Casper the Friendly Finality Gadget

Vitalik Buterin and Virgil Griffith
Ethereum Foundation

September 11, 2017

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

Over the past few years there has been considerable research into "proof of stake"-based blockchain consensus algorithms. In a proof of stake system, a blockchain grows and agrees on new blocks through a process where anyone who holds coins inside of the system can participate, and the amount of influence that any given coin holder has is proportional to the number of coins (or "stake") that they hold. This represents an alternative to proof of work "mining", allowing blockchains to operate without the high hardware and electricity costs that proof of work blockhains require.

There have been two major schools of thought in proof of stake algorithm design. The earlier of the two, *chain-based proof of stake*, tries to closely mirror the mechanics of proof of work, 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[**?**], Blackcoin[**?**] and Iddo Bentov's work[**?**].

The other school, *BFT-based proof of stake*, is based on a thirty year old body of research into *Byzantine fault tolerant consensus algorithms* such as PBFT [**?**]. BFT algorithms tend to have strong and rigorously proven mathematical properties; for example, one can usually mathematically prove that as long as at more than $\frac{2}{3}$ of participants in the protocol are following the protocol correctly, then the algorithm cannot possibly finalize conflicting block hashes at the same time ("safety"); this result holds regardless of network latency. The suggestion of repurposing BFT algorithms for proof of stake was first introduced by Tendermint[**?**].

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.

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.

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[] that often plagues chain-based PoS. The penalty is the malfeasant validator's

entire deposit. This maximum penalty provides a bulwark against violating the protocol by making violations immensely expensive. The economic robustness of protocol guarantees is much higher than the size of the rewards that the system pays out during normal operation. This provides a *much stronger* security guarantee than possible with proof of work.

- We introduce a provably safe way for the validator set to change over time.
- 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.*
- 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 5) and mass liveness fault recovery.

2 The Protocol

In the simple version, we assume there is a set of validators and a *proposal mechanism* which is any system that proposes blocks (such as a proof of work chain). Under normal operation, almost all of the blocks proposed by the proposal mechanism would form a chain, but if the proposal mechanism operation is faulty (in proof of work, this may arise due to high latency or 51% attacks) there may be multiple divergent chains being extended at the same time.

We order a chain of blockhashes into a sequence called a *blockchain* **B**. Within blockchain **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,

$$C_i = \begin{cases} b_0 & \text{if } i = 0, \\ b_{100*i-1} & \text{otherwise.} \end{cases} \tag{2}$$

C_0 is called the *genesis*. An *epoch* is defined as the contiguous sequence of blocks between two checkpoints, including the later checkpoint but not the earlier one. The *epoch of a block* is the index of the epoch containing that hash, e.g., the epoch of block 599 is 5. [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:

$$\langle \textbf{prepare}, \mathrm{h}, e, \mathrm{h}_\star, e_\star, \mathcal{S} \rangle$$

Notation	**Description**
h	a checkpoint hash
e	the epoch of the checkpoint
$\mathrm{h}_\star$	the most recent justified hash
$e_\star$	the epoch of $\mathrm{h}_\star$
$\mathcal{S}$	signature of $(\mathrm{h}, e, \mathrm{h}_\star, e_\star)$ from the validator's private key

$$\langle \textbf{commit}, \mathrm{h}, e, \mathcal{S} \rangle$$

Notation	**Description**
h	a checkpoint hash
e	the epoch of the checkpoint
$\mathcal{S}$	signature from the validator's private key

A checkpoint h is considered *justified* if there exists some $\mathrm{h}_\star$ such that:

1. There exist $\frac{2}{3}$ prepares of the form $\langle \textbf{prepare}, \mathrm{h}, e(\mathrm{h}), \mathrm{h}_\star, e(\mathrm{h}_\star), \mathcal{S} \rangle$
2. $\mathrm{h}_\star$ itself is justified

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

Note that all $\frac{2}{3}$ prepares that justify h must have the same $\mathrm{h}_\star$; half with $\mathrm{h}_\star^1$ and half with $\mathrm{h}_\star^2$ will not suffice. A checkpoint h is considered *finalized* if:

1. h is justified
2. There exist $\frac{2}{3}$ commits of the form $\langle \mathbf{commit}, \mathrm{h}, e(\mathrm{h}), \mathcal{S} \rangle$

The genesis is considered to be justified and finalized, serving as the base case for the recursive definition.

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 $\mathrm{h}_\star$ a checkpoint for any previous epoch (preferably the preceding checkpoint) of h , and which is *justified* (see below), and the $e_\star$ is expected to be the epoch of that checkpoint.

