Casper the Friendly Finality Gadget

Vitalik Buterin and **Virgil Griffith**
Ethereum Foundation

Abstract

We give an introduction to the consensus algorithm details of Casper: the Friendly Finality Gadget, as an overlay on an existing proof of work blockchain such as Ethereum. Casper is a partial consensus mechanism inspired by a combination of existing proof of stake algorithm research and Byzantine fault tolerant consensus theory, which if overlaid onto another blockchain (which could theoretically be proof of work or proof of stake) adds strong *finality* guarantees that improve the blockchain's resistance to transaction reversion (or "double spend") attacks.

1. Introduction

The past few years there has been considerable research into "proof of stake" (PoS) based blockchain consensus algorithms. In a PoS system, a blockchain appends and agrees on new blocks through a process where anyone who holds coins inside of the system can participate, and the influence someone has is proportional to the number of coins (or "stake") they hold. This is a vastly more efficient alternative to proof of work "mining", allowing blockchains to operate without mining's high hardware and electricity costs.

There are two major schools of thought in PoS design. The first, *chain-based proof of stake*, mimics proof of work mechanics featuring a chain of blocks and an algorithm that "simulates" mining by pseudorandomly assigning the right to create new blocks to stakeholders. This includes Peercoin [1], Blackcoin[2], and Iddo Bentov's work[3].

The other school, *Byzantine fault tolerant* (BFT) based proof of stake, is based on a thirty year old body of research into BFT consensus algorithms such as PBFT[4]. BFT algorithms typically have proven mathematical properties; for example, one can usually mathematically prove that as long as $> \frac{2}{3}$ of protocol participants are following the protocol honestly, then, regardless of network latency, the algorithm cannot finalize conflicting block hashes (called "safety"). Repurposing BFT algorithms for proof of stake was first introduced by Tendermint[5].

1.1. Our Work

We follow the BFT tradition, though with some modifications. Casper the Friendly Finality Gadget is an *overlay* atop a *proposal mechanism*—a mechanism which proposes *checkpoints*. Casper is responsible for *finalizing* these checkpoints. Casper provides safety, but does not guarantee liveness—Casper depends on the proposal mechanism for liveness. That is, even if the proposal mechanism is wholly controlled by attackers, Casper prevents attackers from finalizing two conflicting checkpoints, however, the attackers can prevent Casper from finalizing any future checkpoints.

Our algorithm introduces several new properties that BFT algorithms do not necessarily support.

- We flip the emphasis of the proof statement from the traditional "as long as $> \frac{2}{3}$ of validators are honest, there will be no safety failures" to the contrapositive "if there is a safety failure, then $\geq \frac{1}{3}$ of validators violated some protocol rule."

- We add *accountability*. If a validator violates the rules, we can detect the violation, and know who violated the rule. "$\geq \frac{1}{3}$ violated the rules, *and we know who they are*". Accountability allows us to penalize malfeasant validators, solving the *nothing at stake* problem [cite] that plagues chain-based PoS. The penalty is the validators' entire deposits. This maximum penalty is provides a bulwark against violating the protocol by making violations immensely expensive. Protocol guarantees is much higher than the size of the rewards that the system pays out during normal operation. This provides *stronger* security guarantees than with proof of work.
- We introduce a provably safe way for the validator set to change over time (Section 4).
- We introduce a way to recover from attacks where more than $\frac{1}{3}$ of validators drop offline, at the cost of a very weak *tradeoff synchronicity assumption* (Section 5).
- The design of the algorithm as an overlay makes it easier to implement as an upgrade to an existing proof of work chain.

We will describe the protocol in stages, starting with a simple version (Section 2) and then progressively adding features such as validator set changes (Section 4) and mass liveness fault recovery (Section 5).

2. The Casper Protocol

In the simple version, we assume there is a set of validators and a *proposal mechanism* which is any system that proposes a sequence of blocks (such as a proof of work chain)

We order the sequence of blockhashes into a sequence called a *blockchain* $\mathbf{B}$. We assume all blocks $b \in \mathbf{B}$ are unique. The elements of sequence $\mathbf{B}$ obey a total ordering where $b < b'$ if and only if block b precedes block b' in within sequence $\mathbf{B}$. Within blockchain $\mathbf{B}$ is there is a subset called *checkpoints*,

$$\begin{aligned} &\mathbf{B} \equiv (b_0, b_1, b_2, \ldots) \\ &\mathbf{C} \equiv (b_0, b_{99}, b_{199}, b_{299}, \ldots) \ . \end{aligned} \tag{1}$$

This leads to the formula for an arbitrary checkpoint,

$$C_i = \begin{cases} b_0 & \text{if } i = 1, \\ b_{100*(i-2)+99} & \text{otherwise.} \end{cases} \tag{2}$$

The proposal mechanism will initially be the existing Ethereum proof of work chain, making the first version of Casper a *hybrid PoW/PoS algorithm* that relies on proof of work for liveness but not safety, but in future versions the proposal mechanism can be substituted with something else.

An *epoch* is defined as the contiguous sequence of blocks between two checkpoints. The *epoch of a block* is the index of the epoch containing that hash, e.g., the epoch of block 599 is 5.[1]. Likewise, the epoch of checkpoint C_n is simply $n - 1$.

