Casper the Friendly Finality Gadget

Vitalik Buterin and **Virgil Griffith**
Ethereum Foundation

Abstract

We introduce Casper, a parsimonious proof-of-stake-based finality system which overlays on an existing proof of work blockchain. Casper is a partial consensus mechanism combining proof of stake algorithm research and Byzantine fault tolerant consensus theory. We introduce our system, prove some desirable properties, and show defenses against long range revisions and catastrophic crashes.

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") ze holds. This is a vastly more efficient alternative to proof of work "mining' and enables 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 and features a chain of blocks and 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 blocks. Repurposing BFT algorithms for proof of stake was first introduced by Tendermint[5]. Casper follows this BFT tradition, though with some modifications.

1.1. Our Work

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 liveness depends on the chosen proposal mechanism. That is, if attackers wholly control the proposal mechanism, Casper protects against finalizing two conflicting checkpoints, however, the attackers could prevent Casper from finalizing any future checkpoints.

Casper introduces several new properties that BFT algorithms do not necessarily support:

- We add *accountability*. If a validator violates the rules, we can detect the violation and know which validator violated the rule. Accountability allows us to penalize malfeasant validators, solving the *nothing at stake* problem that plagues chain-based PoS. The penalty for violating a rule is a validator's entire deposit. This maximal penalty is the defense against violating the protocol. Because proof of stake security is based on the size of the penalty, which usually exceeds the gains from the mining reward, proof of stake provides *stronger* security incentives than proof of work.

- We introduce a safe way for the validator set to change over time (Section 4).
- We introduce defenses against long range revision attacks as well as attacks where more than $\frac{1}{3}$ of validators drop offline, at the cost of a very weak *tradeoff synchronicity assumption* (Section 5).
- Casper's design 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: the fork choice rule (Section 3), validator set changes (Section 4), and finally defense against attacks (Section 5).

2. The Casper Protocol

In the simple version, we assume there is a fixed 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,

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

Within Ethereum, the proposal mechanism will initially be the existing proof of work chain, making the first version of Casper a *hybrid PoW/PoS system* which relies on proof of work for liveness yet relies on proof of stake for safety. In future versions the PoW proposal mechanism will be replaced with something else.

An *epoch* is defined as a contiguous sequence of 100 blocks terminating at a checkpoint. The *epoch of a block* is the index of the epoch containing that block, 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 zer deposit is the number of deposited coins. From there on each validator's deposit rises and falls with rewards and penalties. Proof of stake's security derives from the size of the deposits, not the number of validators, so 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 $\frac{2}{3}$ of the total deposit size of the entire set 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.

A prepare message $\left\langle \textbf{PREPARE}, \vec{c}, \vec{e}, c, e \right\rangle$ from a validator is valid if and only if:

1. Hash $\vec{c}$ must be the checkpoint for epoch $\vec{e}$. $c = b_{100\vec{e}+99}$
2. Hash $\vec{c}$ must be *already* Justified. Meaning checkpoint $\vec{c}$ must have been Justified before the prepare message. Equivalently, $\vec{c} \in \mathbf{J}$.
3. Hash c must be the checkpoint for epoch e. Equivalently, $c = b_{100e+99}$.
4. Epoch $\vec{e} < e$.
5. The signing validator must be a member of the validator set.

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

Notation	Description
$\overrightarrow{c}$	the hash of any Justified checkpoint
$\overrightarrow{e}$	the epoch of checkpoint $\overrightarrow{c}$
c	any checkpoint hash after $\overrightarrow{c}$
e	the epoch of checkpoint c
$\mathcal{S}$	signature of $\left\langle \textbf{PREPARE}, \overrightarrow{c}, \overrightarrow{e}, c, e \right\rangle$ 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.

A commit message $\langle \textbf{COMMIT}, e, c \rangle$ from a validator is valid if and only if:

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 $[b_{i+1}, b_{i+101}]$.
4. 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 $\overrightarrow{c} \to c$ or committing commiting checkpoint c. If there are prepares for the same target checkpoint coming from multiple sources, the prepared portion for the target is simply the maximum (not the sum!) over all sources. For penalties, validators recognize all published prepares and commits including invalid prepares/commits or prepares/commits that, for whatever reason, did not make it into the chain.

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

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

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

The most notable property of Casper is that it is impossible for two conflicting checkpoints to be Finalized without $\geq \frac{1}{3}$ of the validators violating 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.

