Vitalik Buterin Follow
Mar 2, 2017 · 14 min read · Listen

Minimal Slashing Conditions

Last week Yoichi released a blog post detailing the process of formally proving safety and liveness properties of my "minimal slashing conditions". This is a key component of a Byzantine-fault-tolerant, safe-under-asynchrony and cryptoeconomically safe consensus algorithm that is at the core of my proof of stake roadmap. In this post I want to provide further details on what this algorithm is, what its significance is, and how it generally fits into proof of stake research.

A key goal of Casper is that of achieving "economic finality", which we can semi-formally define as follows:

> *A block B1 is **economically finalized**, with cryptoeconomic security margin $X, if a client has proof that either (i) B1 is going to be part of the canonical chain forever, or (ii) those actors that caused B1 to get reverted are guaranteed to be economically penalized by an amount equal to at least $X.*

Think X ~= $70 million. Basically, if a block is finalized, then that block is part of the chain, and it is very very expensive to cause that to change. Proof of work does not really have this; this is a unique feature of proof of stake[1]. The intention is to make 51% attacks extremely expensive, so that even a majority of validators working together cannot roll back finalized blocks without undertaking an extremely large economic loss — a loss so large that a successful attack would likely on net *increase* the price of the underlying cryptocurrency as the market would more strongly react to the reduction in total coin supply than it would to the need for an emergency hard fork to correct the attack (see here for a deeper overview of the underlying philosophy).

they acted in some way that violates some set of rules ("**slashing conditions**").

A slashing condition might look like this:

> *If a validator sends a signed message of the form*
>
> `["PREPARE", epoch, HASH1, epoch_source1]`
>
> *and a signed message of the form*
>
> *`["PREPARE", epoch, HASH2, epoch_source2]`*
>
> *where* `HASH1 != HASH2` *or* `epoch_source1 != epoch_source2`*, but the* `epoch` *value is the same in both messages, then that validator's deposit is* ***slashed*** *(ie. deleted)*

The protocol defines a set of slashing conditions, and honest validators follow a protocol that is guaranteed not to trigger any of the conditions (note: we sometimes say "violating" a slashing condition as a synonym for "triggering"; think of slashing conditions as being like laws that you are not supposed to break); never sending a `PREPARE` message in the same epoch twice is something that is not too difficult for an honest validator.

"PREPARE" and "COMMIT" are terms borrowed from traditional Byzantine fault tolerant consensus theory. For now, just think of them as being two different types of messages; in the later protocol that we will introduce you can think of consensus as requiring two rounds of agreement, where PREPARE represents the first round and COMMIT represents the second round.

There is also a **finality condition**, which describes when a client can determine that some particular hash is finalized. This is easier; we'll just go ahead and state the sole finality condition in the current version of Casper right now.

> *A* `HASH` *is* ***finalized*** *if, for some particular* `epoch` *number, there exists a set of signed messages of the form*

> *and if you add up the deposited balances of the validators that created these signed messages then you get an amount that is more than 2/3 of the total deposited balances of the current active validator set.*

As shorthand, we can say "the hash was committed in some particular epoch by 2/3 of validators", or "the hash has 2/3 commits in some particular epoch".

The slashing conditions need to satisfy two conditions:

1. **Accountable safety**: if two conflicting hashes get finalized, then it must be provably true that at least 1/3 of validators violated some slashing condition.[2]
2. **Plausible liveness**: unless at least 1/3 of validators violated some slashing condition, there must exist a set of messages that 2/3 of validators can send which finalize some new hash without violating some slashing condition.

Accountable safety is what brings us this idea of "economic finality": if two conflicting hashes get finalized (ie. a fork), then we have mathematical proof that a large set of validators must have violated some slashing condition, and we can submit evidence of this to the blockchain and penalize them[3].

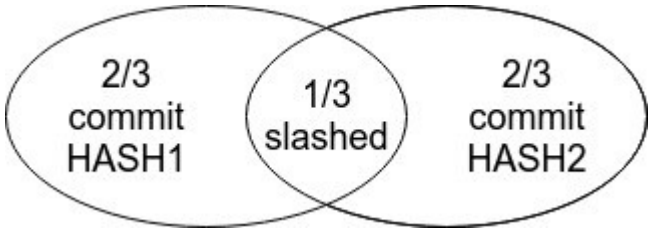
Plausible liveness basically means that "it should not be possible for the algorithm to get '**stuck**' and not be able to finalize anything at all".