Each validator has a *deposit*; when a validator joins their deposit is the number of coins that they deposited, and from there on each validator's deposit rises and falls with rewards and penalties. For the rest of this paper, when we say "$\frac{2}{3}$ of validators", we are referring to a *deposit-weighted* fraction; that is, a set of validators whose sum deposit size equals to at least $\frac{2}{3}$ of the total deposit size of the entire set of validators. "$\frac{2}{3}$ prepares" will be used as shorthand for "prepares from $\frac{2}{3}$ of validators".

Validators can broadcast two types of messages: $\left\langle \textbf{PREPARE}, \vec{c}, \vec{e}, c, e \right\rangle$ and $\langle \textbf{COMMIT}, c, e \rangle$, as detailed in Figure 1.

Every checkpoint c can be *Justified*. Justified checkpoints can then also be *Finalized*. Every checkpoint starts as neither Justified or Finalized.

[1]To get the epoch of a particular block b_i, it is $epoch(b_i) = \lfloor i/100 \rfloor$.

$$
\begin{aligned}
\mathbf{J} &= (c \in \mathbf{C} : \text{valid\_prepares}(c) \geq 2/3) \\
\mathbf{F} &= (j \in \mathbf{J} \ : \text{valid\_commits}(j) \geq 2/3)\ .
\end{aligned}
\tag{3}
$$

Which leads to the pleasing relation, $\mathbf{F} \subseteq \mathbf{J} \subseteq \mathbf{C} \subset \mathbf{B}$.

Notation	Description
$\vec{c}$	the hash of any Justified checkpoint
$\vec{e}$	the epoch of checkpoint $\vec{c}$
c	any checkpoint hash after $\vec{c}$
e	the epoch of checkpoint c
$\mathcal{S}$	signature of $\left\langle \textbf{PREPARE}, \vec{c}, \vec{e}, c, e \right\rangle$ from the validator's private key

(a) **PREPARE** message

Notation	Description
c	a Justified checkpoint hash
e	the epoch of the checkpoint c
$\mathcal{S}$	signature of $\langle \textbf{COMMIT}, e, c \rangle$ from the validator's private key

(b) **COMMIT** message

Figure 1: The schematic of **PREPARE** and **COMMIT** messages.

Requirements for accepting a prepare message (Figure 1a):

1. Hash $\vec{c}$ must be the checkpoint for epoch $\vec{e}$. $c = b_{100\vec{e}+99}$
2. Hash $\vec{c}$ must be Justified. Equivalently, valid_prepares$(c) \geq 2/3$.
3. Hash c must be the checkpoint for epoch e. Equivalently, $c = b_{100e+99}$.
4. Epoch $\vec{e} < e$.

5a. The signing validator must be a member of the validator set.

If all requirements are satisfied, then the sending validator's deposit counts as preparing $\vec{c} \to c$.

Requirements for accepting a commit message (Figure 1b):

1. Hash c must be the checkpoint of epoch e. Equivalently, $c = b_{100e+99}$.
2. Hash c must be *already* Justified. Meaning checkpoint c must have been Justified before the commit message. Equivalently, $c \in \mathbf{J}$.
3. Hash c must have been Justified within the past 100 blocks. Equivalently, if checkpoint c became Justified at block b_i, then the commits will only be accepted between blocks $[b_{i+1}, b_{i+101}]$.

3a. The signing validator must be a member of the validator set.

If all requirements are satisfied, then the sending validator's deposit counts as behind commiting checkpoint c.

Validators only recognize prepares and commits that have been included in blocks (even if those blocks are not part of the main chain).

The most notable property of Casper is that it is impossible for two conflicting checkpoints to be Finalized unless $\geq \frac{1}{3}$ of the validators violated one of the two[2] Casper Commandments (a.k.a. slashing conditions). These are:

I. A VALIDATOR SHALT NOT PUBLISH NONIDENTICAL PREPARES SPECIFYING THE SAME TARGET EPOCH.

In other words, for each epoch e, a validator may prepare at most exactly one $\left(\overrightarrow{c}, \overrightarrow{e}, c\right)$ triplet.

II. A VALIDATOR SHALT NOT COMMIT TO ANY EPOCH BETWEEN THE EPOCHS OF ITS OWN PREPARE STATEMENTS.

Equivalently, a validator may never publish,

$$\left\langle \textbf{PREPARE}, \overrightarrow{c}, \overrightarrow{e}, c_p, e_p \right\rangle \qquad \text{and} \qquad \left\langle \textbf{COMMIT}, e_c, c_c \right\rangle \ , \tag{4}$$

where the epochs satisfy $\overrightarrow{e} < e_c < e_p$.

If a validator violates any commandment, the evidence that the validator did this can be included into the blockchain as a transaction, at which point the validator's entire deposit will be taken away, with a 4% "finder's fee" given to the submitter of the evidence transaction.

2.1. Casper's Fork Choice Rule

Casper is more complicated than standard PoW designs. As such, the fork-choice must be adjusted. This fork-choice rule should be followed by all users, validators, and even the underlying block proposal mechanism. If the users, validators, or block-proposers instead follow the typical fork-choice rule of "always build atop the longest chain", there are pathological scenarios where Casper gets "stuck" and any blocks built atop the longest chain cannot be Finalized (or even Justified) without violating a Commandment—one such pathological scenario is the fork in Figure 2.

In Figure 2a, proposers follow the "build atop the longest chain" rule and proposers build atop A_2. But this runs into problems. A_2 cannot be Justified by Preparing $A_1 \rightarrow A_2$ because that requires first Justifying A_1. Unfortunately, because another block with the same epoch (B_1) is already Justified (has $\frac{2}{3}$ prepares), A_1 cannot be Justified without at least $\frac{1}{3}$ of validators violating Commandment **I** by preparing both A_1 and B_1.

