A Parsimonious Stake-based Finalty Mechanism

Vitalik Buterin
Ethereum Foundation

August 25, 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. Byzantine fault tolerance analysis is included, but economic incentive analysis is out of scope. [To be reworked at the very end]

1. Introduction

[Probably talk about prior Proof of Stake / Finality based work here.]

2. Principles

[This might be stuff that we can roll into the Introduction] Casper the Friendly Finality Gadget is an overlay on top of some kind of "proposal mechanism" - a mechanism which "proposes" blocks which the Casper mechanism can then set in stone by "finalizing" them. Casper depends on the proposal mechanism for liveness, but not safety; that is, if the proposal mechanism is entirely controlled by attackers, then the attackers can prevent Casper from finalizing any future checkpoints, but cannot cause a safety failure in Casper—i.e., they cannot force Casper to finalize two conflicting blocks.

The base mechanism is heavily inspired by partially synchronous systems such as Tendermint [**?**] and PBFT [**?**] , and thus has $\frac{1}{3}$ Byzantine fault tolerance and is safe under asynchrony and dependent on the proposal mechanism for liveness.

In Section 6.1 we introduce a modification which increases Byzantine fault tolerance to $\frac{1}{2}$, with the proviso that attackers with size $\frac{1}{3} < \alpha < \frac{1}{2}$ can delay new blocks being finalized by some period of time D (think $D \approx 3$ weeks), at the cost of a "tradeoff synchrony assumption" where fault tolerance decreases as network latency goes up, decreasing to potentially zero when network latency reaches D.

In the Casper Phase 1 implementation for Ethereum, the "proposal mechanism" is the existing proof of work chain, modified to have a greatly reduced block reward because the chain no longer relies as heavily on proof of work for security, and we describe how the Casper mechanism, and fork choice rule, can be "overlaid" onto the proof of work mechanism in order to add Casper's guarantees.

3. The Protocol

In the Casper protocol, there is a set of validators, and in each epoch validators have the ability to send two kinds of messages:

Notation	Description
h	the hash to justify
e	the current epoch
$\mathrm{h}_\star$	the most recent justified hash
$e_\star$	the epoch containing hash $\mathrm{h}_\star$
$\mathcal{S}$	signature from the validator's private key of the tuplet $(\mathrm{h}, e, \mathrm{h}_\star, e_\star)$.

(a) **PREPARE** format

Notation	Description
h	the hash to finalize
e	the current epoch
$\mathcal{S}$	signature from the validator's private key

(b) **COMMIT** format

Table 1: The schematic of the **PREPARE** and **COMMIT** messages.

Each validator has a *deposit size*; when a validator joins their deposit size is equal to the number of coins that they deposited, and from there on each validator's deposit size 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.

Every hash h has one of three possible states: *fresh*, *justified*, and *finalized*. Every hash starts as *fresh*. The hash at the beginning of the current epoch converts from fresh to *justified* if, during the current epoch e, $\frac{2}{3}$ Prepares are sent of the form

$$\langle \textbf{PREPARE}, e, \mathrm{h}, e_\star, \mathrm{h}_\star, \mathcal{S} \rangle \tag{1}$$

for some specific $e_\star$ and $\mathrm{h}_\star$. A hash h can only be justified if and only if its $\mathrm{h}_\star$ is already justified or finalized.

Additionally, a hash converts from justified to *finalized*, if $\frac{2}{3}$ of validators Commit

$$\langle \textbf{COMMIT}, e, \mathrm{h}, \mathcal{S} \rangle \ , \tag{2}$$

for the same e and h as in eq. 3. The h is the block hash of the block at the start of the epoch. A hash h being justified entails that all fresh (non-finalized) preceding blocks are also justified. A hash h being finalized entails that all preceding 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 $e_\star$ and $\mathrm{h}_\star$.