[2]Earlier versions of Casper had four slashing conditions [6], but we have reduced to two slashing conditions. We removed the conditions: (i) Committed hashes must already be Justified, and (ii) prepare messages must point to an already Justified ancestor. This is a design choice. And we made this choice so that violations are not dependent on the state of the chain.

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

Equivalently, a validator may never publish,

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

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

If a validator violates any commandment, the evidence of the violation 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. Reliable enforcement of Casper's commandments rests on the block proposal mechanism, which is outside the scope of this paper. However, in current ethereum, stopping the enforcement of slashing conditions requires $> 50\%$ of the proof-of-work hashing power.

2.1. Proving 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, if $\geq \frac{2}{3}$ of valdiators follow the protocol, then it's always possible to finalize a new checkpoint without any validator violating a commandment.

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 2). 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 2a), 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 2b), 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 2c), 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**. □

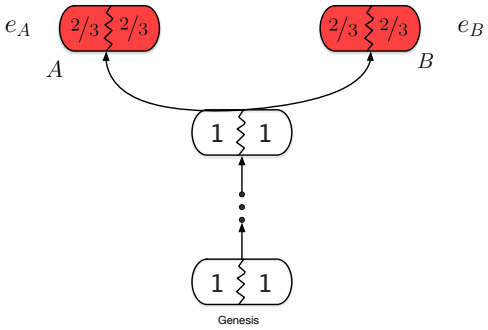
Theorem 2 (Plausible Liveness)**.** *Regardless of what previous events took place, if $\geq \frac{2}{3}$ of validators never commit to an Unjustified block, a new checkpoint can be finalized without violating any Commandment.*

Proof. Suppose all validators have sent some arbitrary 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.

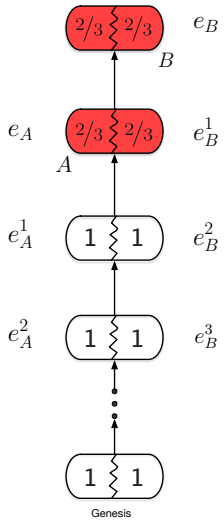
□

3. Casper's Fork Choice Rule

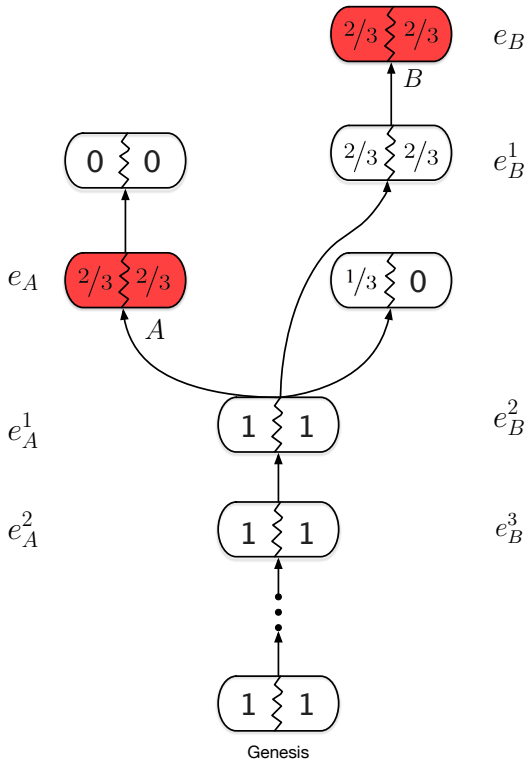
Casper is more complicated than standard PoW designs. As such, the fork-choice must be adjusted. This fork-choice rule should be followed by all users, validators, and even the underlying block proposal mechanism. If the users, validators, or block-proposers instead follow the typical fork-choice rule 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 3.