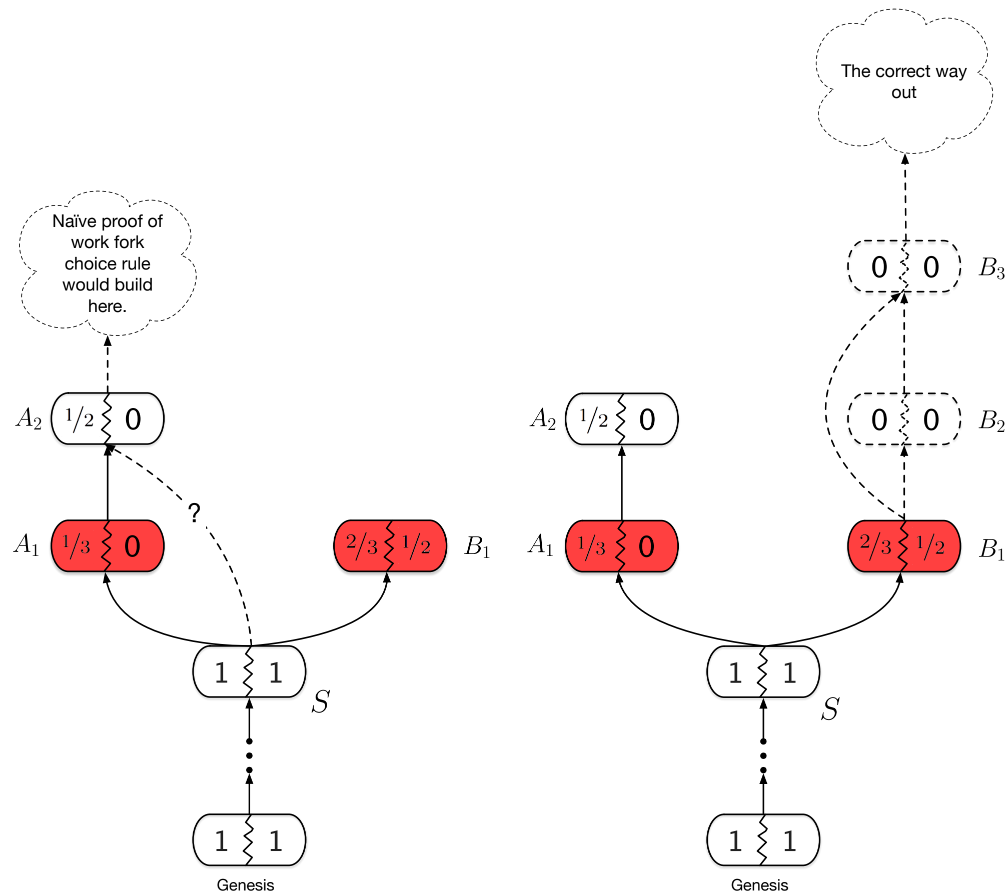
Since we can't justify A_1, could we instead Justify A_2 by preparing $S \rightarrow A_2$? Unfortunately, preparing $S \rightarrow A_2$ to Justification requires some validators to violate Commandment **II** as $\frac{1}{2}$ have already commited to B_1. This same argument applies to any ancestor of S. Therefore A_2 cannot be Justified by any means without violating a Commandment. Then because A_2 cannot be Justified, it cannot be Finalized. Even more unfortunate, this same argument also applies to all successors of A_2. Ergo, if validators are Rational and follow the longest-chain fork-choice rule, *no checkpoints would be justified or finalized ever again*.

Fortunately, there's another way; in Figure 2b we see the way out. Say we instead built atop B_1 to create B_2. Unfortunately B_2 cannot be Justified because preparing $B_1 \rightarrow B_2$ to justification requires some validators to violate Commandment **I** as $\frac{1}{2}$ have already prepared to A_2 (and likewise cannot prepare $S \rightarrow B_2$ because $\frac{1}{2}$ have already committed to B_1). So there's no way to Justify B_2. Then because B_2 cannot be Justified, it cannot be Finalized. But say we keep building and create B_3. Validators *can* prepare $B_1 \rightarrow B_3$ without violating Commandment **I**. And secondly, because A_2 has no commits, all validators can prepare $B_1 \rightarrow B_3$ without violating Commandment **II**. Once B_3 is Justified, it can be Finalized without obstacles.

Inspired by the successful escape in Figure 2b, we define a novel fork choice rule.[3] Instead of blindly following the longest-chain, we:

[2] Earlier versions of Casper had four slashing conditions [6], but we can reduce to two because of the requirements that (i) Committed hashes must already be Justified, and (ii) prepare messages must point to an already Justified ancestor. These requirements ensure that blocks will not register commits or prepares that violate the other two slashing conditions, making them superfluous.

[3] Casper's commit-following fork choice rule is analogous to the "greedy heaviest observed subtree" (GHOST) rule for proof of work chains[7].



(a) Longest-chain forkchoice rule.

(b) Casper-aware forkchoice rule.

Figure 2: A pathological scenario demonstrating the outcome of two fork-choice rules. In (a) we see the result of the typical "build atop the longest chain" fork choice rule. In (b) we see the result of our Casper-aware fork choice rule (Listing 3). Each pill represents a checkpoint. In each pill, the left number is the proportion of validators who prepared that checkpoint, and the right number is the proportion of validators who committed that checkpoint. Red pills are conflicting checkpoints.