An *epoch* is a period of 100 blocks; epoch n begins at block $n * 100$ and ends at block $n * 100 + 99$. The final in each epoch in a *checkpoint block*. Checkpoint blocks are special

A *checkpoint for epoch* n is a block with number $n * 100 - 1$; in a smoothly running blockchain there will usually be only one checkpoint per epoch, but due to natural network latency or deliberate attacks there may be multiple competing checkpoints during some epochs. The *predecessor* of a checkpoint is the checkpoint in the prior epoch, and an *lineage* of a checkpoint is the set of all preceding checkpoints, recursively, to the Genesis block $\mathbb{G}$.

Probably define some mathematical notation here for Block B, Checkpoint C, Predecessor, Successor, Lineage, etc.

We define the *lineage hash* of a checkpoint as follows:

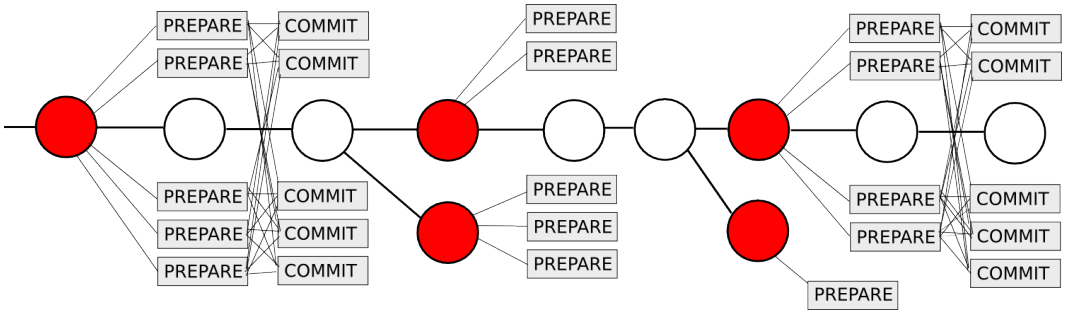

Figure 1: Illustrating the sequence of Prepares and Commits. [revise?]

- The lineage hash of the implied "genesis checkpoint" of epoch 0 is thirty two zero bytes.
- The lineage hash of any other checkpoint is the keccak256 hash [cite] of the lineage hash of its predecessor checkpoint concatenated with the hash of the checkpoint.

$$AH(n) \equiv \texttt{keccak256}\left[\texttt{CONCAT}\left(AH(n-1), \texttt{keccak256}[C_n]\right)\right] \tag{3}$$

Lineage hashes thus form a direct hash chain, and otherwise have a one-to-one correspondence with checkpoint hashes.

During epoch n, validators are expected to send prepare and commit messages specifying n as their e, and the lineage hash of some checkpoint for epoch n as their h. 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.

Each validator has a *deposit size*; when a validator joins their deposit size is equal to the number of coins that they deposited, and from there on each validator's deposit size rises and falls as the validator receives 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 combined deposit size equals to at least $\frac{2}{3}$ of the total deposit size of the entire set of validators. We initially consider the set of validators, and their deposit sizes to be static static, but in Section 6 show how to validators can join and leave the validator set.

We also add the following new requirements:

- For a checkpoint to be finalized, it must be justified.
- For a checkpoint to be justified, the $\mathrm{h}_\star$ used to justify it must itself be justified.
- Prepare and commit messages are only accepted as part of blocks; that is, for a client to consider a checkpoint as finalized, the client must see a chain [notation?] such that the chain terminating at that block, $\frac{2}{3}$ commits for that hash have been included and processed.

This immensely simplifies our finalty mechanism because we can now have 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 [**?**]. 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 competing blocks, and (ii) it is impossible for two incompatible checkpoints to be finalized unless at least $\frac{1}{3}$ of the validators violated a Casper Commandment (a.k.a. slashing conditions),

I. A validator shalt not publish two or more nonidentical Prepares for same epoch. This is equivalent to that each validator may Prepare to exactly one (h, $e_\star$, $\mathrm{h}_\star$) triplet per epoch.

II. A validator shalt not publish an Commit between the epochs of a Prepare statement. Equivalently, a validator will not publish

$$\langle \textbf{PREPARE}, e_p, \mathrm{h}_p, e_\star, \mathrm{h}_\star, \mathcal{S} \rangle \qquad \text{AND} \qquad \langle \textbf{COMMIT}, e_c, \mathrm{h}_c, \mathcal{S} \rangle \ ,$$

where the epochs satisfy $e_\star < e_c < e_p$.