(a) $e_A = e_B$



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



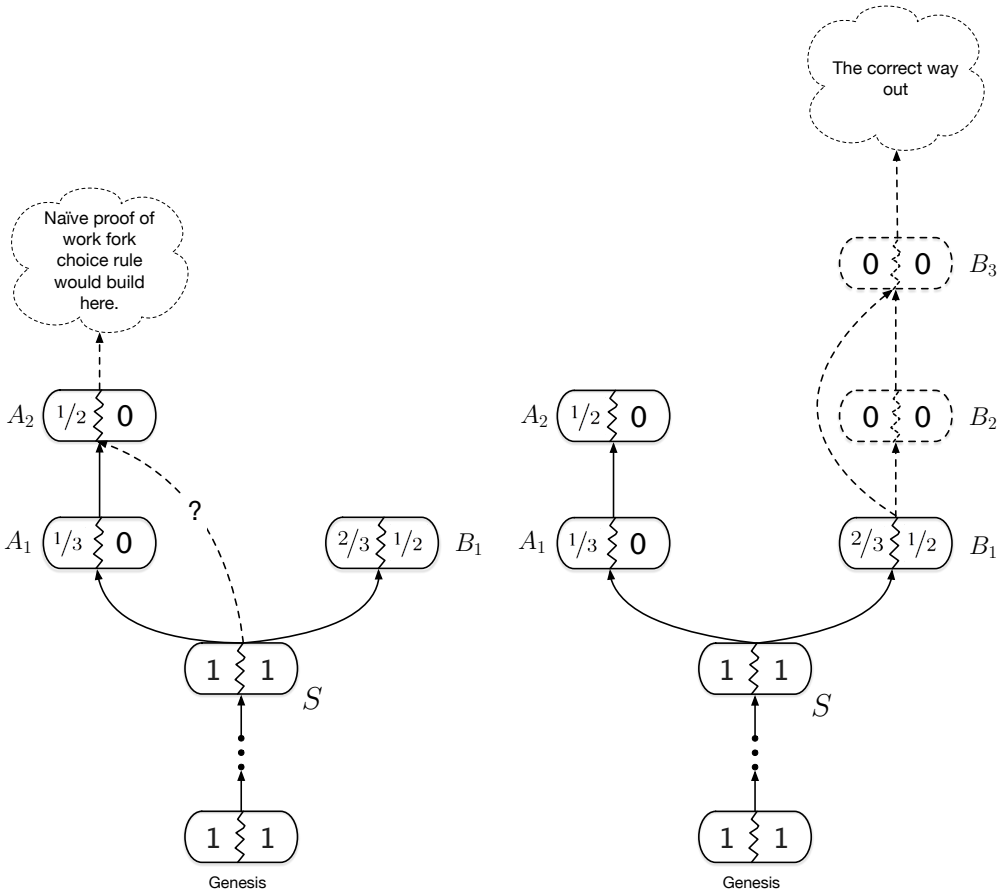
(c) $e_B^i < e_A < e_B$

Figure 2: Illustrating the three scenarios in Theorem 1. Each pill represents a checkpoint. In each pill, the left/right number is the proportion of validators who prepared/committed that checkpoint.

In Figure 3a, 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 3b 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.



(a) Longest-chain forkchoice rule.

(b) Casper-aware forkchoice rule.

Figure 3: 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.

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

[3]Casper's commit-following fork choice rule is analogous to the "greedy heaviest observed subtree" (GHOST) rule for proof of work chains[7]. 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 closest to achieving finality. Commits on a descendant are implicitly commits on its ancestors, and so if a given descendant of a given block has the most commits, 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* gives the right properties. Finalizing a checkpoint requires $\frac{2}{3}$ commits

1. Favor checkpoints by the proportion of commits.
2. Then, if multiple checkpoints have the same proportion of commits, favor checkpoints by the proportion of prepares.
3. Then, if multiple checkpoints have the same proportion of prepares, favor checkpoints by depth (longest chain).
4. Then, if 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 $\geq \frac{2}{3}$ of validators only commit to Justified checkpoints, there *never* needs to be a violation of either Commandment (see Theorem 2). [It remains ambiguous to me whether we look at the whole descening lineage or just the immediate child.]

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 $C_{n-1} \rightarrow C_n$ 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 $b_{99} \rightarrow b_{199}$ *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, $\mathbf{P} = (G, C_2, C_5, C_6, C_9)$, and so on for future perfectly finalized checkpoints. Then, the checkpoints in a given dynasty k are,

$$\mathcal{D}(k) \equiv \{c \in \mathbf{C} : \mathbf{P}_k < c \leq \mathbf{P}_{k+1}\} \ , \tag{5}$$

where $1 \leq k \leq |\mathbf{P}| - 2$. 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. This leads to the pleasing relation, $\mathbf{P} \subseteq \mathbf{F} \subseteq \mathbf{J} \subseteq \mathbf{C} \subset \mathbf{B}$.

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 $d + 2$; 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 is withdrawn. If, during the withdrawal delay, the validator violates any Commandment, the deposit is forfeited.

To support dynamic validator sets, we add the condition that 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 when different messages are signed by different validator sets. These modifications are in Figure 4.