Validators only recognize prepares and commits that have been included in blocks (even if those blocks are not part of the main chain). [2]

2.1 Casper Commandments

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 Casper Commandments (a.k.a. slashing conditions). These are:

I. A validator shalt not publish two or more nonidentical Prepares for same epoch.

In other words, a validator may prepare at most exactly one (h , $e_\star$, $\mathrm{h}_\star$) triplet for any given epoch e .

II. A validator shalt not publish a Commit between the epochs of a Prepare statement.

Equivalently, a validator may not publish both $\langle \mathbf{prepare}, e_p, \mathrm{h}_p, e_\star, \mathrm{h}_\star, \mathcal{S} \rangle$ AND $\langle \mathbf{commit}, e_c, \mathrm{h}_c, \mathcal{S} \rangle$ if $e_\star < e_c < e_p$.

[2] 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 descendants, similarly to the longest chain rule and GHOST[**?**] in proof of work. Also, because the blockchain will not accept (i) commits that do not point to an already justified checkpoint, (ii) prepares that do not point to an already justified $\mathrm{h}_\star$ value, or (iii) prepares or commits that supply incorrect epoch numbers, this makes two of the four slashing conditions in earlier versions of Casper[**?**] superfluous.

If a validator violates a slashing condition, the evidence that they 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.

Finally, we define the "ideal execution" of the Casper protocol during an epoch n, as every validator preparing C_n with $\mathrm{h}_\star = C_{n-1}$ and commiting C_n. For example, during epoch 2 (blocks $200 \ldots 299$), all validators prepare b_{199} with $\mathrm{h}_\star = b_{99}$ and commit b_{199}.

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.

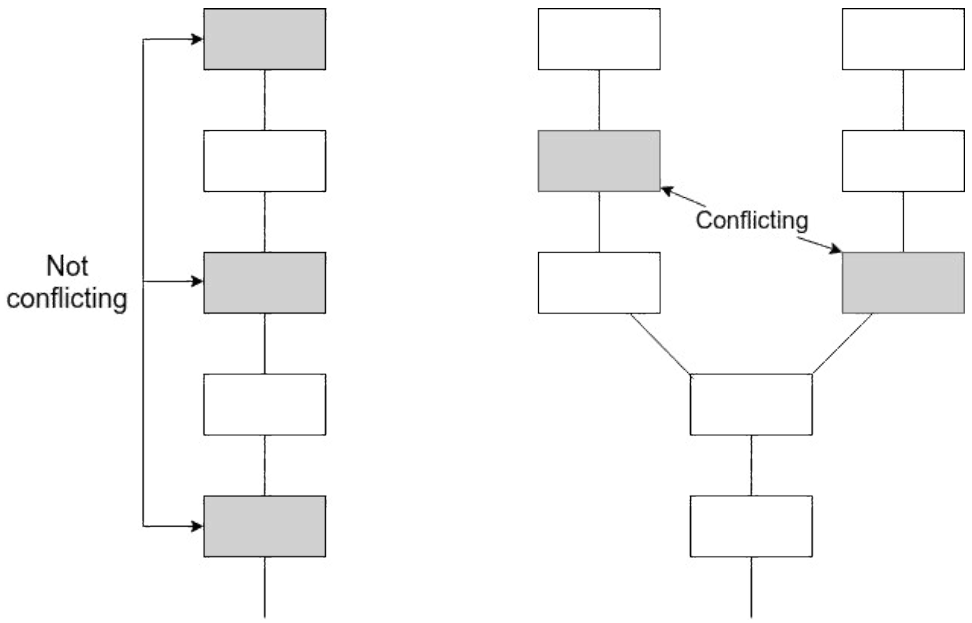
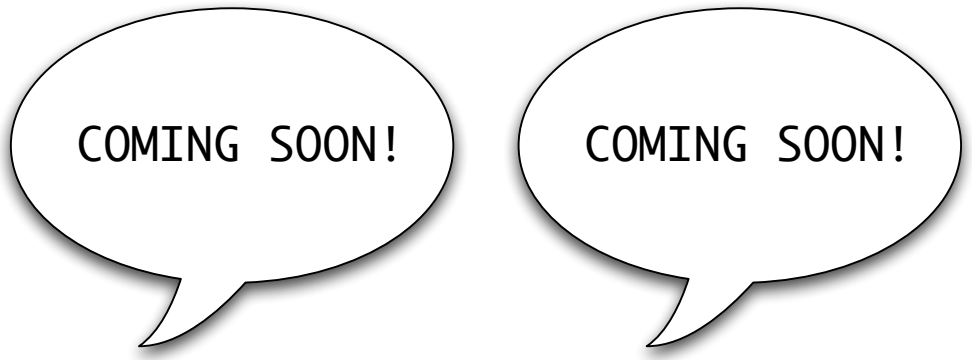


Figure 1: Two checkpoints are *conflicting* if they are on distinct chains, i.e. one is not an ancestor or a descendant of the other.