1. Favor checkpoints by the proportion of commits.
2. When multiple checkpoints have the same proportion of commits, favor checkpoints by the proportion of prepares.
3. When multiple checkpoints have the same proportion of prepares, favor checkpoints by depth (longest chain).
4. When multiple checkpoints have the same depth, choose randomly.

A complete specification for our forkchoice rule is in Listing 3. The Casper fork-choice rule ensures that the selected head will always be one that minimizes validator penalties. In fact, as long as $> \frac{2}{3}$ of validators only commit to Justified checkpoints, there *never* needs to be a violation of either Commandment (see Theorem 2).

3. Proofs of Safety and Plausible Liveness

We prove Casper's two fundamental properties: *accountable safety* and *plausible liveness*. Accountable safety means that two conflicting checkpoints cannot be Finalized unless $\geq \frac{1}{3}$ of validators violate a slashing condition (meaning at least one third of the total deposit is lost). Plausible liveness means that, regardless of any previous events, it is always possible for $\frac{2}{3}$ of honest validators to finalize a new checkpoint.

Theorem 1 (Accountable Safety)**.** *Two conflicting checkpoints cannot be Finalized unless* $\geq \frac{1}{3}$ *of validators violate a slashing condition.*

Proof. Suppose the two conflicting checkpoints are A in epoch e_A and B in epoch e_B (see Figure 3). If both are Finalized, this entails $\frac{2}{3}$ commits and $\frac{2}{3}$ prepares in epochs e_A and e_B. In the trivial case where $e_A = e_B$ (Figure 3a), this implies that some intersection of $\geq \frac{1}{3}$ of validators must have violated Commandment **I**. If instead $e_A \neq e_B$, there exist two chains $G < \cdots < e_A^2 < e_A^1 < e_A$ and $G < \cdots < e_B^2 < e_B^1 < e_B$ of Justified checkpoints, both terminating at the genesis block G. Without loss of generality, suppose $e_A < e_B$. Then, there must be some e_B^i such that $e_B^i \leq e_A < e_B$. If $e_B^i = e_A$ (Figure 3b), then checkpoints A and B^i both have $\frac{2}{3}$ prepares, thus $\geq \frac{1}{3}$ of validators violated Commandment **I**. If instead $e_B^i < e_A$ (Figure 3c), checkpoint A has at least $\frac{2}{3}$ commits and checkpoint e_B^i has at least $\frac{2}{3}$ prepares with $e_B^i < e_A < e_B$. Therefore $\geq \frac{1}{3}$ of validators violated Commandment **II**. □

Theorem 2 (Plausible Liveness)**.** *Regardless of what previous events took place, if* $\geq \frac{2}{3}$ *of validators never commit to an Unjustified block, a new checkpoint can be finalized without violating any Commandment.*

Proof. Suppose all validators have sent some sequence of prepare and commit messages. Let c_j at epoch e_j be the highest-epoch Justified checkpoint. From the premise, $\geq \frac{2}{3}$ of validators have not committed to any epoch after e_j. Hence, Commandment **II** doesn't stop these validators from preparing any checkpoint c at epoch e where $e > e_j$ and then committing c. More concretely, these validators will always be able to publish $\langle \textbf{PREPARE}, c_j, e_j, c, e \rangle$ and then publish $\langle \textbf{COMMIT}, c, e \rangle$ without violating any Commandment.

□

4. Enabling Dynamic Validator Sets

The set of validators needs to be able to change. New validators must be able to join, and existing validators must be able to leave. To accomplish this, we define a variable in the state called the *dynasty counter*. [We don't actually need this in the state. It can be computed by each validator.] When a would-be valdiator's deposit message is included in dynasty d, then the validator will be *inducted* at the start of dynasty $d + 2$. We call $d + 2$ this validator's *start dynasty*.

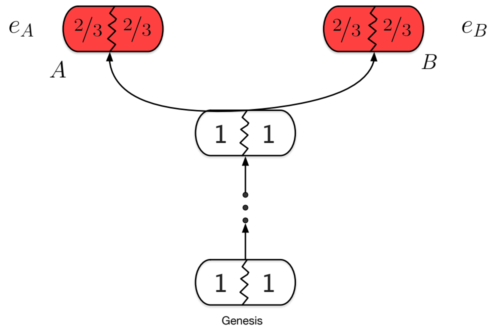
The dynasty counter increments if and only if there's been a *perfectly finalized checkpoint*. We define checkpoint C_n as "perfectly finalized" if and only if during epoch $n - 1$ every validator prepares $C_{n-1} \to C_n$ and commits C_n. For example, checkpoint C_3 (a.k.a., b_{199}) is perfectly finalized if during epoch 2, blocks $200 \ldots 299$ (before block 300), *all* validators prepare $b_{99} \to b_{199}$ *and* commit b_{199}.