Lemma 1 (Enforcable Slashing Conditions)**.** *If a group of attackers violates a slashing condition, their deposits will be slashed as long as [condition here.]*

Proof. [Proof goes here.] □

Earlier versions of Casper had four slashing conditions, [**?**] but we can reduce to two because of the three new requirements above; they ensure that blocks will not register commits or prepares that violate the other two conditions.

4. Proofs of Safety and Plausible Liveness

We give a proof of two properties of Casper: *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 $\frac{\mathrm{TD}}{3}$ is lost). 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.*

Plausible liveness means that it is always possible for $\frac{2}{3}$ of honest validators to finalize a new checkpoint, regardless of what previous events took place.

In Figure 2, we see two conflicting checkpoints A (epoch e_A) and B (epoch e_B) are finalized.

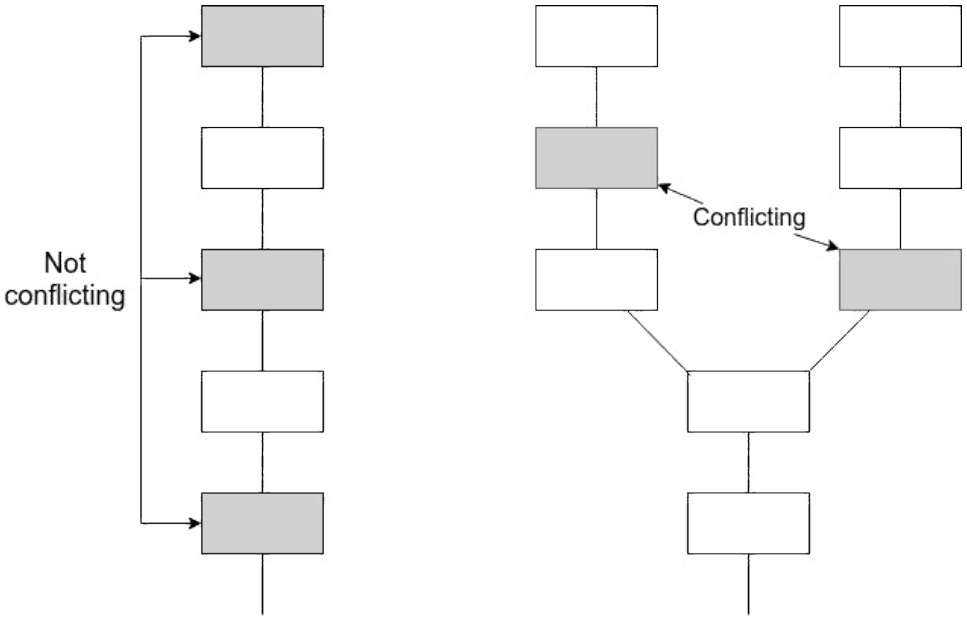

Figure 2: [fill me in.]

Theorem 2 (Accountable Safety)**.** *Two conflicting checkpoints cannot be finalized unless* $\geq \frac{1}{3}$ *of validators violate a slashing condition (meaning at least* $\frac{\mathrm{TD}}{3}$ *is lost).*

Proof. 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 **NO_DBL_PREPARE**. In other cases, there must exist two chains $\mathbb{G} < \ldots < e_A^2 < e_A^1 < e_A$ and $\mathbb{G} < \ldots < e_B^2 < e_B^1 < e_B$ 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 **NO_DBL_PREPARE**. 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 **PREPARE_COMMIT_CONSISTENCY**. This proves accountable safety. □

Theorem 3 (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. Honest validators have not committed on any block which is not justified. Hence, neither slashing condition stops them from making prepares on a child of M, using e_M as $e_\star$, and then committing this child. □

5. Fork Choice Rule

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 (i.e. the proof of work chain) gets into a state where it never ends up creating a checkpoint that could get finalized.