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

$$\begin{aligned} \mathbf{J} &= (c \in \mathbf{C} : \min\left[\text{valid\_prepares}(c, d-1), \text{valid\_prepares}(c, d)\right] \geq 2/3) \\ \mathbf{F} &= (j \in \mathbf{J} \ : \min\left[\text{valid\_commits}(j, d-1), \text{valid\_commits}(j, d)\right] \geq 2/3) \ , \end{aligned} \tag{6}$$

where d is the current dynasty.

of a *single* checkpoint, and so unlike GHOST we simply look for the maximum commit count instead of trying to sum up all commits in an entire subtree.

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):

4b. The signing validator must be a member of the validator set for the specified dynasty.

Figure 4: The revised conditions for accepting a prepare or commit message.

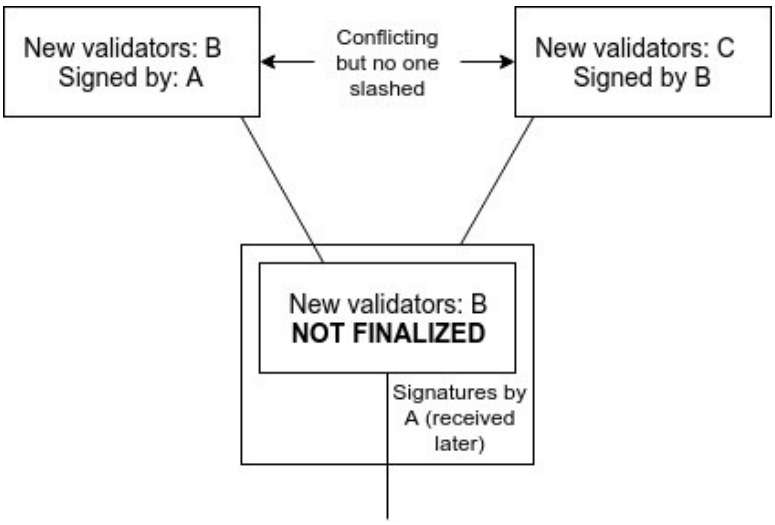


Figure 5: Without the validator set stitching mechanism, it's possible for two conflicting checkpoints to be Finalized with no validators slashed. [I don't get this. -Virgil]

5. Stopping Attacks

There are two well-known attacks against proof-of-stake systems: *long range revisions* and *catastrophic crashes*. We discuss each in turn.

5.1. Long Range Revisions

The withdrawal delay after a validator's end dynasty introduces a synchronicity assumption *between validators and clients*. Once a coalition of validators has withdrawn their deposits, if that coalition had more than $\frac{2}{3}$ of deposits *long ago in the past* can use its historical supermajority to Finalize conflicting checkpoints without fear of getting slashed (because they have already withdrawn their money). This is called the *long-range revision attack*.

Suppose that every client can be relied on to refresh its view of the blockchain every δ days. Then, if a client hears about two conflicting Finalized checkpoints, c at time t and c' at time t', there are two cases. If, from the point of view of all clients, $c < c'$, then all clients will accept c [we do mean c here right?] and there is no ambiguity. If on the other hand different clients see different orders (i.e., some see $c < c'$, others see $c > c'$), then it must be the case that for all clients, $|t - t'| \leq 4\delta$. If $\delta << \frac{\omega}{4}$, then there plenty of time during which slashing evidence can be included in both chains. [When we say 4δ, why 4?]

If a chain does not include slashing evidence, then clients must reject the misbehaving chain via user-activated soft fork[4] (see Section 5.2). In practice, this means that if the set the withdrawal delay $\omega = 120$ days, each client will need to refresh zer view of the chain at least once every 30

[4]Clients can sometimes automatically detect a long range revision attack. Explicitly, if a client sees a slashing condition violation on checkpoint c, and then after an "evidence inclusion timeout", the head *still* hasn't slashed the malfeasant validator, the client warns the user about a plausible attack.

days to avoid accepting blocks from Byzantine validators with nothing at stake (and thus cannot be penalized for fraudulent behavior).

Theorem 3 (Refresh Rate)**.** *If a user refreshes its view of the blockchain at least once every $\omega/4$ days, it will not be fooled a long range revision attack.*