We define a growing sequence of the perfectly finalized checkpoints $\mathbf{P}$ starting simply as, $\mathbf{P} = (G)$. Then, anytime a checkpoint is perfectly finalized, we append that checkpoint to sequence $\mathbf{P}$. So if checkpoints C_2, C_5, C_6, C_9 are perfectly finalized, $\mathbf{P}$ becomes, $\mathbf{P} = (G, C_2, C_5, C_6, C_9)$, and so on for future perfectly finalized checkpoints. From the list of perfectly finalized checkpoints, we can compute the dynasty counter, $d \equiv |\mathbf{P}|$, and $\mathcal{D}(k)$ (where $k \in [1, |\mathbf{P}| - 1]$) as the set of checkpoints within dynasty k,

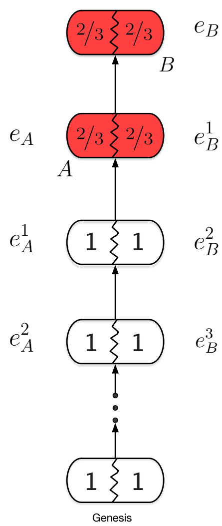
$$\mathcal{D}(k) \equiv \begin{cases} \{c \in \mathbf{C} : \mathbf{P}_k < c\} & \text{if } k = |\mathbf{P}| - 1, \\ \{c \in \mathbf{C} : \mathbf{P}_k < c \leq \mathbf{P}_{k+1}\} & \text{otherwise.} \end{cases} \tag{5}$$

So using the example above, $\mathcal{D}(1) = \{C_1, C_2\}$, $\mathcal{D}(2) = \{C_3, C_4, C_5\}$ and $\mathcal{D}(2) = \{C_6\}$, $\mathcal{D}(3) = \{C_7, C_8, C_9\}$, etc.

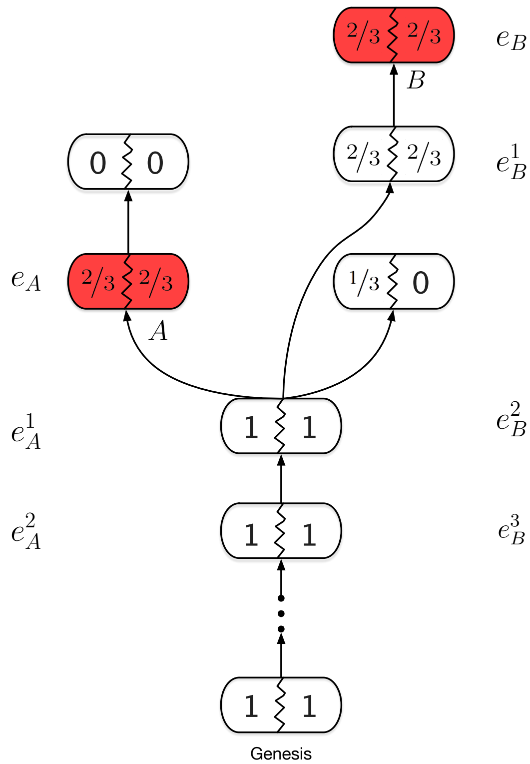
To leave the validator set, the validator must send a "withdraw" message. If their withdraw message gets included during dynasty d, the validator similarly leaves the validator set at the start of dynasty



(a) $e_A = e_B$



(b) $e_B^i = e_A < e_B$



(c) $e_B^i < e_A < e_B$

Figure 3: Illustrating the three scenarios in Theorem 1. Each pill represents a checkpoint. In each pill, the left number is the proportion of validators who prepared that checkpoint, and the right number is the proportion of validators who committed that checkpoint. Red pills are conflicting checkpoints.

$d+2$; we call $d+2$ the validator's *end dynasty*. At the start of the end dynasty, the validator's deposit is locked for a long period of time, the *withdrawal delay*, (think "four months") before the deposit can be withdrawn. If, during the withdrawal delay, the validator violates any Commandment, the deposit is forfeited.

$$\mathcal{V}(k) = \{\text{validator set for dynasty } k\text{. The accepted validators for checkpoints } \mathcal{D}(k).\} \ . \qquad (6)$$

Which leads to the pleasing relations, $\mathbf{P} \subseteq \mathbf{F} \subseteq \mathbf{J} \subseteq \mathbf{C} \subset \mathbf{B}$ and $\mathcal{D}(k) \subseteq \mathbf{C} \ \ \forall k$.

Revised requirement for accepting a prepare message (Figure 1a):

5b. The signing validator must be a member of the validator set for a specified dynasty.

Revised requirement for accepting a commit message (Figure 1b):

3b. The signing validator must be a member of the validator set for a specified dynasty.

For a checkpoint to be Justified, it must be prepared by a set of validators which contains (i) at least $\frac{2}{3}$ of the current dynasty and (ii) at least $\frac{2}{3}$ of validators comprising the previous dynasty. Finalization works the same. The current and previous dynasties will usually greatly overlap; but if the two validator sets substantially differ, this "stitching" mechanism mitigates the risk of a finality reversion or other failure can happen because different messages are signed by different validator sets and so equivocation is avoided.

We can write this mathematically by rewriting eq. (3) to become,

$$\begin{aligned}\mathbf{J} &= (c \in \mathbf{C} : \min\left[\text{valid\_prepares}(c, d-1), \text{valid\_prepares}(c, d)\right] \geq 2/3) \\ \mathbf{F} &= (j \in \mathbf{J}\ : \min\left[\text{valid\_commits}(j, d-1), \text{valid\_commits}(j, d)\right] \geq 2/3)\ ,\end{aligned} \tag{7}$$

where d is the current dynasty index.