In Figure 3 we see one possible example. 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 fork choice rule used by participants in the underlying proposal mechanism (as well as users and validators): 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. 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 commit-following part of this rule can be viewed in some ways as mirroring the "greegy heaviest observed subtree" (GHOST) rule that has been proposed for proof of work chains [**?**]. 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

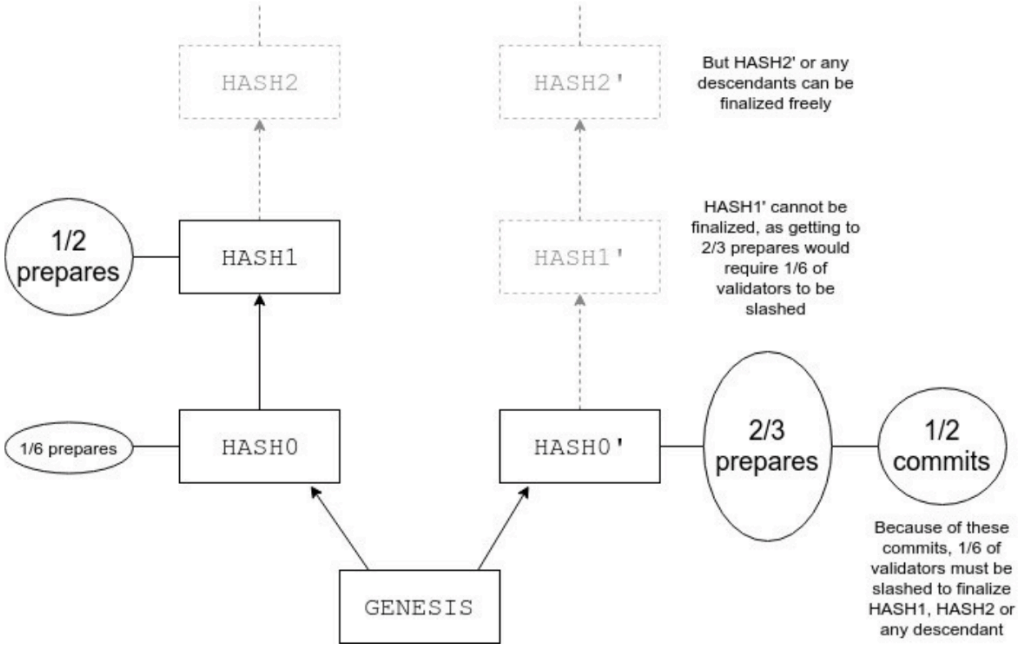

Figure 3: [fill me in.]

most faithfully. Finalizing a checkpoint requires $\frac{2}{3}$ commits within a *single* epoch, and so we do not try to sum up commits across epochs and instead simply take the maximum.

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.

6. 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 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 finalized, and then they need to wait again for that epoch to be finalized; after this, they become part of the validator set. We call such a validator's *start dynasty* $n+2$. [The conditions here feel different. Do we mean we want to have two *perfect finalizations* (finalizing the epoch immediately prior) or simply two finalizations?]

For a validator to leave, ze 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 four months [how determined?] 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 divergences 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.