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. If both are finalized, this implies $\frac{2}{3}$ commits and $\frac{2}{3}$ prepares in epochs e_A and e_B. In the trivial case where $e_A = e_B$, this implies that some intersection of $\frac{1}{3}$ of validators must have violated slashing condition (1). In other cases, there must exist two chains $G < \ldots < e_A^2 < e_A^1 < e_A$ and $G < \ldots < e_B^2 < e_B^1 < e_B$ of justified checkpoints, both terminating at the genesis. Suppose without loss of generality that $e_A > e_B$. Then, there must be some e_A^i that either $e_A^i = e_B$ or $e_A^i > e_B > e_A^{i+1}$. In the first case, since A^i and B both have $\frac{2}{3}$ prepares, at least $\frac{1}{3}$ of validators violated slashing condition (I). Otherwise, B has $\frac{2}{3}$ commits and there exist $\frac{2}{3}$ prepares with $e > B$ and $e_\star < B$, so at least $\frac{1}{3}$ of validators violated slashing condition (II). □



$e_A = e_B$ $\qquad\qquad$ $e_A < e_B$

Figure 2: Illustrating the two scenarios in Theorem 1.

Theorem 2 (Plausible Liveness)**.** *It is always possible for* $\frac{2}{3}$ *of honest validators to finalize a new checkpoint, regardless of what previous events took place.*

Proof. Suppose that all existing validators have sent some sequence of prepare and commit messages. Let M with epoch e_M be the highest-epoch checkpoint that was justified, and let $n \geq e_M$ be the highest epoch in which

an honest validator prepared. Honest validators have not committed on any block which is not justified. Hence, neither slashing condition stops them from making prepares on a descendant of M in epoch $n+1$, using e_M as $e_\star$, and then committing this child. □

4 Tweaking the Proposal Mechanism

Although Casper is chiefly an overlay on top of a proposal mechanism, in order to translate the *plausible liveness* proven into the previous section into *actual liveness in practice*, the proposal mechanism needs to be Casper-aware. If the proposal mechanism isn't Casper-aware, as is the case with a proof of work chain that follows the typical fork-choice rule of "always build atop the longest chain", Casper can get stuck where no further checkpoints are finalized. We see one such example in Figure 3.

In this case, $HASH1$ or any descendant thereof cannot be finalized without slashing $\frac{1}{6}$ of validators. However, miners on a proof of work chain would interpret $HASH1$ as the head and forever keep mining descendants of it, ignoring the chain based on $HASH0'$ which actually could get finalized.

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. This necessitates modifying the proposal mechanism so that, in the event that some checkpoint is justified and receives any commits, any new blocks that are proposed are descendants of that checkpoint, and not conflicting checkpoints that have fewer or no commits.

In the case where the proposal mechanism is a proof of work chain, this entails modifying the fork choice rule: instead of blindly following a longest-chain rule, there needs to be an overriding rule that (i) finalized checkpoints are favored, and (ii) when there are no further finalized checkpoints, checkpoints with more (justified) commits are favored.

One complete description of such a rule would be:

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

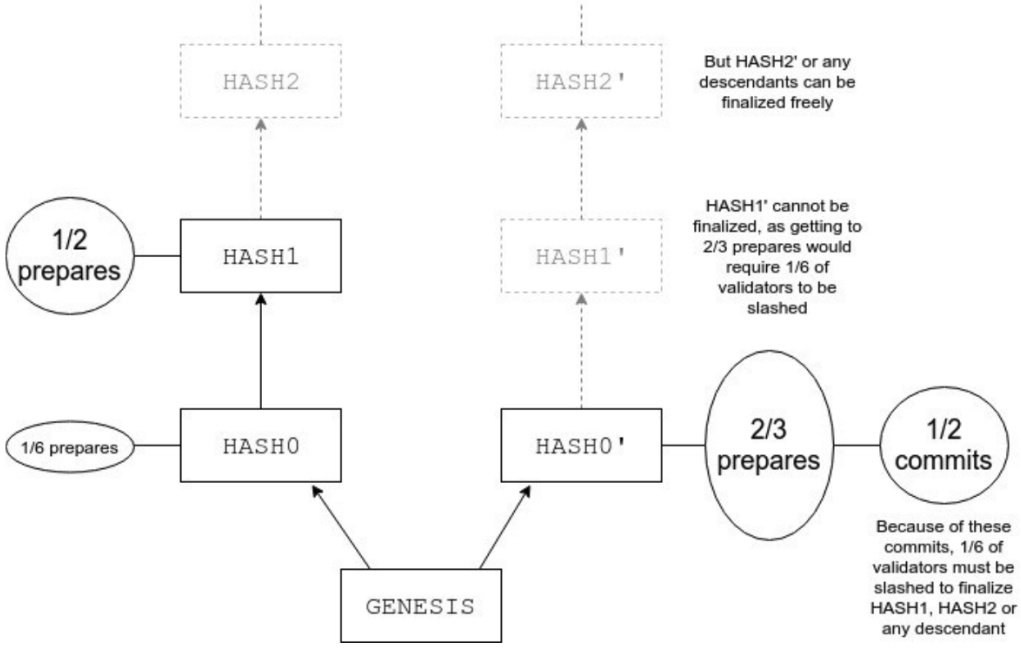


Figure 3: Miners following the traditional proof of work fork choice rule would create blocks on HASH1, but because of the slashing conditions only blocks on top of HASH1' can be finalized.

The commit-following part of this rule can be viewed as mirroring the "greegy heaviest observed subtree" (GHOST) rule that has been proposed for proof of work chains[**?**]. [3]

[3]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 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.

This rule ensures that if there is a checkpoint such that no conflicting checkpoint can be finalized without at least some validators violating slashing conditions, then this is the checkpoint that will be viewed as the "head" and thus that validators will try to commit on. [4]

5 Allowing Dynamic Validator Sets

The set of validators needs to be able to change. New validators need to be able to join, and existing validators need to be able to leave. To accomplish this, we define a variable kept track of in the state called the *dynasty* counter. When a user sends a "deposit" transaction to become a validator, if this transaction is included in dynasty n, then the validator will be *inducted* in dynasty $n + 2$. The dynasty counter increments when the chain detects that the checkpoint of the current epoch that is part of its own history has been *perfectly finalized* (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. We call such a validator's *start dynasty* $n + 2$.

For a validator to leave, they must send a "withdraw" message. If their withdraw message gets included during dynasty n, the validator similarly leaves the validator set during dynasty $n+2$; we call $n+2$ their *end dynasty*. When a validator withdraws, their deposit is locked for a long period of time (the *withdrawal delay*, for now think "four months") before they can take their money out; if they are caught violating a slashing condition within that time then their deposit is forfeited.

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 (that is, validators with $startDynasty \leq curDynasty < endDynasty$), and (ii) at least $\frac{2}{3}$ of the previous dyansty (that is, validators with $startDynasty \leq curDynasty - 1 < endDynasty$. Finalization with commits works similarly. The current and previous dynasties will usually greatly overlap; but in cases where they substantially diverge this "stitching" mechanism ensures that dynasty diver-

[4] Favoring checkpoint with even a single commit, instead of $\frac{1}{3} + \epsilon$, is desired to ensure that plausible liveness translates into actual liveness even in scenarios where up to $\frac{1}{3}$ of validators are offline.

gences do not lead to situations where a finality reversion or other failure can happen because different messages are signed by different validator sets and so equivocation is avoided.

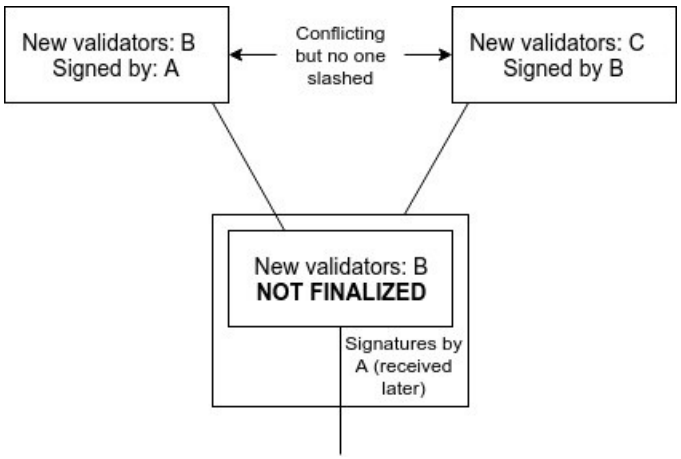


Figure 4: Without the validator set stitching mechanism, it's possible for two conflicting checkpoints to be finalized with no validators slashed

5.1 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.