To illustrate the meaning of these two concepts, we can consider two toy algorithms, one which satisfies safety but not liveness, and one which satisfies liveness but not safety.

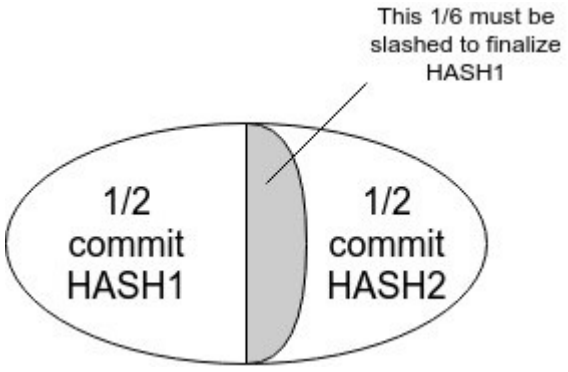
> ***Algorithm 1**: every validator has exactly one opportunity to send a message of the form* `["COMMIT", HASH]`*. If 2/3 of validators send a* `COMMIT` *for the same hash, that hash is finalized. Sending two* `COMMIT` *messages violates a slashing condition.*




overlap, so 1/3 of validators get slashed. But it is not plausibly live: if 1/2 commit A and 1/2 commit B (a totally reasonable thing that could happen accidentally), then 1/6 of validators must voluntarily slash themselves in order to finalize a hash.



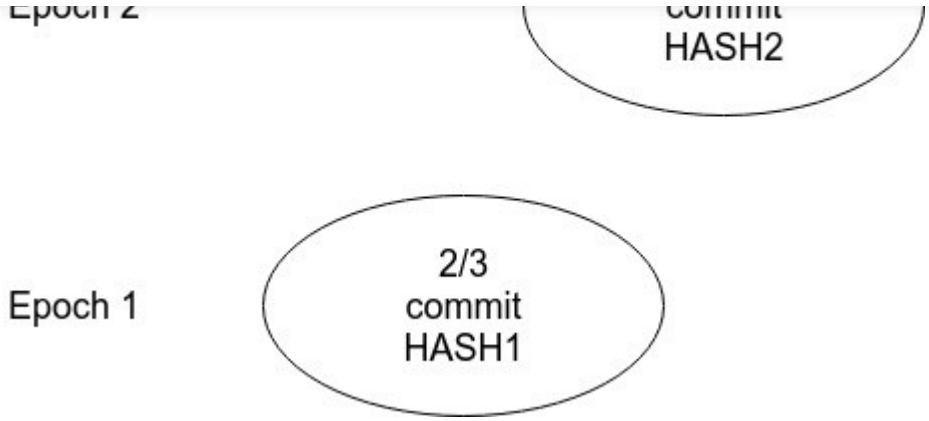
Safety proof



But we can get stuck

> ***Algorithm 2:*** *every validator has exactly one opportunity to send a message of the form* `["COMMIT", HASH, epoch]`*. If 2/3 of validators send a* `COMMIT` *for the same hash with the same* `epoch` *number, that hash is finalized. Sending two* `COMMIT` *messages with different hashes with the same* `epoch` *number violates a slashing condition.*

This gets around the problem in the previous algorithm, as if during one `epoch` we get a 50/50 situation then we can simply try again in the next `epoch`. But this introduces a safety flaw: two different hashes can get finalized in different epochs!



No more safety.

It turns out that it is possible to get both at the same time, but this is nontrivial, and requires 4 slashing conditions, plus 1000 lines of code for Yoichi to formally prove that it actually works.

The slashing conditions are:

> ***[COMMIT_REQ]*** *If a validator sends a signed message of the form*
>
> `["COMMIT", epoch, HASH]`
>
> *then unless, for some specific value* `epoch_source`*, with* `-1 <= epoch_source < epoch`*, messages of the form*
>
> `["PREPARE", epoch, HASH, epoch_source]`
>
> *have been signed and broadcasted by 2/3 of validators, then that validator's deposit is slashed.*

In plain English, sending a commit requires seeing 2/3 prepares.

