The following is joint work with Barnabé Monnot (@barnabe), Stefanos Leonardos (@sleonardos), and Georgios Piliouras (@georgios).

Medium-term security against 51% attacks is an integral research question as emphasized in Vitalik's earlier posts on timeliness detectors and 51% attacks in Casper FFG. The main challenge is to achieve additional security without "major modifications to longest-chain-based fork choice rules" and without "including new classes of attestations" - in other words: without increasing the communication complexity.

In this post, we build upon the idea that "timeliness detectors can offer a kind of finality" and try to contribute to this discussion. We aim to design a framework that increases security against 51% attacks while addressing the above challenges. As such, we propose a CBC-like finality gadget that can be overlaid onto longest-chain proof-of-stake protocols like LMD GHOST and which completely piggybacks on the underlying proof-of-stake algorithm, especially on the existing attestations. As in CBC Casper, each user i

is free to set her own security threshold \alpha_i \in [0, 1]

that represents the highest attacker-controlled stake fraction that she is willing to tolerate.

The core idea behind our algorithm is that we can leverage the blockchain architecture to efficiently implement a variation of Lamport's SM algorithm from the original Byzantine Generals paper. In the context of the Byzantine generals problem, the SM algorithm achieves high security by tracking which generals have provably indicated that they have seen each message. To do so, each general does not just send a single signed message, but they also, upon receiving another general's message, add their own signature and send the combined message to the other generals. In this way, a "chain" of signed messages is created. The original algorithm by Lamport has a high message overhead, but in a blockchain, an attestation for a block is also an attestation for each of its ancestors, and long chains of signed messages therefore emerge naturally.

In concrete terms, our approach is as follows. During protocol execution, we track for each block the fraction of validators that have supported

that block or one of its descendants through a block proposal or attestation. Each validator finalizes a block if the block's supporting fraction has become high enough to meet her chosen security threshold. To determine the supporting fractions efficiently, we also track for each validator the latest block that she has proposed or attested to. Every time a new block proposal or attestation is processed by the algorithm, the supporting fraction in all blocks that are between the validator's latest block and the new one is increased. As the blockchain grows, the supporting fraction for every main chain block will typically grow to 100% as well.

In practice, it is possible that a validator proposes or attests to a block that is on a branch of the blockchain that is different from her latest block's branch. In this case, the algorithm efficiently switches branch, and incorporates any possible inconsistencies between the validator's deposits on the two branches of the blockchain. Meanwhile, the algorithm determines whether any slashing conditions

have been violated; if so, the offending validator is penalized. As in Casper FFG, we aim to have two slashing conditions. The first one is the same: if a validator produces two conflicting blocks or attestations within the same time slot, then this is clearly evidence of misbehavior. However, we have a different slashing condition than Casper FFG for validators who switch branch in a later time slot: in Casper FFG, switching branches is allowed if a block has been finalized on the new branch. However, in our setting there is no protocol-wide agreement on whether a block is finalized or not (as different users may have different thresholds). Hence, we instead require that the validator is only allowed to switch to a stronger

branch: that is, the validator is only allowed to propose or attest to a block on a different branch if this block would be preferable to the validator's latest block using the underlying protocol's fork-choice rule.

In the remainder of this post we will discuss our assumptions about the underlying proof-of-stake protocol, our notation, and the algorithm. We conclude with a discussion of the algorithm's properties and some open questions.

The underlying protocol

We make the following assumptions about the underlying chain-based proof-of-stake protocol. Time is divided into slots (e.g., 5 seconds per slot). In every slot, a random beacon is used to select a committee and a block proposer from a known set of validators. The role of the committee members is to produce attestations

– each committee member attests to the block that she currently considers to be the head of the blockchain. Each attestation includes the attesting validator's index, a time slot number, a target block, and a valid signature. The block proposer creates a new block that contains (among other data fields) a reference to the previous block, its time slot index, the blocks' transactions (including deposits/withdrawals of stake), and attestations to any of the block's ancestors that had not been included in any of its ancestors. As in Ethereum's GHOST, we may assume that included attestations cannot be from too far in the past (e.g., older than 6 slots), but we do not require this in the following.

Notation

We use the following notation. Blocks and validators are represented by integers (for example, for blocks this integer could

be the hash of the block). We represent attestations by three-dimensonal integer arrays/tuples, i.e., a \in \mathbb{N}^3

such that a = (V^{\textit{att}}(a), T^{\textit{att}}(a), B^{\textit{att}}(a))

such that V^{\textit{att}}(a)

is the validator who creates the attestation, T^{\textit{att}}(a)

is the time slot in which the attestation is produced, and B^{\textit{att}}(a)

is the block that it attests to.

