] Michael Neuder, Daniel J. Moroz, Rithvik Rao, and David C. Parkes. 2020. Defending Against Malicious Reorgs in Tezos Proof-of-Stake. In *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*. 46–58.

[24] Prysm. 2022. Code Consensus Client. https://github.com/prysmaticlabs/prysm/tree/cafe0bd1f81298ee34b450ab8d7e74b0036e9803

[25] Specification Pull Request. 2019. Bouncing attack patch. https://github.com/ethereum/consensus-specs/pull/1465

[26] Caspar Schwarz-Schilling, Joachim Neu, Barnabé Monnot, Aditya Asgaonkar, Ertem Nusret Tas, and David Tse. 2022. Three Attacks on Proof-of-Stake Ethereum. In *Financial Cryptography and Data Security*, Ittay Eyal and Juan Garay (Eds.). Springer International Publishing, 560–576.

[27] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*.

[28] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* (2014), 1–32.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009