> ***[PREPARE_REQ]*** *If a validator sends a signed message of the form*

> `["PREPARE", epoch_source, ANCESTOR_HASH, epoch_source_source]`
>
> *, where* `ANCESTOR_HASH` *is the* `(epoch - epoch_source)` *— degree ancestor of* `HASH` *, have been signed and broadcasted by 2/3 of validators, then that validator's deposit is slashed.*

If you make a prepare in some epoch pointing to some particular previous epoch, then you need to have seen 2/3 prepares in that epoch, and those prepares must point to the same previous epoch (eg. 2/3 prepares in epoch 41 pointing to epoch 35 are fine, 2/3 prepares in epoch 41 with half pointing to epoch 35 and half pointing to 37 are not; 5/6 prepares in epoch 41 with 4/5 of them pointing to epoch 35 is also fine, because 4/5 of 5/6 is 2/3, and you can just ignore the other 1/6).

By "n-th degree ancestor", we mean an ancestor in the blockchain hash-chaining sense of the term, where for example Ethereum block 3017225 is the 15th-degree ancestor of Ethereum block 3017240. Note that a block may only have one parent, and so only one n-th degree ancestor for any specific value of n.

> ***[PREPARE_COMMIT_CONSISTENCY]*** *If a validator sends a signed message of the form*
>
> `["COMMIT", epoch1, HASH1]`
>
> *and a prepare of the form*
>
> `["PREPARE", epoch2, HASH2, epoch_source]`
>
> *where* `epoch_source < epoch1 < epoch2` *, then, irrespective of whether or not* `HASH1 != HASH2` *, the validator is slashed.*

If you make a commit during some epoch, then you clearly saw 2/3 prepares during that epoch, and so any future prepares that you do should better be referencing that epoch or something newer.

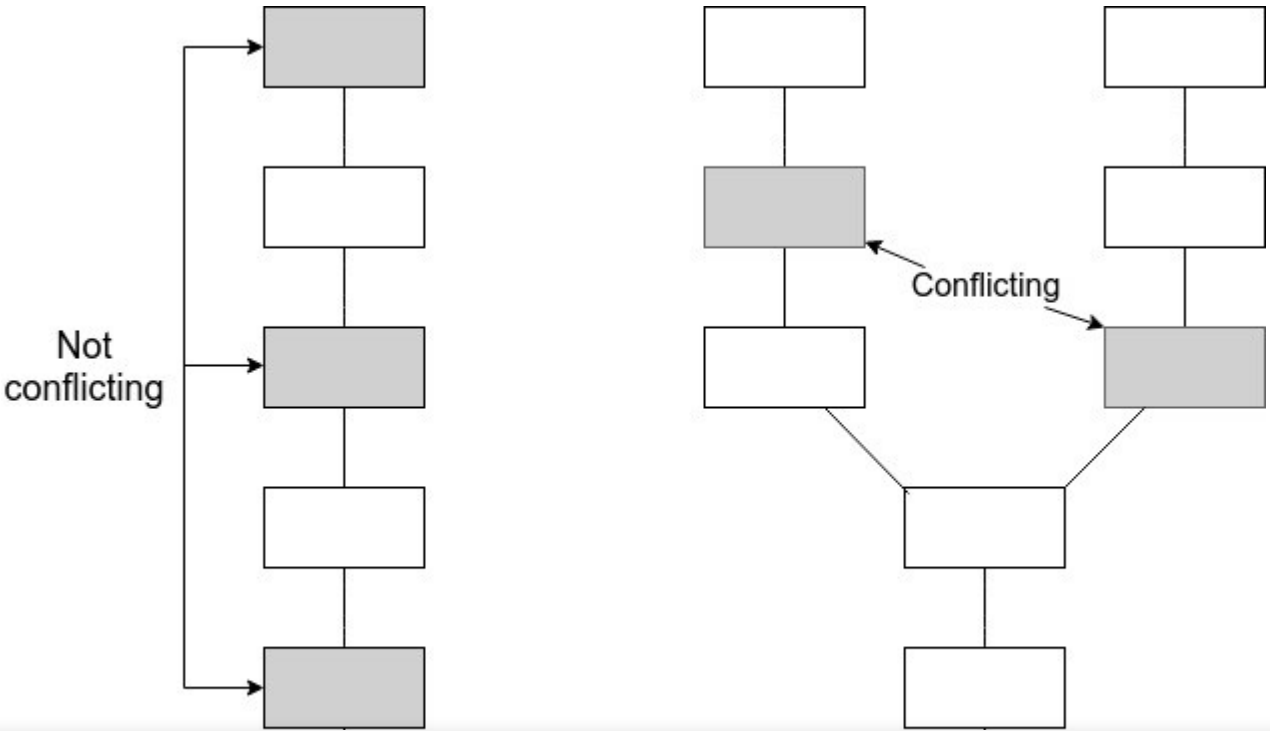
> ***[NO_DBL_PREPARE]*** *If a validator sends a signed message of the form*