We also consider deposits/withdrawals, which are transactions that convert non-staked ETH into staked ETH or vice versa. We represent a deposit/withdrawal \delta

as a tuple (V^{\textit{dep}}(\delta),Z^{\textit{dep}}(\delta))

such that V^{\textit{dep}}(\delta) \in \mathbb{N}

is the validator who made the deposit/withdrawal and Z^{\textit{dep}}(\delta) \in \mathbb{Z}

its size (positive values of Z^{\textit{dep}}(\delta)

indicate new deposits, whereas negative values indicate withdrawals). Deposits also change due to rewards

: we denote by \rho^{\textit{prop}}

the block proposal reward and by \rho^{\textit{att}}

the reward paid to a validator when her attestation is included in a block. We assume that all stakes and rewards can be represented by integers (namely in wei).

For a block b \in \mathbb{N},

let A(b) \subset \mathbb{N}^3

be the set of attestations included in b

(which attest to the ancestors of b

), and \Delta(b) \subset \mathbb{N}^2

the set of deposits/withdrawals included in (and effective starting from) b.

Let P (b) \in \mathbb{N}

denote the previous block of block b \in \mathbb{N},

let V^{\textit{prop}} (b) \in \mathbb{N}

denote the proposer of b,

and let T^{\textit{prop}} (b) \in \mathbb{N}

denote the time slot in which block b

was proposed. Let g

denote the genesis block. We also define H(b) \in \mathbb{N}

as the effective height

of block b,

which is the number of steps needed to go back to g

(this is not necessarily the same as the timeslot index due to forks or proposers being temporarily offline).

Algorithm

Each validator runs a blockchain client, that is, the software application that handles the interactions with the blockchain, in particular the sending, receiving, and processing of blocks and attestations. The client keeps track of the variables D, L, S,

and S^{\textit{max}},

which represent the following quantities from the perspective of the client:

- for each validator v \in \mathbb{N},
- D(v) \in \mathbb{N}

represents v

's deposit,

- L(v) \in \mathbb{N}

represents the last block that v

proposed or attested to,

- D(v) \in \mathbb{N}

represents v

's deposit,

- L(v) \in \mathbb{N}

represents the last block that v

proposed or attested to,

- for each known block b \in \mathbb{N},
- S(b) \in \mathbb{N}

represents the stake that has supported b

(by validators proposing or attesting to it) or one of its descendants.

- S^{\textit{max}}(b) \in \mathbb{N}

represents the total possible stake S^{\textit{max}}(b)

that can ever

support b.

- S(b) \in \mathbb{N}

represents the stake that has supported b

(by validators proposing or attesting to it) or one of its descendants.

- S^{\textit{max}}(b) \in \mathbb{N}

represents the total possible stake S^{\textit{max}}(b)

that can ever

support b.

It also keeps track of a list B

of previously processed blocks. The complete algorithm then works as follows. It is initialized using the function \texttt{initialize},

which takes a genesis block and a list of validator indices and deposits as input, which works as follows (a working version of the code in this post can be found here):

def **init**(self, genesis, validators, deposits): self.g = genesis self.B = [genesis] self.S = {} self.S_max = {} self.D = {} self.L = {} self.S[genesis] = 0 self.S_max[genesis] = 0 for i in range(len(validators)): v = validators[i] d = deposits[i] self.D[v] = d self.L[v] = genesis self.S[genesis] += d self.S_max[genesis] += d

In words, the genesis block is stored as g,

and the values for L,

S,

and S^{\textit{max}}

are set as if every validator attested to g.

When the client receives a new block b,

it executes \texttt{ProcessBlock}

:

def processBlock(self, b): isValid = self.validateBlock(b) if isValid: self.B.append(b) self.S[b] = 0 self.S_max[b] = self.S_max[b.previous] + BLOCK_REWARD for attestation in b.attestations: v = attestation.validator t = attestation.targetblock self.processAttestation(v, t) self.S_max[b] += ATTESTATION_REWARD for delta in b.deltas: z = delta.size self.S_max[b] += z self.processAttestation(b.proposer, b)

First, b

is validated, which means checking that:

1. It has not been processed before
2. All of its ancestor blocks have been processed
3. None of its attestations/transactions have already been included in any of its ancestors, and
4. The attestations are sorted by the height H

of the target block.

If the block is valid, then the supporting stake S

of b

is initialized to 0,

and it is given the same potential supporting stake S^{\textit{max}}

as the previous block plus the block proposal reward \rho^{\textit{prop}}.

Next, for every attestation a

in A(b),

the algorithm increases S^{\textit{max}}

by the attestation reward \rho^{\textit{att}},

and the function \texttt{processAttestation}

(as described below) is called. Next, the impact of the deposits/withdrawals on S^{\textit{max}}

is processed. Finally, the block is treated as an attestation by the block proposer for itself.