Proof. [proof goes here.] □

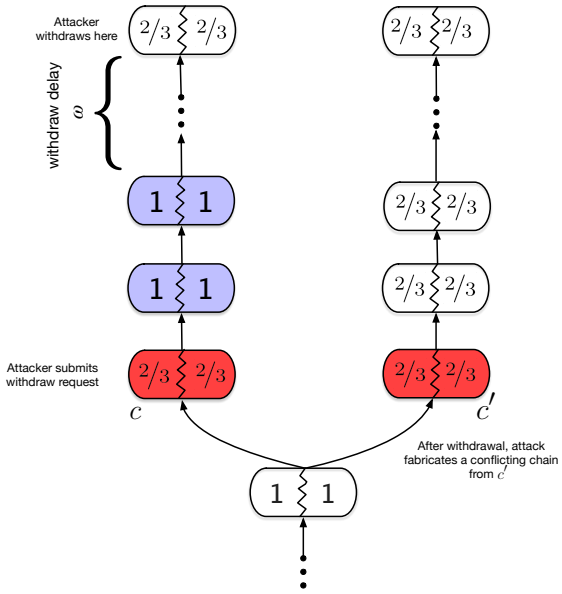


Figure 6: Illustrating the long-range revision attack. It's unclear whether to favor checkpoint c or c'. [Is this correct? It seems like taking the maximum over the commits will show the chain from checkpoint c is the better choice.]

5.2. 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 the validators themselves are malicious. Intuitively, from this point on, no future checkpoint can reach the $\frac{2}{3}$ commits to become Finalized.

We can recover from this by instituting an "inactivity leak" which slowly drains the deposit of any validator who does not prepare or commit checkpoints, until eventually zer deposit sizes decrease low enough that the validators who *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 fail to prepare or 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.

This drained ether can be burned or returned to the validator after ω days. The derivation of the exact formula for the inactivity leak is 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 only one of the two checkpoints as seen in Figure 7. [Don't know what this means.]

If the goal is simply to achieve maximally close to 50% fault tolerance, then each client should simply favor the Finalized checkpoint that it received first. 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 Commandment **II**

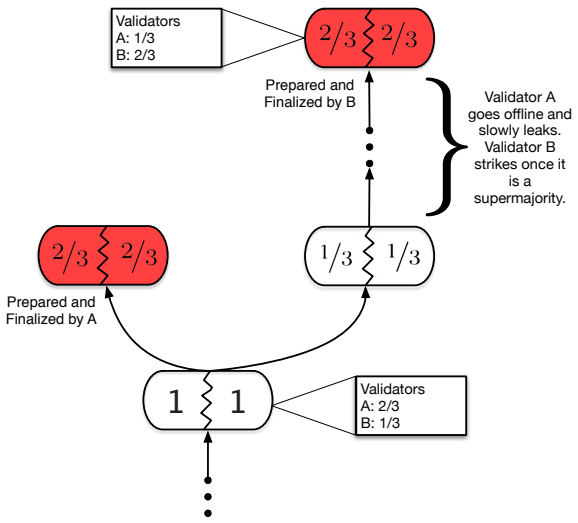


Figure 7: The checkpoint on the left can be Finalized immediately. The checkpoint on the right can be Finalized after some time, once offline validator deposits sufficiently deplete. [improve this figure]

precludes the attacking validators from Finalizing even one block on the dominant chain. [I don't know what this is trying to say.]

At least until their balances decrease to the point where the minority can commit, so such a fork would be very expensive to attackers. [better explain this.]

6. Conclusions

We presented Casper, a novel proof of stake system derived from the Byzantine fault tolerance literature. Casper includes: two slashing conditions, a novel fork choice rule based on [7], and dynamic validator sets. Finally we introduced two extensions to Casper to defend against two common attacks.

Future Work. The current Casper system builds upon a proof of work block proposal mechanism. We wish to conver the block proposal mechanism to proof of stake. Future papers will explain and analyze the financial incentives within Casper and their consequences.

Acknowledgements. We thank Vlad Zamfir for frequent discussions.

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

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.

$$\mathcal{D}(k) \equiv \begin{cases} \{c \in \mathbf{C} : \mathbf{P}_k < c\} & \text{if } k = |\mathbf{P}| - 1, \\ \{c \in \mathbf{C} : \mathbf{P}_k < c \leq \mathbf{P}_{k+1}\} & \text{otherwise.} \end{cases} \tag{7}$$

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.

$$\mathcal{V}(k) = \{\text{validator set for dynasty } k. \text{ The accepted validators for checkpoints } \mathcal{D}(k).\} \ . \tag{8}$$

- 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."

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 of a forked chain

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?
- In the fork-choice rule it remains ambiguous to me whether we're looking at the immediate successor or the whole lineage of successors.
- [fill me in!]

D.2. Notes On Suggested Terminology

- parent → predecessor.
- child → successor (unless want to emphasize there can be multiple candidate successors)
- ancestors → lineage

- to refer to the set of { predecessor, successor } $\rightarrow$ adjacent