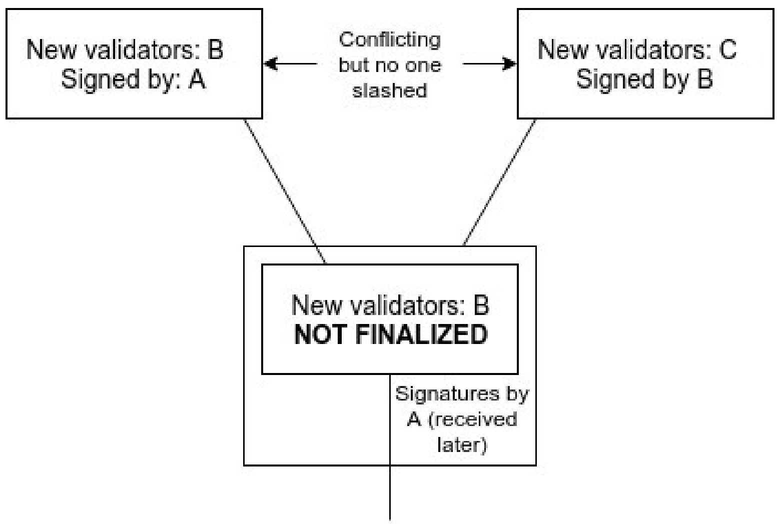


Figure 4: Without the validator set stitching mechanism, it's possible for two conflicting checkpoints to be Finalized with no validators slashed

not happy with this notation yet.

4.1. Long Range Attacks

Note that the withdrawal delay introduces a synchronicity assumption *between validators and clients*. Because validators can withdraw their deposits after the withdrawal delay, there is an attack where a coalition of validators which had more than $\frac{2}{3}$ of deposits *long ago in the past* withdraws their deposits, and then uses their historical deposits to finalize a new chain that conflicts with the original chain without fear of getting slashed. Despite violating slashing conditions to make a chain split, because the attacker has already withdrawn on both chains they do not lose any money. This is called the *long-range atack*.

We solve this problem by simply having clients not accept a Finalized checkpoint that conflicts with Finalized checkpoints that they already know about. Suppose that clients can be relied on to log on at least once every δ days, and the withdrawal delay is W. Suppose attackers send one Finalized checkpoint at time 0, and then another right after. In the worst case, the first checkpoint arrives at all clients at time 0, and that the second reaches a client at time δ. The client will then know of the fraud, and will be able to create and publish an evidence transaction. We then add a consensus rule that requires clients to reject chains that do not include evidence transactions that the client has known about for time δ. Hence, clients will not accept a chain that has not included the evidence transaction within time $2 * \delta$. So if $W > 2 * \delta$ then slashing conditions are enforcible.

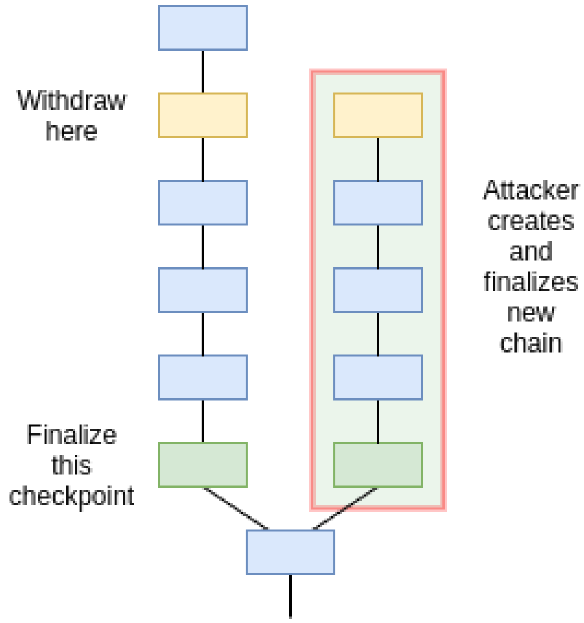


Figure 5: Illustrating the long-range attack.

In practice, this means that if the withdrawal delay is four months, then clients will need to log on at least once every two months to avoid accepting bad chains for which attackers cannot be penalized. [convert this into a lemma/theorem.]

5. Recovering From Castastrophic Crashes

Suppose that $> \frac{1}{3}$ of validators crash-fail at the same time—i.e, they are no longer connected to the network due to a network partition, computer failure, or are malicious actors. Then, no later checkpoint will be able to get Finalized.

We can recover from this by instituting a "leak" which dissipates the deposits of validators that do not prepare or commit, until eventually their deposit sizes decrease low enough that the validators that *are* preparing and committing are a $\frac{2}{3}$ supermajority. The simplest possible formula is something like "validators with deposit size D lose $D * p$ in every epoch in which they do not prepare and commit", though to resolve catastrophic crashes more quickly a formula which increases the rate of dissipation in the event of a long streak of non-Finalized blocks may be optimal.

The dissipated portion of deposits can either be burned or simply forcibly withdrawn and immediately refunded to the validator; which of the two strategies to use, or what combination, is an economic incentive concern and thus outside the scope of this paper.

Note that this does introduce the possibility of two conflicting checkpoints being Finalized, with validators only losing money on one of the two checkpoints as seen in Figure 6.

If the goal is simply to achieve maximally close to 50% fault tolerance, then clients should simply favor the Finalized checkpoint that they received earlier. However, if clients are also interested in defeating 51% censorship attacks, then they may want to at least sometimes choose the minority chain. All forms of "51% attacks" can thus be resolved fairly cleanly via "user-activated soft forks" that reject what would normally be the dominant chain. Particularly, note that finalizing even one block on the dominant chain precludes the attacking validators from preparing on the minority chain because of Commandment II, at least until their balances decrease to the point where the minority can commit, so such a fork would also serve the function of costing the majority attacker a very large portion of their deposits.