The function \texttt{processAttestation}

works as follows:

def processAttestation(self, v, b): lastBlock = self.L[v] blocks = self.getBlocksBetween(b, lastBlock) if len(blocks) == 0 and b != lastBlock or len(blocks) > 0 and blocks[len(blocks)-1].previous != lastBlock: self.processConflict(v, b, lastBlock) else: self.processForward(blocks, v) self.L[v] = b

First, the algorithm determines the chain of blocks from b

back to L(v)

: if there is no such chain because the blocks are on different branches, then the function returns the chain of blocks between b

and g.

If the chain of block indeed ends at g,

and L(v) \neq g,

then there has been a conflict, which is processed using the \texttt{processConflict}

function. If not, the \texttt{processForward}

function is called. Finally, L(v)

is updated to b.

We first discuss normal operation, which means that \texttt{processForward}

is called. This function works as follows:

def processForward(self, blocks, v): for b in reversed(blocks): self.updateDeposit(b, v, 1) self.S[b] += self.D[v]

def updateDeposit(self, b, v, multiplier): for a in b.attestations: if v == a.validator: self.D[v] += multiplier * ATTESTATION_REWARD for delta in b.deltas: if v == delta.validator: self.D[v] += multiplier * delta.size if v == b.proposer: self.D[v] += multiplier * BLOCK_REWARD

In words, the algorithm iterates over each block b

in the \texttt{blocks}

array and processes all the changes in D(v)

due to b

and its included attestations and deposits/withdrawals. Meanwhile, it increases the supporting stake S

for each block b

given the validator's current deposit D.

If the new block b

is in conflict with v

's last attested block, then the following function is called:

def processConflict(self, v, newBlock, lastBlock): slashable = self.isSlashable(newBlock, lastBlock) if slashable: self.slash(v) else: c = self.findLastCommonAncestor(newBlock, lastBlock) oldChain = self.getBlocksBetween(lastBlock, c) newChain = self.getBlocksBetween(newBlock, c) l = self.getLastAttestedBlock(newChain, v) newChainPre = self.getBlocksBetween(l.previous, c) newChainPost = self.getBlocksBetween(newBlock, l.previous) self.updateDeposits(oldChain, v, -1) self.updateDeposits(newChainPre, v, 1) self.processForward(newChainPost, v)

def updateDeposits(self, blocks, v, multiplier): for b in blocks: self.updateDeposit(b, v, multiplier)

That is, the algorithm first determines whether a slashable offense has been committed. If not, then the algorithm finds the last common ancestor of the two blocks. The changes to the validator's deposit on the old branch are rolled back, and the new branch is divided into the part to which the validator had previously attested and the part to which she had not (this is necessary because we do not rule out that the validator switches to a chain and then back). The validator's deposit is then updated to reflect her balance at the end of the first part of the chain, and the \texttt{processForward}

function is then called on the remainder of the new branch.

Example

To illustrate the algorithm, we present a sample evolution of the supporting stakes in a toy system. Our system has five validators: v_1,\ldots,v_5,

with deposits of 10,15,20,25,

and 30

ETH, respectively. The total stake equals 100

ETH. In this example, we assume that a committee of size 2

is sampled in every round, and that the reward for block proposers is 10

ETH and 1

ETH for attestations. For simplicity, we consider the case with no forks – i.e., there are no two blocks with the same effective height H.

We assume that in each time slot, the new block is created and broadcast at the start of the slot, and that the attestations are sent out after the first half of the slot has passed. Due to network latency, an attestation may not correspond to the latest block, and an attestation may not make into the next round's block.

The following table contains a sample execution and contains for each of the first seven rounds of the protocol the new block, the proposer, the committee, and the attestations included in that block's round.

round

new block

proposer

committee

include attestations

1

b_1

v_1

v_3,v_4

-

2

b_2

v_2

v_4,v_5

(v_4,1,b_1)

3

b_3

v_3

v_1,v_2

(v_3,1,b_1),

(v_4,2,b_1),

(v_5,2,b_2)

4

b_4

v_5

v_1,v_5

(v_1,3,b_3),

(v_2,3,b_2)

5

b_5

v_4

v_2,v_5

(v_1,4,b_3),

(v_5,4,b_4)

6

b_6

v_3

v_1,v_3

(v_2,5,b_5),

(v_5,5,b_5)

7

b_7

v_5

v_1,v_2

(v_1,6,b_6),

(v_3,6,b_6)