Suppose that clients can be relied on to log on at least once every time δ (think $\delta \approx 1$ month). Then, if a client hears about two conflicting finalized checkpoints, C_1 at time T_1 and C_2 at time T_2, there are two cases. If, from the point of view of all clients, $T_2 > T_1$, then all clients will accept C_2 and there is no ambiguity. If on the other hand different clients see different orders (i.e. some see $T_2 > T_1$, others see $T_1 > T_2$), then it must be the case that for all clients, $|T_2 - T_1| \leq 4\delta$. If the withdrawal delay is much greater than 4δ, then there is plenty of time during which slashing evidence can be included in both chains, and we make an assumption that if a chain does

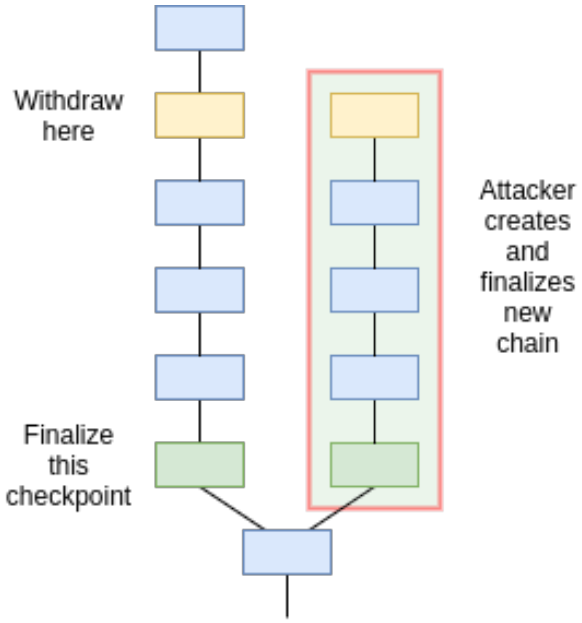


Figure 5: Despite violating slashing conditions to make a chain split, because the attacker has already withdrawn on both chains they do not lose any money. This is often called a *long-range atack*.

not include slashing evidence, then clients will reject this via a soft fork (see later section).

This rejection can also be partially automated, by implementing a mechanism where if a client sees a slashing condition violation before they see some block B with timestamp t, and then the head is a descendant of B with timestamp greater than $t + EIT$ ("evidence inclusion timelimit") that does has not yet punished the malfeasant validator, the client shows a warning.

6 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 become 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 be-

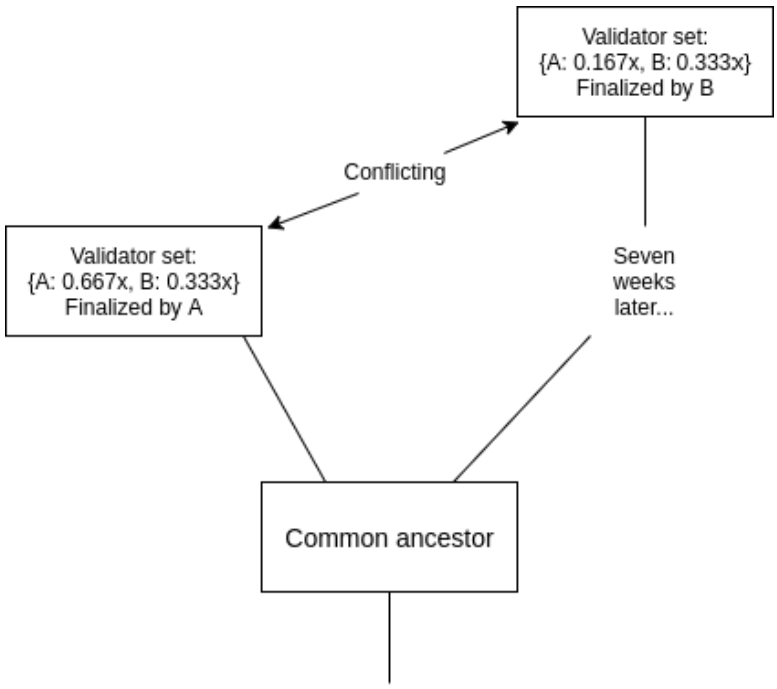


Figure 6: The checkpoint on the left can be finalized immediately. The checkpoint on the right can be finalized after some time, once offline validator deposits sufficiently dissipate.

cause 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.

7 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.

8 Acknowledgements

We thank Virgil Griffith for review and Sandro Lera for mathematics.