> *and a signed message of the form*
>
> `["PREPARE", epoch, HASH2, epoch_source2]`
>
> *where* `HASH1 != HASH2` *or* `epoch_source1 != epoch_source2`*, but the* `epoch` *value is the same in both messages, then the validator is slashed*

Can't prepare twice in a single epoch.

With these four slashing conditions, it turns out that both accountable safety and plausible liveness hold.

Note that with the above rules it is possible for two *different* hashes to get finalized— if two hashes are both part of the same history then they can both get finalized, and in fact an ever-growing chain where more and more hashes at the end of the chain get finalized is exactly what we *want*.



slashed.

Now, let's put things together. There exists a pool of active validators (which anyone is free to join, with some delay, by submitting a deposit, and which any participant can leave and then, after some much higher delay, withdraw their funds), and these validators have the right to sign and send messages of the form:

```
["PREPARE", epoch, HASH, epoch_source]
```

```
["COMMIT", epoch, HASH]
```

If there are enough `COMMIT` messages for some `HASH` during some particular `epoch`, then that `HASH` is finalized. Hashes are chained to each other, with each hash pointing to some previous hash, and we expect to see an ever-growing chain of hashes with newer and newer hashes in the chain getting finalized as time goes on. We add economic rewards for validators to send these `PREPARE` and `COMMIT` messages, so that enough messages get sent in time for finalization to actually happen.

In general, you can take any Byzantine-fault-tolerant consensus algorithm[4] that has the "live under synchrony, safe under asynchrony" property of PBFT, and convert it into a set of slashing conditions that give you accountable safety and plausible liveness; the conditions above were inspired by a combination of PBFT and Tendermint, but there can also be other starting points that produce different results.

Note that *plausible liveness* is not the same thing as *actual liveness*; plausible liveness means that we theoretically can always finalize something, but it could always be the case that we just repeatedly get unlucky and never end up finalizing anything. To solve this problem, we need to come up with a **proposal mechanism**, and make sure that the proposal mechanism has the property that it actually does help us achieve liveness.




finalize something once the proposal mechanism stops being faulty.