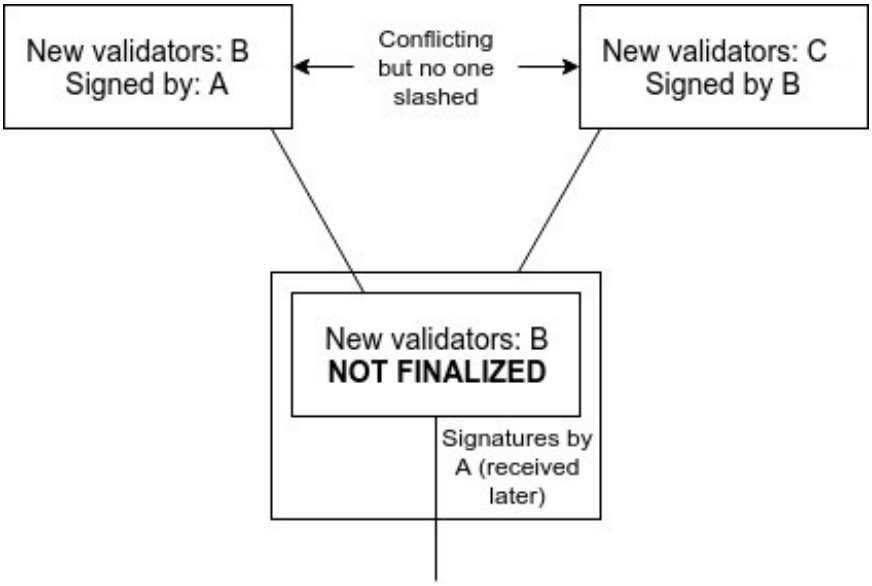

Figure 4: [This is meant to demonstrate the need for the "stitching" of dynamic validator sets, correct?]

6.1. 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" (eq. 4) which increases the longer validators do not prepare or commit any checkpoints, until eventually their deposit sizes decrease low enough that the validators that *are* preparing and committing are a $\frac{2}{3}$ supermajority.

Given two constants B, the number of blocks until leak completely dissipates an inactive vaidator's deposit, and s the "steepness" of the curve, we have the formula for the validator leak as a function of the number of inactive blocks as,

$$\text{leak}(x) = \frac{s+1}{B^{s+1}} x^s \text{ , [notation likely to be changed.]} \tag{4}$$

for which the derivation is given in Appendix A. One can set parameters B and s depending on the desired penalty curves.

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 5.

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%

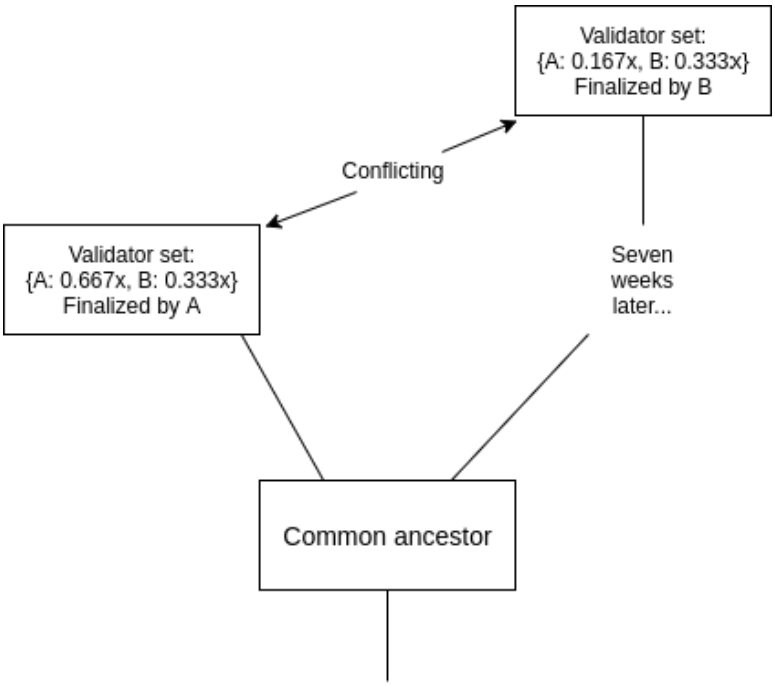

Figure 5: [caption here.]

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.

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.

Future Work. [fill me in]

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

Appendix

A. Leak Formula Derivation

[Put details of the derivation of the leak formula here.]

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.]

for the same e and h as in eq. 3. The h is the block hash of the block at the start of the epoch. A hash h being justified entails that all fresh (non-finalized) ancestor blocks are also justified. A hash h 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 $e_\star$ and $\mathrm{h}_\star$.

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*.

C. Notes To Authors

C.1. Questions

- True/False: The Dynasty counter increments iff there's been a finalization?

C.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 } → adjacent

C.3. Todo

- ~~Reference the various Figures within the text so we more easily know what goes with what.~~
- [fill me in]