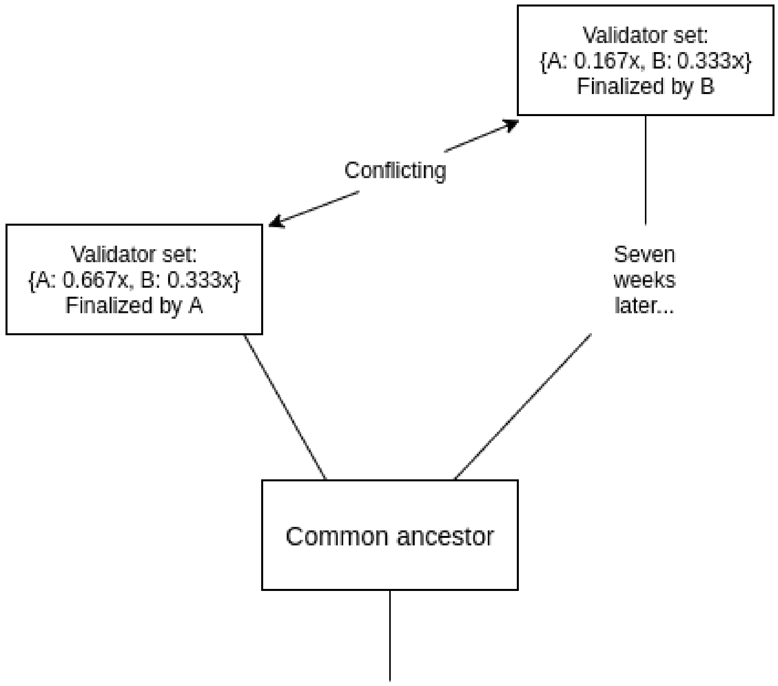


Figure 6: The checkpoint on the left can be Finalized immediately. The checkpoint on the right can be Finalized after some time, once offline validator deposits sufficiently dissipate.

6. Conclusions

This introduces the basic workings of Casper the Friendly Finality Gadget's prepare and commit mechanism and fork choice rule, in the context of Byzantine fault tolerance analysis. Separate papers will serve the role of explaining and analyzing incentives inside of Casper, and the different ways that they can be parametrized and the consequences of these paramtrizations.

Future Work. [fill me in]

Acknowledgements. We thank Virgil Griffith for review.

References

[1] King, S. & Nadal, S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake **19** (2012). URL `https://decred.org/research/king2012.pdf`.

[2] Vasin, P. Blackcoin's proof-of-stake protocol v2 (2014). URL `http://blackcoin.co/blackcoin-pos-protocol-v2-whitepaper.pdf`.

[3] Bentov, I., Gabizon, A. & Mizrahi, A. Cryptocurrencies without proof of work. In Sion, R. (ed.) *International Conference on Financial Cryptography and Data Security*, 142–157 (Springer, 2016).

[4] Castro, M., Liskov, B. & et. al. Practical byzantine fault tolerance. In Leach, P. J. & Seltzer, M. (eds.) *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, vol. 99, 173–186 (1999).

[5] Kwon, J. Tendermint: Consensus without mining (2014). URL `https://tendermint.com/static/docs/tendermint.pdf`.

[6] Buterin, V. Minimal slashing conditions (2017). URL `https://medium.com/@VitalikButerin/minimal-slashing-conditions-20f0b500fc6c`.

[7] Sompolinsky, Y. & Zohar, A. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. **2013** (2013).

A. Code For Valid_prepares and Valid_commits

```
def valid_prepares(checkpoint_source, epoch_source, checkpoint, epoch, d):

    # code goes here

    return z
```

Listing 1: Algorithm for counting up the prepare portion behind a checkpoint. [fill this in]

```
def valid_commits(checkpoint, epoch, d):

    # code goes here

    return z
```

Listing 2: Algorithm for counting up the commit portion behind a checkpoint. [fill this in]

B. Unused Text

[This is where text goes that for which a home hasn't been found yet. If no home is found, it will be deleted.]

We define an *epoch* as a range of 100 blocks (e.g., blocks 600...699 are epoch 6), and a *checkpoint* of an epoch is the final block in that epoch.

The proposal mechanism will initially be the existing Ethereum proof of work chain, making the first version of Casper a *hybrid PoW/PoS algorithm* that relies on proof of work for liveness but not safety, but in future versions the proposal mechanism can be substituted with something else.

for the same e and c as in eq. (4). The c is the block hash of the block at the start of the epoch. A hash c being justified entails that all fresh (non-finalized) ancestor blocks are also justified. A hash c being finalized entails that all ancestor blocks are also finalized, regardless of whether they were previously fresh or justified. An "ideal execution" of the protocol is one where, at the start of every epoch, every validator Prepares and Commits the first blockhash of each epoch, specifying the same $\vec{e}$ and $\vec{c}$.

In the Casper protocol, there exists a set of validators, and in each *epoch* (see below) validators may send two kinds of messages:

$$[PREPARE, epoch, hash, epoch_{source}, hash_{source}]$$

and

$$[COMMIT, epoch, hash]$$

If, during an epoch e, for some specific ancestry hash h, for any specific ($epoch_{source}, hash_{source}$ pair), there exist $\frac{2}{3}$ prepares of the form

$$[PREPARE, e, h, epoch_{source}, hash_{source}]$$

, then h is considered *justified*. If $\frac{2}{3}$ commits are sent of the form

$$[COMMIT, e, h]$$

then h is considered *finalized*.

During epoch n, validators are expected to send prepare and commit messages with $e = n$ and h equal to a checkpoint of epoch n. Prepare messages may specify as $\vec{c}$ a checkpoint for any previous epoch (preferably the preceding checkpoint) of c, and which is *justified* (see below), and the $\vec{e}$ is expected to be the epoch of that checkpoint.