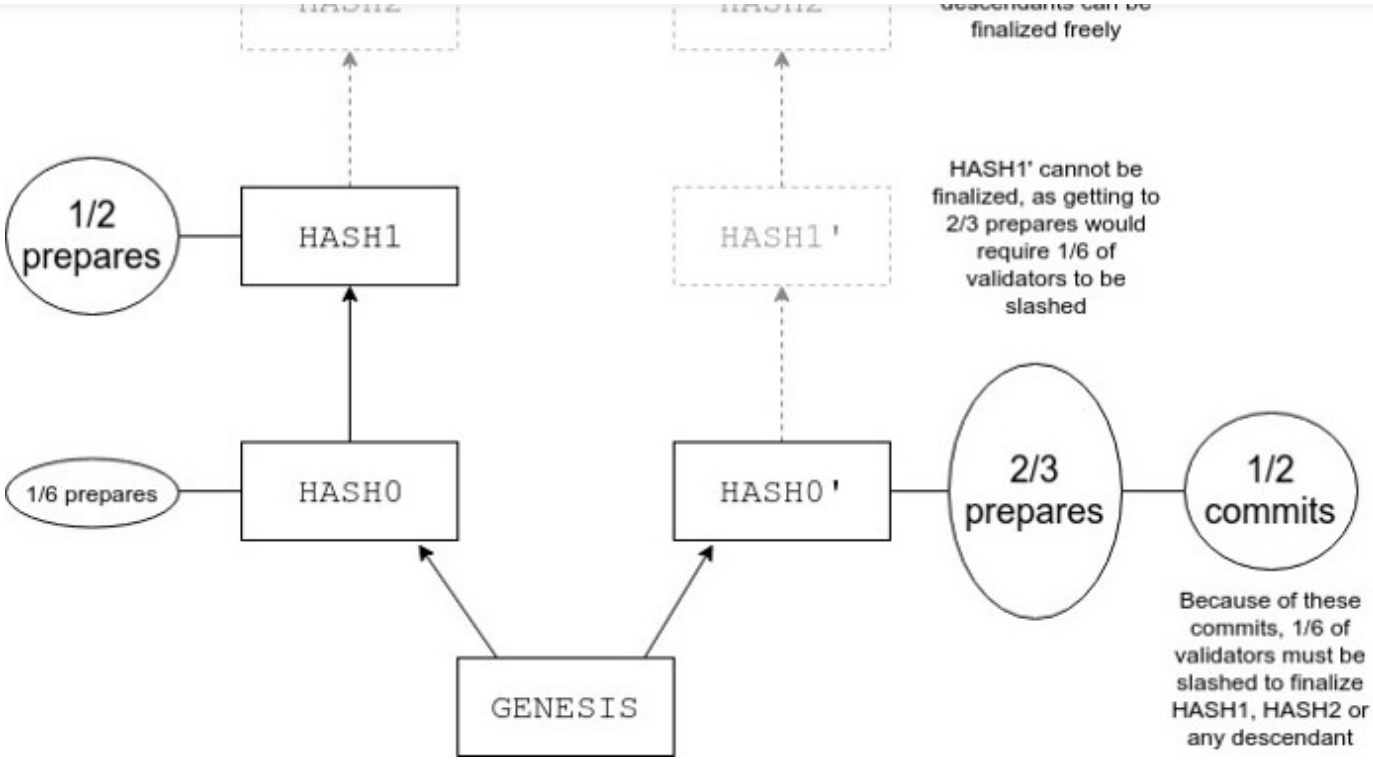
In many traditional Byzantine-fault-tolerant consensus algorithms, the proposal mechanism and the rest of the algorithm are closely tied together; in PBFT, every view (roughly equivalent to an epoch) is assigned a single validator, and that validator is free to propose whatever they want. The validator may misbehave by proposing nothing, proposing invalid hashes or proposing multiple hashes, but the other parts of the PBFT machinery ensure that such actions are not fatal, and the algorithm eventually switches over to the next epoch. Here, we can actually combine our slashing conditions with many different kinds of proposal mechanisms, as long as they satisfy a few conditions.

First of all, the proposal mechanism must propose **one hash per epoch**, and it must be a valid hash (validity conditions can be very complex; in Ethereum's case it involves executing and verifying the execution of the entire Ethereum state transition function, as well as verifying data availability).

Second, the **hashes must form a chain**; that is, a hash submitted for epoch N must have a parent that was submitted for epoch N-1, a 2nd degree ancestor that was submitted for epoch N-2, etc.

Third, **the hashes must be hashes that the slashing conditions do not prevent validators from finalizing.** This is more subtle. Consider a situation where, in epoch 0 the proposal mechanism proposes a hash `HASH0`, then in epoch 1 the proposal mechanism proposes a hash `HASH1` (a direct child of `HASH0`) and for whatever reason neither of these hashes reach enough prepares to get any commits. Then, the proposal mechanism (because of some temporary fault) proposes another hash `HASH0'` for epoch 0, and this gets 2/3 prepares and 1/2 commits.



Now, the proposal mechanism has two choices. One possibility is that it could propose `HASH2` (a direct child of `HASH1`), then `HASH3` (a direct child of `HASH2`), and so on. However, the slashing conditions ensure that none of these hashes could get committed without 1/6 of validators being slashed. The other, and correct, possibility is that it should propose `HASH1'` (a direct child of `HASH0'`), with the expectation that that hash may never get finalized because perhaps its competitor `HASH1` already got more than 1/3 prepares and so `HASH1'` can't get the 2/3 that it needs, and then propose `HASH2'` (a direct child of `HASH1'`). `HASH2'` *can* get committed, and the mechanism can then continue proposing new hashes, each a child of the previous.