The first block contains no attestation, and the second block contains only one attestation as the second has not yet been received. Validator v_4

is chosen as a committee member in both rounds 1 and 2, but attests to the same block as she has not received block b_2

when broadcasting its attestation for round 2. Similarly, v_1

attests to the same block b_3

in both round 3 and 4. The evolution of the supporting stake fraction S/S^{\textit{max}}

during these seven rounds is as follows:

round

b_1

b_2

b_3

b_4

b_5

b_6

b_7

1

\frac{20}{110}

-

-

-

-

-

-

2

\frac{60}{110}

\frac{25}{121}

-

-

-

-

-

3

\frac{110}{110}

\frac{75}{121}

\frac{31}{134}

-

-

-

-

4

\frac{110}{110}

\frac{95}{121}

\frac{82}{134}

\frac{41}{146}

-

-

-

5

\frac{110}{110}

\frac{121}{121}

\frac{109}{134}

\frac{68}{146}

\frac{37}{158}

-

-

6

\frac{110}{110}

\frac{121}{121}

\frac{134}{134}

\frac{125}{146}

\frac{136}{158}

\frac{41}{170}

-

7

\frac{110}{110}

\frac{121}{121}

\frac{134}{134}

\frac{146}{146}

\frac{158}{158}

\frac{106}{170}

\frac{53}{182}

After b_1

is processed, the deposit of v_1

increases from 10

to 20

due to the block proposal reward. The maximum supporting stake, namely 100+10=110

ETH, is obtained by summing the total initial deposits (100)

plus the block proposal reward for v_1

(10)

. The stake of v_1

is added to the supporting stake of block b_1,

which means that 20

out of 110

stake units support b_1.

In the next round, v_2

proposes a block b_2

that includes the attestation to b_1

by validator v_4

from round 1, who was a committee member in that round. However, the attestation by v_3

to b_1

has not been received yet by v_2

so it is not included. Validator v_2

earns the block reward and v_4

earns one stake unit for its attestation, and the maximum supporting stake for b_2

is therefore equal to 121.

After b_2

has been processed, v_1,

v_2,

and v_4

have all implicitly attested to block b_1.

In block 1, the deposit sizes of these three validators were 10+10=20,

15,

and 25,

respectively. Therefore, the supporting stake for b_1

equals 10+15+25=60,

and the supporting stake fraction equals \frac{60}{110}.

Meanwhile, only v_2

has attested to block 2, and since the deposit of v_2

in round 2 was equal to 15+10=25,

this means that the supporting stake for block 2 after round 2 equals \frac{25}{121}.

In round 3, validator v_3

proposes block b_3

which includes a second attestation by v_4

for b_1,

an attestation by v_5

for b_2,

and also v_3

's own attestation to b_1

from the previous round. Because v_5

's attestation for b_2

is also implicitly an attestation for b_1,

all validators have now attested to b_1,

and its supporting stake equals \frac{110}{110}.

In this round, v_3

earns the block reward and one stake unit for its attestation, and v_4

and v_5

earn a stake unit each for their attestation, taking the maximum supporting stake for block b_3

to 134.

(Note that an attester only earns a reward when their attestation is included in a block.) Validator v_3

is the only one to have attested to block b_3,

and since her deposit size equals 20+10+1=31

(initial deposit plus one block proposal reward and one attestation reward), the supporting stake fraction for b_3

after round 3 equals \frac{31}{134}.

As time progresses, the supporting stake for all blocks increases, and at the end of round 7 all validators have attested to blocks b_1,

b_2,

b_3,

b_4,

and b_5,

so the supporting stake fraction equals 100\%.

Next Steps

An initial implementation of the algorithm can be foundhere. We are still working on refining the algorithm to handle edge cases, and on implementing the slashing conditions. We are also working on integrating the algorithm with the beacon runner by the Robust Incentive Group over the course of the next few weeks. We welcome comments and suggestions from the Ethereum community.

Final Remarks

- Note that the maximum supporting stake fraction and the validator deposits for each block are fixed when the block is created. That is, if a validator's deposit changes in block b,

then this has no impact on the strength of her support for any of b

's ancestors.

- Building on the previous remark: in the above example, the supporting stake fraction for each block is higher than the supporting stake fraction for any of its descendants. In general, this is not necessarily the case: for example, if a validator who controls 20% of the stake goes permanently offline, then the blocks immediately after the last block to which she attested will never

be able to get a supporting stake fraction that is higher than roughly 80%. However, as time progresses, the stake of the other validators will probably increase due to rewards, so the offline validator's relative stake fraction may shrink to zero, and the supporting stake for new blocks to 100%.

To ensure safe finalization, it is probably not enough to just consider the supporting stake fraction for a block, but also the smallest supporting stake fraction for any of its ancestors. If the supporting stake fraction for a block in the far past is never able to go above a certain threshold due to permanently offline validators, then it may be helpful to agree not to revert blocks from too far in the past. This is left to the security preferences of each user.

- Should it be possible to include an attestation to block b

in block b'

if b

and b'

are in conflict? (This would be similar to uncles in the legacy chain.)

- If an attestation attests to a block that is too far in the past, should it still be possible to include it, or should the rewards be reduced?