Honest validators will never violate slashing conditions, so this implies the usual Byzantine fault tolerance safety property, but expressing this in terms of slashing conditions means that we are actually proving a stronger claim: if two conflicting checkpoints get finalized, then at least $\frac{1}{3}$ of validators were malicious, *and we know whom to blame, and so we can maximally penalize them in order to make such faults expensive*.

This simplifies our finalty mechanism because it allows it to be expressed as a fork choice rule where the "score" of a block only depends on the block and its children, putting it into a similar category as more traditional PoW-based fork choice rules such as the longest chain rule and GHOST[7].

Unlike GHOST, however, this fork choice rule is also *finality-bearing*: there exists a "finality" mechanism that has the property that (i) the fork choice rule always prefers Finalized blocks over non-Finalized blocks, and

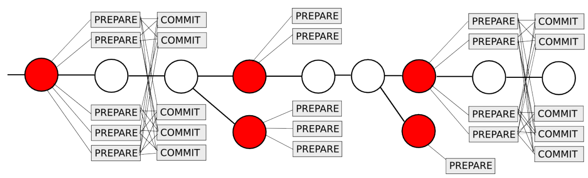
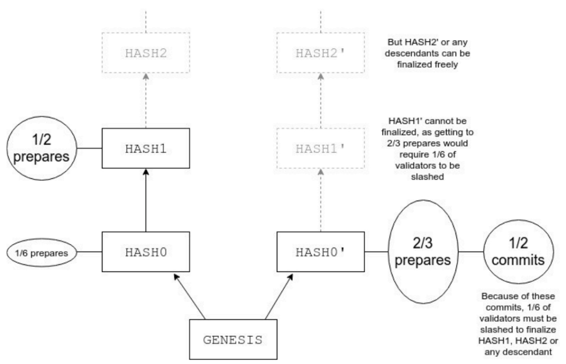


Figure 7: Illustrating prepares, commits and checkpoints. Arrows represent *dependency* (e.g., a commit depends on there being $\frac{2}{3}$ existing prepares).



1. Start with HEAD equal to the genesis of the chain.
2. Select the descendant checkpoint of HEAD with the most commits (only Justified checkpoints are admissible)
3. Repeat (2) until no descendant with commits exists.
4. Choose the longest proof of work chain from there.

The mechanism described above ensures *plausible liveness*; however, it by itself does not ensure *actual liveness*—that is, while the mechanism cannot get stuck in the strict sense, it could still enter a scenario where the proposal mechanism (e.g., the proof of work chain) gets into a state where it never ends up creating a checkpoint that could get Finalized.

The symmetry is as follows. In GHOST, a node starts with the head at the genesis, then begins to move forward down the chain, and if it encounters a block with multiple children then it chooses the child that has the larger quantity of work built on top of it (including the child block itself and its descendants).

In this algorithm, we follow a similar approach, except we repeatedly seek the child that comes the closest to achieving finality. Commits on a descendant are implicitly commits on all of its lineage, and so if a given descendant of a given block has more commits than any other descendant, then we know that all children along the chain from the head to this descendant are closer to finality than any of their siblings; hence, looking for the *descendant* with the most commits and not just the *child* replicates the GHOST principle most faithfully.

(that is, the checkpoint of epoch e must be Finalized during epoch e, and the chain must learn about this before epoch e ends). In simpler terms, when a user sends a "deposit" transaction, they need to wait for the transaction to be perfectly Finalized, and then they need to wait again for the next epoch to be Finalized; after this, they become part of the validator set.

In fact, when *any* checkpoint gets $k > \frac{1}{3}$ commits, no conflicting checkpoint can get Finalized without $k - \frac{1}{3}$ of validators getting slashed.

C. Full Fork Choice Rule

```
from random import shuffle

def get_head(genesisblock):

    head = genesisblock

    while True:
        S = successors( head )

        if not S:
            return head

        # choose the successor with the greatest commits
        max_commit = max( map(valid_commits, S) )
        S = [ s for s in S if valid_commits(S) == max_commit ]

        if len(S) == 1:
            head = S[0]
            continue

        # choose the succesor with the greatest prepares
        max_prepare = max( map(valid_prepares, S) )
        S = [ s for s in S if valid_prepares(S) == max_prepare ]

        if len(S) == 1:
            head = S[0]
            continue

        # choose the succesor with the greatest depth (longest chain)
        max_depth = max( map(depth, S) )
        S = [ s for s in S if depth(S) == max_depth ]

        if len(S) == 1:
            head = S[0]
            continue

        # choose a random successor
        shuffle(S)
        head = S.pop()
```

Listing 3: Algorithm for determining the head

D. Notes To Authors

D.1. Questions

- If there are Prepares for the same hash+epoch pair from multiple sources, do they add? Or do we simply take the max?
- If checkpoint X is Justified, can it be Finalized without violating any Commandment?
- [fill me in!]

D.2. Notes On Suggested Terminology

- parent $\rightarrow$ predecessor.
- child $\rightarrow$ successor (unless want to emphasize there can be multiple candidate successors)
- ancestors $\rightarrow$ lineage
- to refer to the set of $\{$ predecessor, successor $\}$ $\rightarrow$ adjacent

D.3. Todo

- [Reference the various Figures within the text so we more easily know what goes with what.]
- [fill me in]
- Try checkpoint figures without arrows.

In the other way...