One immediate thought that some might have is: can we make a traditional proof-of-work blockchain, using the longest chain rule, be the proposal mechanism? Every 100th block could be a kind of checkpoint, where the hash of block N * 100 becomes a proposal for epoch N. But this is not guaranteed to work by itself, because in the situation above the proposal mechanism would try to propose `HASH2` and not `HASH1'`, and so it would never end up finalizing any hashes (it's not quite what we call getting "stuck", as getting out of this situation does not require anyone to get slashed, but it

A fork choice rule is a function, evaluated by the client, that takes as input the set of blocks and other messages that have been produced, and outputs to the client what the "canonical chain" is. "Longest valid chain wins" is a simple fork choice rule and works well for PoW; Zohar and Sompolinsky's GHOST is a more complicated example. We can define a fork choice rule that allows a blockchain to serve as the proposal mechanism for a consensus algorithm, and have the above properties, as follows:

1. Start off with `HEAD` being the genesis.
2. Find the valid descendant of `HEAD` that has 2/3 prepares and the largest number of commits
3. Set `HEAD` to equal that descendant and go back to step 2.
4. When step 2 can no longer find a descendant with 2/3 prepares and any commits, use the blockchain's underlying fork choice rule (longest-chain, GHOST, whatever) to find the tip.

Note that in our example above, this would end up favoring `HASH0'` over `HASH1` , so it has the right desired behavior. Note also that if there is a finalized chain, it always selects the finalized chain.

The above slashing rules ensure that a specific type of fault is very expensive to create: a finality-reverting fork. However, there are other types of faults that this does not cover; particularly, finalizing an invalid hash, and finalizing a hash that represents a chain that has some data unavailable. Currently, the simplest known way to get around this issue with absolute cryptoeconomic security is to be a full node — download and validate all blocks, so that you can simply ignore invalid hashes. Determining that a given hash is finalized is thus a two-step process of (i) checking for the 2/3 commits, and (ii) checking that the chain up to the hash is valid.

has the effect that if this message is submitted into a chain where the given hash actually is the hash at that epoch, the validator gets a small reward, but if it is not then they pay a large penalty. Hence, validators will only send this message if they are really sure that a given hash is part of the canonical chain that clients see, and will continue to be forever (a good time for a validator to do this might be after they personally fully validate a blockchain up to the hash, and check that it has 2/3 commits).

The second is to give light clients access to various cryptoeconomic techniques that will allow them to verify data availability and validity very efficiently with the help of some weak honest minority assumptions; this will likely involve a combination of erasure codes and interactive verification. This is a similar kind of research to what is being done for sharding, and there is an intimate connection between the two — the first approach described above requires the validator themselves to be a full node, whereas the second does not, and sharding is ultimately all about creating a blockchain where *no one* is a full node. Expect to see more blog posts on this topic later.

Special thanks to Yoichi Hirai, River Keefer and Ed for reviewing.

Notes:

1. *Some might argue that proof of work has economic finality with security margin R * k, where R is the block reward and k is the number of blocks that are being reverted, but this isn't really true as if you make a successful 51% attack then you get compensated for your work with block rewards — the* ***budget*** *requirement for a 51% attack is indeed roughly R * k, but the* ***cost*** *if you succeed is zero.*

2. *The 1/3 ratio is taken from traditional Byzantine fault-tolerance theory and is chosen to maximize the total level of fault tolerance. In general, you can replace any mention of 2/3 in the slashing conditions in this article with any value t > 1/2; you can then compute the degree of fault tolerance from a liveness standpoint as 1-t (since you can't get to t if more than 1-t are offline), and the degree of fault tolerance from a safety standpoint as 2t-1 (if t finalize A and t finalize B, that's 2t > 1 altogether, so at least 2t-1 of them have to be duplicates). t = 2/3 maximizes the minimum of the two (1-t = 1/3, 2t - 1 = 1/3); you can*




situations, and the broader concept of "subjective finality thresholds", is a deep and interesting topic for another day.

3. *Unless of course the deposits have already been withdrawn; but that ("long range attacks") is also a topic for another post, albeit this time it is a post that was made over two years ago*

4. *At least, as long as it only uses hashes and signatures for cryptography. If it uses threshold signatures, then it is harder, as a coalition of 2/3 malicious nodes can forge any other participant's signature.*

Thanks to River Keefer